FLINT

Fast Library for Number Theory

Version 2.4.5

17 Feb 2015

William Hart*, Fredrik Johansson†, Sebastian Pancratz‡

* EPSRC Grant EP/G004870/1, DFG Priority Program SPP1489
† Supported by Austrian Science Foundation (FWF) Grant Y464-N18
‡ Supported by European Research Council Grant 204083

Contents

1 Introduction 1

2 Configuring FLINT 3

3 TLS, reentrancy and single mode 5

4 ABI and architecture support 7

5 C++ wrapper 9

6 Garbage collection 11

7 Building, testing, installing and using FLINT 13

8 FLINT extension modules 15

9 Test code 17

10 Reporting bugs 19

11 Contributors 21

12 Tuning FLINT 23

13 Example programs 25

14 FLINT macros 27

15 Memory management 29

16 Temporary allocation 31

17 Platform-safe types, format specifiers and constants 33

18 fmpz 35

18.1 Introduction . . . 35
18.2 Simple example . . . 36
18.3 Memory management . . . 36
18.4 Random generation . . . 36
18.5 Conversion . . . 37
18.6 Input and output . . . 39
18.7 Basic properties and manipulation . . . 40
18.8 Comparison . . . 42
18.9 Basic arithmetic . . . 42
18.10 Greatest common divisor . . . 47
18.11 Modular arithmetic . . . 48
18.12 Bit packing and unpacking . . . 48

18.13 Logic Operations . . . 49
18.14 Chinese remaindering . . . 50
18.15 Primality testing . . . 51

19 fmpz_vec **53**
19.1 Memory management . . . 53
19.2 Randomisation . . . 53
19.3 Bit sizes and norms . . . 53
19.4 Input and output . . . 54
19.5 Conversions . . . 54
19.6 Assignment and basic manipulation . . . 55
19.7 Comparison . . . 55
19.8 Sorting . . . 55
19.9 Addition and subtraction . . . 56
19.10 Scalar multiplication and division . . . 56
19.11 Sums and products . . . 58
19.12 Reduction mod p . . . 58
19.13 Gaussian content . . . 58

20 fmpz_factor **59**
20.1 Factoring integers . . . 59

21 fmpz_mat **61**
21.1 Introduction . . . 61
21.2 Simple example . . . 61
21.3 Memory management . . . 62
21.4 Basic assignment and manipulation . . . 62
21.5 Random matrix generation . . . 62
21.6 Input and output . . . 64
21.7 Comparison . . . 65
21.8 Transpose . . . 65
21.9 Modular reduction and reconstruction . . . 65
21.10 Addition and subtraction . . . 66
21.11 Matrix-scalar arithmetic . . . 67
21.12 Matrix multiplication . . . 68
21.13 Inverse . . . 68
21.14 Trace . . . 69
21.15 Determinant . . . 69
21.16 Charactersitic polynomial . . . 70
21.17 Rank . . . 70
21.18 Nonsingular solving . . . 70
21.19 Row reduction . . . 71
21.20 Modular gaussian elimination . . . 72
21.21 Nullspace . . . 72
21.22 Echelon form . . . 72

22 fmpz_poly **75**
22.1 Introduction . . . 75
22.2 Simple example . . . 75
22.3 Definition of the fmpz_poly_t type . . . 76
22.4 Memory management . . . 76
22.5 Polynomial parameters . . . 77
22.6 Assignment and basic manipulation . . . 77
22.7 Randomisation . . . 78
22.8 Getting and setting coefficients . . . 79

22.9 Comparison . . . 80
22.10 Addition and subtraction . . . 80
22.11 Scalar multiplication and division . . . 81
22.12 Bit packing . . . 82
22.13 Multiplication . . . 83
22.14 Squaring . . . 86
22.15 Powering . . . 88
22.16 Shifting . . . 89
22.17 Bit sizes and norms . . . 90
22.18 Greatest common divisor . . . 90
22.19 Gaussian content . . . 93
22.20 Square-free . . . 94
22.21 Euclidean division . . . 94
22.22 Division with precomputed inverse . . . 97
22.23 Divisibility testing . . . 98
22.24 Power series division . . . 99
22.25 Pseudo division . . . 99
22.26 Derivative . . . 101
22.27 Evaluation . . . 102
22.28 Newton basis . . . 103
22.29 Interpolation . . . 103
22.30 Composition . . . 104
22.31 Taylor shift . . . 105
22.32 Power series composition . . . 105
22.33 Power series reversion . . . 106
22.34 Square root . . . 107
22.35 Signature . . . 108
22.36 Hensel lifting . . . 108
22.37 Input and output . . . 111
22.38 Modular reduction and reconstruction . . . 113
22.39 Products . . . 114
22.40 Newton basis conversion . . . 114
22.41 Roots . . . 115

23 fmpz_poly_factor **117**
23.1 Memory management . . . 117
23.2 Manipulating factors . . . 117
23.3 Input and output . . . 118
23.4 Factoring algorithms . . . 118

24 fmpq **121**
24.1 Introduction . . . 121
24.2 Memory management . . . 121
24.3 Canonicalisation . . . 122
24.4 Basic assignment . . . 122
24.5 Comparison . . . 122
24.6 Conversion . . . 123
24.7 Input and output . . . 124
24.8 Random number generation . . . 125
24.9 Arithmetic . . . 125
24.10 Modular reduction and rational reconstruction . . . 127
24.11 Rational enumeration . . . 128
24.12 Continued fractions . . . 129

25 fmpq_mat **131**
25.1 Introduction . . . 131
25.2 Memory management . . . 131
25.3 Entry access . . . 131
25.4 Basic assignment . . . 132
25.5 Addition, scalar multiplication . . . 132
25.6 Input and output . . . 132
25.7 Random matrix generation . . . 133
25.8 Special matrices . . . 133
25.9 Basic comparison and properties . . . 133
25.10 Integer matrix conversion . . . 133
25.11 Modular reduction and rational reconstruction . . . 134
25.12 Matrix multiplication . . . 134
25.13 Trace . . . 135
25.14 Determinant . . . 135
25.15 Nonsingular solving . . . 135
25.16 Inverse . . . 136
25.17 Echelon form . . . 136

26 fmpq_poly **137**
26.1 Introduction . . . 137
26.2 Memory management . . . 137
26.3 Polynomial parameters . . . 139
26.4 Accessing the numerator and denominator . . . 139
26.5 Random testing . . . 139
26.6 Assignment, swap, negation . . . 140
26.7 Getting and setting coefficients . . . 141
26.8 Comparison . . . 142
26.9 Addition and subtraction . . . 143
26.10 Scalar multiplication and division . . . 144
26.11 Multiplication . . . 146
26.12 Powering . . . 146
26.13 Shifting . . . 147
26.14 Euclidean division . . . 147
26.15 Euclidean division . . . 148
26.16 Power series division . . . 149
26.17 Greatest common divisor . . . 149
26.18 Derivative and integral . . . 151
26.19 Square roots . . . 151
26.20 Transcendental functions . . . 151
26.21 Evaluation . . . 154
26.22 Interpolation . . . 155
26.23 Composition . . . 155
26.24 Power series composition . . . 156
26.25 Power series reversion . . . 157
26.26 Gaussian content . . . 158
26.27 Square-free . . . 159
26.28 Input and output . . . 159

27 fmpz_poly_q **161**
27.1 Introduction . . . 161
27.2 Simple example . . . 161
27.3 Memory management . . . 162
27.4 Randomisation . . . 162

27.5 Assignment . . . 163
27.6 Comparison . . . 163
27.7 Addition and subtraction . . . 163
27.8 Scalar multiplication and division . . . 164
27.9 Multiplication and division . . . 164
27.10 Powering . . . 165
27.11 Derivative . . . 165
27.12 Evaluation . . . 165
27.13 Input and output . . . 165

28 fmpz_poly_mat **167**
28.1 Simple example . . . 167
28.2 Memory management . . . 168
28.3 Basic properties . . . 168
28.4 Basic assignment and manipulation . . . 168
28.5 Input and output . . . 168
28.6 Random matrix generation . . . 169
28.7 Special matrices . . . 169
28.8 Basic comparison and properties . . . 169
28.9 Norms . . . 170
28.10 Transpose . . . 170
28.11 Evaluation . . . 170
28.12 Arithmetic . . . 170
28.13 Row reduction . . . 172
28.14 Trace . . . 172
28.15 Determinant and rank . . . 173
28.16 Inverse . . . 173
28.17 Nullspace . . . 173
28.18 Solving . . . 173

29 nmod_vec **175**
29.1 Memory management . . . 175
29.2 Modular reduction and arithmetic . . . 175
29.3 Random functions . . . 176
29.4 Basic manipulation and comparison . . . 176
29.5 Arithmetic operations . . . 177
29.6 Dot products . . . 177

30 nmod_poly **179**
30.1 Introduction . . . 179
30.2 Simple example . . . 179
30.3 Definition of the nmod_poly_t type . . . 180
30.4 Helper functions . . . 180
30.5 Memory management . . . 180
30.6 Polynomial properties . . . 181
30.7 Assignment and basic manipulation . . . 181
30.8 Randomization . . . 182
30.9 Getting and setting coefficients . . . 182
30.10 Input and output . . . 183
30.11 Comparison . . . 183
30.12 Shifting . . . 184
30.13 Addition and subtraction . . . 184
30.14 Scalar multiplication and division . . . 185
30.15 Bit packing and unpacking . . . 185

30.16 KS2/KS4 Reduction . . . 186
30.17 Multiplication . . . 187
30.18 Powering . . . 189
30.19 Division . . . 191
30.20 Derivative and integral . . . 196
30.21 Evaluation . . . 197
30.22 Multipoint evaluation . . . 197
30.23 Interpolation . . . 198
30.24 Composition . . . 199
30.25 Taylor shift . . . 200
30.26 Modular composition . . . 200
30.27 Greatest common divisor . . . 202
30.28 Power series composition . . . 206
30.29 Power series reversion . . . 207
30.30 Square roots . . . 209
30.31 Transcendental functions . . . 210
30.32 Products . . . 213
30.33 Subproduct trees . . . 213
30.34 Inflation and deflation . . . 214

31 nmod_poly_factor **215**
31.1 Factorisation . . . 215

32 nmod_mat **219**
32.1 Introduction . . . 219
32.2 Memory management . . . 219
32.3 Basic properties and manipulation . . . 220
32.4 Printing . . . 220
32.5 Random matrix generation . . . 220
32.6 Comparison . . . 221
32.7 Transpose . . . 221
32.8 Addition and subtraction . . . 221
32.9 Matrix-scalar arithmetic . . . 221
32.10 Matrix multiplication . . . 222
32.11 Trace . . . 222
32.12 Determinant and rank . . . 222
32.13 Inverse . . . 222
32.14 Triangular solving . . . 223
32.15 Nonsingular square solving . . . 224
32.16 LU decomposition . . . 224
32.17 Reduced row echelon form . . . 224
32.18 Nullspace . . . 225

33 nmod_poly_mat **227**
33.1 Memory management . . . 227
33.2 Basic properties . . . 227
33.3 Basic assignment and manipulation . . . 228
33.4 Input and output . . . 228
33.5 Random matrix generation . . . 228
33.6 Special matrices . . . 228
33.7 Basic comparison and properties . . . 229
33.8 Norms . . . 229
33.9 Evaluation . . . 229
33.10 Arithmetic . . . 229

33.11 Row reduction . . . 231
33.12 Trace . . . 232
33.13 Determinant and rank . . . 232
33.14 Inverse . . . 232
33.15 Nullspace . . . 232
33.16 Solving . . . 233

34 fmpz_mod_poly **235**
34.1 Introduction . . . 235
34.2 Simple example . . . 235
34.3 Definition of the fmpz_mod_poly_t type . . . 236
34.4 Memory management . . . 236
34.5 Randomisation . . . 237
34.6 Attributes . . . 238
34.7 Assignment and basic manipulation . . . 238
34.8 Conversion . . . 239
34.9 Comparison . . . 239
34.10 Getting and setting coefficients . . . 239
34.11 Shifting . . . 240
34.12 Addition and subtraction . . . 240
34.13 Scalar multiplication . . . 241
34.14 Multiplication . . . 241
34.15 Powering . . . 242
34.16 Division . . . 244
34.17 Power series inversion . . . 248
34.18 Greatest common divisor . . . 248
34.19 Derivative . . . 251
34.20 Evaluation . . . 251
34.21 Multipoint evaluation . . . 251
34.22 Composition . . . 252
34.23 Modular composition . . . 253
34.24 Subproduct trees . . . 255
34.25 Radix conversion . . . 256
34.26 Input and output . . . 257

35 fmpz_mod_poly_factor **259**
35.1 Factorisation . . . 259

36 fq **263**
36.1 Context Management . . . 263
36.2 Memory management . . . 264
36.3 Basic arithmetic . . . 265
36.4 Roots . . . 266
36.5 Output . . . 266
36.6 Randomisation . . . 267
36.7 Assignments and conversions . . . 267
36.8 Comparison . . . 268
36.9 Special functions . . . 268
36.10 Bit packing . . . 269

37 fq_vec **271**
37.1 Memory management . . . 271
37.2 Randomisation . . . 271
37.3 Input and output . . . 271
37.4 Assignment and basic manipulation . . . 271

37.5 Comparison . . . 272
37.6 Addition and subtraction . . . 272
37.7 Scalar multiplication and division . . . 272
37.8 Dot products . . . 272

38 fq_mat **275**
38.1 Memory management . . . 275
38.2 Basic properties and manipulation . . . 275
38.3 Printing . . . 276
38.4 Window . . . 276
38.5 Random matrix generation . . . 276
38.6 Comparison . . . 277
38.7 Addition and subtraction . . . 278
38.8 Matrix multiplication . . . 278
38.9 LU decomposition . . . 278
38.10 Reduced row echelon form . . . 279
38.11 Triangular solving . . . 279

39 fq_poly **281**
39.1 Memory management . . . 281
39.2 Polynomial parameters . . . 282
39.3 Randomisation . . . 283
39.4 Assignment and basic manipulation . . . 283
39.5 Getting and setting coefficients . . . 284
39.6 Comparison . . . 284
39.7 Addition and subtraction . . . 285
39.8 Scalar multiplication and division . . . 285
39.9 Multiplication . . . 286
39.10 Squaring . . . 288
39.11 Powering . . . 289
39.12 Shifting . . . 291
39.13 Norms . . . 292
39.14 Euclidean division . . . 292
39.15 Greatest common divisor . . . 295
39.16 Divisibility testing . . . 295
39.17 Derivative . . . 296
39.18 Evaluation . . . 296
39.19 Composition . . . 296
39.20 Output . . . 300
39.21 Inflation and deflation . . . 301

40 fq_poly_factor **303**
40.1 Memory Management . . . 303
40.2 Basic Operations . . . 303
40.3 Irreducibility Testing . . . 304
40.4 Factorisation . . . 305

41 fq_nmod **307**
41.1 Context Management . . . 307
41.2 Memory management . . . 308
41.3 Basic arithmetic . . . 309
41.4 Roots . . . 310
41.5 Output . . . 310
41.6 Randomisation . . . 311
41.7 Assignments and conversions . . . 311

41.8 Comparison . . . 312
41.9 Special functions . . . 312
41.10 Bit packing . . . 313

42 fq_nmod_vec **315**
42.1 Memory management . . . 315
42.2 Randomisation . . . 315
42.3 Input and output . . . 315
42.4 Assignment and basic manipulation . . . 315
42.5 Comparison . . . 316
42.6 Addition and subtraction . . . 316
42.7 Scalar multiplication and division . . . 316
42.8 Dot products . . . 317

43 fq_nmod_mat **319**
43.1 Memory management . . . 319
43.2 Basic properties and manipulation . . . 319
43.3 Printing . . . 320
43.4 Window . . . 320
43.5 Random matrix generation . . . 321
43.6 Comparison . . . 321
43.7 Addition and subtraction . . . 322
43.8 Matrix multiplication . . . 322
43.9 LU decomposition . . . 323
43.10 Reduced row echelon form . . . 323
43.11 Triangular solving . . . 323

44 fq_nmod_poly **325**
44.1 Memory management . . . 325
44.2 Polynomial parameters . . . 326
44.3 Randomisation . . . 327
44.4 Assignment and basic manipulation . . . 327
44.5 Getting and setting coefficients . . . 328
44.6 Comparison . . . 328
44.7 Addition and subtraction . . . 329
44.8 Scalar multiplication and division . . . 329
44.9 Multiplication . . . 330
44.10 Squaring . . . 333
44.11 Powering . . . 334
44.12 Shifting . . . 336
44.13 Norms . . . 336
44.14 Euclidean division . . . 336
44.15 Greatest common divisor . . . 339
44.16 Divisibility testing . . . 340
44.17 Derivative . . . 340
44.18 Evaluation . . . 341
44.19 Composition . . . 341
44.20 Output . . . 345
44.21 Inflation and deflation . . . 346

45 fq_nmod_poly_factor **349**
45.1 Memory Management . . . 349
45.2 Basic Operations . . . 349
45.3 Irreducibility Testing . . . 350
45.4 Factorisation . . . 351

46 fq_zech **353**
46.1 Context Management . . . 353
46.2 Memory management . . . 354
46.3 Basic arithmetic . . . 355
46.4 Roots . . . 356
46.5 Output . . . 356
46.6 Randomisation . . . 357
46.7 Assignments and conversions . . . 357
46.8 Comparison . . . 358
46.9 Special functions . . . 359
46.10 Bit packing . . . 359

47 fq_zech_vec **361**
47.1 Memory management . . . 361
47.2 Randomisation . . . 361
47.3 Input and output . . . 361
47.4 Assignment and basic manipulation . . . 361
47.5 Comparison . . . 362
47.6 Addition and subtraction . . . 362
47.7 Scalar multiplication and division . . . 362
47.8 Dot products . . . 363

48 fq_zech_mat **365**
48.1 Memory management . . . 365
48.2 Basic properties and manipulation . . . 365
48.3 Printing . . . 366
48.4 Window . . . 366
48.5 Random matrix generation . . . 367
48.6 Comparison . . . 367
48.7 Addition and subtraction . . . 368
48.8 Matrix multiplication . . . 368
48.9 LU decomposition . . . 369
48.10 Reduced row echelon form . . . 369
48.11 Triangular solving . . . 369

49 fq_zech_poly **371**
49.1 Memory management . . . 371
49.2 Polynomial parameters . . . 372
49.3 Randomisation . . . 373
49.4 Assignment and basic manipulation . . . 373
49.5 Getting and setting coefficients . . . 374
49.6 Comparison . . . 374
49.7 Addition and subtraction . . . 375
49.8 Scalar multiplication and division . . . 375
49.9 Multiplication . . . 376
49.10 Squaring . . . 379
49.11 Powering . . . 380
49.12 Shifting . . . 382
49.13 Norms . . . 382
49.14 Euclidean division . . . 382
49.15 Greatest common divisor . . . 385
49.16 Divisibility testing . . . 386
49.17 Derivative . . . 386
49.18 Evaluation . . . 387

49.19 Composition . . . 387
49.20 Output . . . 391
49.21 Inflation and deflation . . . 392

50 fq_zech_poly_factor **395**
50.1 Memory Management . . . 395
50.2 Basic Operations . . . 395
50.3 Irreducibility Testing . . . 396
50.4 Factorisation . . . 397

51 padic **399**
51.1 Introduction . . . 399
51.2 Data structures . . . 399
51.3 Context . . . 400
51.4 Memory management . . . 400
51.5 Randomisation . . . 401
51.6 Assignments and conversions . . . 401
51.7 Comparison . . . 403
51.8 Arithmetic operations . . . 403
51.9 Exponential . . . 404
51.10 Logarithm . . . 405
51.11 Special functions . . . 407
51.12 Input and output . . . 407

52 padic_mat **409**
52.1 Module documentation . . . 409
52.2 Macros . . . 409
52.3 Memory management . . . 409
52.4 Basic assignment . . . 410
52.5 Conversions . . . 410
52.6 Entries . . . 411
52.7 Comparison . . . 411
52.8 Input and output . . . 411
52.9 Random matrix generation . . . 412
52.10 Transpose . . . 412
52.11 Addition and subtraction . . . 412
52.12 Scalar operations . . . 412
52.13 Multiplication . . . 413

53 padic_poly **415**
53.1 Module documentation . . . 415
53.2 Memory management . . . 415
53.3 Polynomial parameters . . . 416
53.4 Randomisation . . . 417
53.5 Assignment and basic manipulation . . . 417
53.6 Getting and setting coefficients . . . 418
53.7 Comparison . . . 418
53.8 Addition and subtraction . . . 419
53.9 Scalar multiplication . . . 419
53.10 Multiplication . . . 420
53.11 Powering . . . 420
53.12 Series inversion . . . 420
53.13 Derivative . . . 421
53.14 Shifting . . . 421
53.15 Evaluation . . . 421

53.16 Composition . . . 421
53.17 Input and output . . . 422
53.18 Testing . . . 423

54 qadic **425**
54.1 Data structures . . . 425
54.2 Context . . . 425
54.3 Memory management . . . 426
54.4 Properties . . . 426
54.5 Randomisation . . . 426
54.6 Assignments and conversions . . . 427
54.7 Comparison . . . 427
54.8 Basic arithmetic . . . 428
54.9 Special functions . . . 429
54.10 Output . . . 433

55 arith **435**
55.1 Introduction . . . 435
55.2 Primorials . . . 435
55.3 Harmonic numbers . . . 435
55.4 Stirling numbers . . . 435
55.5 Bell numbers . . . 437
55.6 Bernoulli numbers and polynomials . . . 438
55.7 Euler numbers and polynomials . . . 440
55.8 Legendre polynomials . . . 441
55.9 Multiplicative functions . . . 441
55.10 Cyclotomic polynomials . . . 442
55.11 Swinnerton-Dyer polynomials . . . 443
55.12 Landau's function . . . 443
55.13 Dedekind sums . . . 444
55.14 Number of partitions . . . 445
55.15 Sums of squares . . . 447
55.16 MPFR extras . . . 447

56 ulong_extras **449**
56.1 Introduction . . . 449
56.2 Simple example . . . 449
56.3 Random functions . . . 450
56.4 Basic arithmetic . . . 451
56.5 Miscellaneous . . . 451
56.6 Basic arithmetic with precomputed inverses . . . 451
56.7 Greatest common divisor . . . 453
56.8 Jacobi and Kronecker symbols . . . 454
56.9 Modular Arithmetic . . . 454
56.10 Prime number generation and counting . . . 456
56.11 Primality testing . . . 458
56.12 Square root and perfect power testing . . . 460
56.13 Factorisation . . . 461
56.14 Arithmetic functions . . . 464
56.15 Factorials . . . 464
56.16 Primitive Roots and Discrete Logarithms . . . 465

57 long_extras **467**
57.1 Properties . . . 467
57.2 Random functions . . . 467

58 fft **469**
58.1 Split/combine FFT coefficients . . . 469
58.2 Test helper functions . . . 470
58.3 Arithmetic modulo a generalised Fermat number . . . 470
58.4 Generic butterflies . . . 470
58.5 Radix 2 transforms . . . 471
58.6 Matrix Fourier Transforms . . . 474
58.7 Negacyclic multiplication . . . 476
58.8 Integer multiplication . . . 477
58.9 Convolution . . . 478

59 qsieve **479**
59.1 Quadratic sieve . . . 479

60 perm **481**
60.1 Memory management . . . 481
60.2 Assignment . . . 481
60.3 Composition . . . 481
60.4 Parity . . . 481
60.5 Randomisation . . . 482
60.6 Input and output . . . 482

61 longlong.h **483**
61.1 Auxiliary asm macros . . . 483

62 mpn_extras **485**
62.1 Macros . . . 485
62.2 Utility functions . . . 485
62.3 Divisibility . . . 485
62.4 Division . . . 486
62.5 GCD . . . 487
62.6 Special numbers . . . 487
62.7 Random Number Generation . . . 488

63 flintxx **489**
63.1 Introduction . . . 489
63.2 Overview . . . 490
63.3 Notations and conventions for the C++ interface documentation . . . 492
63.4 flint_exception . . . 494
63.5 frandxx . . . 494
63.6 ltuple . . . 494
63.7 permxx . . . 494
63.8 fmpzxx . . . 495
63.8.1 C++ particulars . . . 495
63.8.2 Memory management . . . 496
63.8.3 Random generation . . . 496
63.8.4 Conversion . . . 496
63.8.5 Input and output . . . 497
63.8.6 Basic properties and manipulation . . . 497
63.8.7 Comparision . . . 497
63.8.8 Basic arithmetic . . . 498
63.8.9 Greatest common divisor . . . 500
63.8.10 Modular arithmetic . . . 500
63.8.11 Bit packing and unpacking . . . 500
63.8.12 Logic operations . . . 500

63.8.13 Chinese remaindering . . . 501
63.8.14 Primality testing . . . 501
63.9 fmpz_factorxx . . . 501
63.10 fmpz_matxx . . . 502
63.10.1 Not yet split into subsections . . . 503
63.10.2 C++ particulars . . . 503
63.10.3 Memory management . . . 503
63.10.4 Basic assignment and manipulation . . . 503
63.10.5 Input and output . . . 503
63.10.6 Comparison . . . 504
63.10.7 Conversion . . . 504
63.10.8 Randomisation . . . 504
63.10.9 Transpose . . . 505
63.10.10 Modular reduction and reconstruction . . . 506
63.10.11 Arithmetic . . . 506
63.10.12 Inverse . . . 507
63.10.13 Trace . . . 507
63.10.14 Determinant . . . 507
63.10.15 Characteristic polynomial . . . 507
63.10.16 Rank . . . 507
63.10.17 Non-singular solving . . . 507
63.10.18 Row reduction . . . 508
63.10.19 Modular gaussian elimination . . . 508
63.10.20 Nullspace . . . 508
63.11 fmpz_polyxx . . . 508
63.11.1 C++ particulars . . . 508
63.11.2 Memory management . . . 509
63.11.3 Polynomial parameters . . . 509
63.11.4 Assignment and basic manipulation . . . 509
63.11.5 Randomisation . . . 510
63.11.6 Getting and setting coefficients . . . 510
63.11.7 Comparison . . . 511
63.11.8 Addition and subtraction . . . 511
63.11.9 Scalar multiplication and division . . . 511
63.11.10 Bit packing . . . 511
63.11.11 Multiplication . . . 512
63.11.12 Squaring . . . 512
63.11.13 Powering . . . 513
63.11.14 Shifting . . . 513
63.11.15 Bit sizes and norms . . . 513
63.11.16 Greatest common divisor . . . 513
63.11.17 Gaussian content . . . 514
63.11.18 Square-free . . . 514
63.11.19 Euclidean division . . . 514
63.11.20 Divisibility testing . . . 514
63.11.21 Power series division . . . 514
63.11.22 Pseudo division . . . 515
63.11.23 Derivative . . . 515
63.11.24 Evaluation . . . 515
63.11.25 Interpolation . . . 516
63.11.26 Composition. . . . 516
63.11.27 Taylor shift . . . 516
63.11.28 Power series composition . . . 516
63.11.29 Power series reversion . . . 516
63.11.30 Square root . . . 517
63.11.31 Signature . . . 517

63.11.32 Hensel lifting . . . 517
63.11.33 Input and output . . . 517
63.11.34 Modular reduction and reconstruction . . . 518
63.11.35 Products . . . 518
63.11.36 Roots . . . 518
63.12 fmpz_poly_factorxx . . . 518
63.12.1 Memory management . . . 518
63.12.2 Manipulating factors . . . 519
63.12.3 Factoring algorithms . . . 519
63.13 fmpqxx . . . 519
63.13.1 C++ particulars . . . 519
63.13.2 Memory management . . . 519
63.13.3 Canonicalisation . . . 520
63.13.4 Basic assignment . . . 520
63.13.5 Comparison . . . 520
63.13.6 Conversion . . . 520
63.13.7 Input and output . . . 521
63.13.8 Random number generation . . . 521
63.13.9 Arithmetic . . . 521
63.13.10 Modular reduction and rational reconstruction . . . 521
63.13.11 Rational enumeration . . . 522
63.13.12 Continued fractions . . . 522
63.14 fmpq_matxx . . . 522
63.14.1 Memory management . . . 523
63.14.2 Input and output . . . 523
63.14.3 Entry access . . . 523
63.14.4 Basic assignment . . . 523
63.14.5 Random matrix generation . . . 523
63.14.6 Special matrices . . . 523
63.14.7 Basic properties . . . 523
63.14.8 Integer matrix conversion . . . 524
63.14.9 Modular reduction and rational reconstruction . . . 524
63.14.10 Matrix multiplication . . . 524
63.14.11 Trace . . . 525
63.14.12 Determinant . . . 525
63.14.13 Nonsingular solving . . . 525
63.14.14 Inverse . . . 525
63.14.15 Echelon form . . . 525
63.15 fmpq_polyxx . . . 525
63.15.1 C++ particulars . . . 525
63.15.2 Memory management . . . 526
63.15.3 Polynomial parameters . . . 526
63.15.4 Accessing the numerator and denominator . . . 526
63.15.5 Random testing . . . 526
63.15.6 Assignment . . . 527
63.15.7 Getting and setting coefficients . . . 527
63.15.8 Comparison . . . 527
63.15.9 Arithmetic . . . 528
63.15.10 Powering . . . 528
63.15.11 Shifting . . . 528
63.15.12 Euclidean division . . . 528
63.15.13 Power series division . . . 528
63.15.14 Greatest common divisor . . . 528
63.15.15 Derivative and integral . . . 529
63.15.16 Square roots . . . 529
63.15.17 Transcendental functions . . . 529

63.15.18 Evaluation . . . 529
63.15.19 Interpolation . . . 530
63.15.20 Composition . . . 530
63.15.21 Power series composition . . . 530
63.15.22 Power series reversion . . . 530
63.15.23 Gaussian content . . . 530
63.15.24 Square-free . . . 530
63.15.25 Input and output . . . 530
63.16 fmpz_poly_qxx . . . 531
63.16.1 Memory management . . . 531
63.16.2 Randomisation . . . 531
63.16.3 Assignment . . . 531
63.16.4 Comparison . . . 532
63.16.5 Powering . . . 532
63.16.6 Derivative . . . 532
63.16.7 Input and output . . . 532
63.17 fmpz_poly_matxx . . . 532
63.17.1 Input and output . . . 533
63.17.2 Basic properties . . . 533
63.17.3 Basic assignment and manipulation . . . 533
63.17.4 Standard matrices . . . 533
63.17.5 Random matrix generation . . . 533
63.17.6 Basic comparison and properties . . . 534
63.17.7 Norms . . . 534
63.17.8 Transpose . . . 534
63.17.9 Arithmetic . . . 534
63.17.10 Row reduction . . . 535
63.17.11 Trace . . . 535
63.17.12 Determinant and rank . . . 535
63.17.13 Inverse . . . 535
63.17.14 Nullspace . . . 535
63.17.15 Solving . . . 536
63.18 nmodxx . . . 536
63.19 nmod_polyxx . . . 538
63.19.1 Conversion . . . 539
63.19.2 Input and output . . . 539
63.19.3 Memory management . . . 540
63.19.4 Polynomial properties . . . 540
63.19.5 Assignment and basic manipulation . . . 540
63.19.6 Randomisation . . . 540
63.19.7 Getting and setting coefficients . . . 541
63.19.8 Input and output . . . 541
63.19.9 Comparison . . . 541
63.19.10 Scalar multiplication and division . . . 541
63.19.11 Bit packing and unpacking . . . 541
63.19.12 Multiplication . . . 541
63.19.13 Powering . . . 542
63.19.14 Division . . . 542
63.19.15 Derivative and integral . . . 543
63.19.16 Evaluation . . . 543
63.19.17 Interpolation . . . 544
63.19.18 Composition . . . 544
63.19.19 Taylor Shift . . . 544
63.19.20 Modular composition . . . 544
63.19.21 Greatest common divisor . . . 545
63.19.22 Power series composition . . . 545

63.19.23 Power series reversion . . . 545
63.19.24 Square roots . . . 545
63.19.25 Transcendental functions . . . 546
63.19.26 Products . . . 546
63.19.27 Inflation and deflation . . . 546
63.19.28 Factorisation . . . 546
63.20 nmod_matxx . . . 548
63.20.1 Conversion . . . 549
63.20.2 Input and output . . . 549
63.20.3 Memory management . . . 549
63.20.4 Basic properties and manipulation . . . 549
63.20.5 Random matrix generation . . . 550
63.20.6 Transpose . . . 550
63.20.7 Matrix multiplication . . . 550
63.20.8 Trace . . . 551
63.20.9 Determinant and rank . . . 551
63.20.10 Inverse . . . 551
63.20.11 Triangular solving . . . 551
63.20.12 Non-singular square solving . . . 551
63.20.13 LU decomposition . . . 551
63.20.14 Reduced row echelon form . . . 552
63.20.15 Nullspace . . . 552
63.21 nmod_poly_matxx . . . 552
63.21.1 Input and output . . . 552
63.21.2 Memory management . . . 552
63.21.3 Basic assignment and manipulation . . . 553
63.21.4 Standard matrices . . . 553
63.21.5 Random matrix generation . . . 553
63.21.6 Basic comparison and properties . . . 553
63.21.7 Norms . . . 554
63.21.8 Arithmetic . . . 554
63.21.9 Row reduction . . . 554
63.21.10 Transpose . . . 554
63.21.11 Trace . . . 554
63.21.12 Determinant and rank . . . 554
63.21.13 Inverse . . . 555
63.21.14 Nullspace . . . 555
63.21.15 Solving . . . 555
63.22 fmpz_mod_polyxx . . . 555
63.22.1 Input and output . . . 556
63.22.2 Memory management . . . 556
63.22.3 Randomisation . . . 556
63.22.4 Attributes . . . 557
63.22.5 Assignment and swap . . . 557
63.22.6 Conversion . . . 557
63.22.7 Comparison . . . 557
63.22.8 Getting and setting coefficients . . . 557
63.22.9 Shifting . . . 558
63.22.10 Multiplication . . . 558
63.22.11 Powering . . . 558
63.22.12 Division . . . 558
63.22.13 Power series inversion . . . 559
63.22.14 Greatest common divisor . . . 559
63.22.15 Derivative . . . 560
63.22.16 Evaluation . . . 560
63.22.17 Composition . . . 560

63.22.18 Modular composition . . . 560
63.22.19 Radix conversion . . . 560
63.23 fmpz_mod_poly_factorxx . . . 561
63.24 padicxx . . . 562
63.24.1 Context . . . 563
63.24.2 C++ particulars . . . 563
63.24.3 Input and output . . . 563
63.24.4 Data structures . . . 563
63.24.5 Memory management . . . 564
63.24.6 Randomisation . . . 564
63.24.7 Conversion . . . 564
63.24.8 Arithmetic operations . . . 565
63.24.9 Exponential . . . 565
63.24.10 Logarithm . . . 565
63.24.11 Special functions . . . 566
63.25 padic_polyxx . . . 566
63.25.1 C++ particulars . . . 566
63.25.2 Input and output . . . 566
63.25.3 Memory management . . . 566
63.25.4 Polynomial parameters . . . 567
63.25.5 Randomisation . . . 567
63.25.6 Assignment an basic manipulation . . . 567
63.25.7 Getting and setting coefficients . . . 568
63.25.8 Comparison . . . 568
63.25.9 Arithmetic . . . 569
63.25.10 Powering . . . 569
63.25.11 Series inversion . . . 569
63.25.12 Derivative . . . 569
63.25.13 Shifting . . . 569
63.25.14 Evaluation and composition . . . 569
63.25.15 Testing . . . 569
63.26 padic_matxx . . . 569
63.26.1 C++ particulars . . . 570
63.26.2 Input and output . . . 570
63.26.3 Memory management . . . 570
63.26.4 Basic assignment . . . 571
63.26.5 Conversion . . . 571
63.26.6 Entries . . . 571
63.26.7 Comparison . . . 571
63.26.8 Random matrix generation . . . 571
63.26.9 Transpose . . . 571
63.26.10 Arithmetic . . . 571
63.27 qadicxx . . . 572
63.27.1 Context . . . 572
63.27.2 C++ particulars . . . 572
63.27.3 Data structures . . . 572
63.27.4 Data structures . . . 573
63.27.5 Memory management . . . 573
63.27.6 Randomisation . . . 573
63.27.7 Conversion . . . 573
63.27.8 Arithmetic operations . . . 574
63.27.9 Exponential . . . 574
63.27.10 Logarithm . . . 574
63.27.11 Special functions . . . 574
63.28 arithxx . . . 575
63.28.1 Primorials . . . 575

63.28.2 Harmonic numbers . . . 575
63.28.3 Stirling numbers . . . 575
63.28.4 Bell numbers . . . 576
63.28.5 Bernoulli numbers and polynomials . . . 576
63.28.6 Euler numbers and polynomials . . . 576
63.28.7 Legendre polynomials . . . 577
63.28.8 Multiplicative functions . . . 577
63.28.9 Cyclotomic polynomials . . . 577
63.28.10 Swinnerton-Dyer polynomials . . . 577
63.28.11 Landau's function . . . 577
63.28.12 Dedekind sums . . . 577
63.28.13 Number of partitions . . . 577
63.28.14 Sums of squares . . . 578

64 profiler **579**
64.1 Timer based on the cycle counter . . . 579
64.2 Framework for repeatedly sampling a single target . . . 580
64.3 Memory usage . . . 580
64.4 Simple profiling macros . . . 581

65 interfaces **583**
65.1 Introduction . . . 583
65.2 NTL Interface . . . 583

A Extending the C++ wrapper **585**
A.1 Introduction . . . 585
A.2 Overview of flintxx . . . 585
A.2.1 Patterns for implementing unified coefficient access . . . 586
A.2.2 Nmod classes . . . 587
A.2.3 Matrix classes . . . 588
A.2.4 Padic classes . . . 591
A.2.5 Vector classes . . . 592
A.3 Some tidbits and caveats . . . 592
A.4 Rules and standard methods . . . 594
A.4.1 Standard methods . . . 595
A.4.2 Global functions . . . 596
A.4.3 flintxx classes . . . 597
A.5 Convenience macros . . . 597
A.5.1 flintxx/rules.h . . . 597
A.5.2 flintxx/expression.h . . . 599
A.5.3 flintxx/flint_classes.h . . . 600
A.5.4 flintxx/matrix.h . . . 601
A.6 Helper functions . . . 601
A.6.1 flintxx/flint_exception.h . . . 601
A.6.2 permxx.h . . . 601

References **603**

§1. Introduction

FLINT is a C library of functions for doing number theory. It is highly optimised and can be compiled on numerous platforms. FLINT also has the aim of providing support for multicore and multiprocessor computer architectures. To this end, the library is threadsafe, with few exceptions noted in the appropriate place.

FLINT is currently maintained by William Hart of Technishe Universität in Kaiserslautern. FLINT was originally designed by William Hart and David Harvey. Since then FLINT was rewritten as FLINT 2 by William Hart, Fredrik Johansson and Sebastian Pancratz. Many other substantial contributions have been made by other authors. (See the Contributors list below for a list.)

FLINT 2 and following should compile on any machine with GCC and a standard GNU toolchain, however it is specially optimised for x86 (32 and 64 bit) machines. There is also limited optimisation for ARM and ia64 machines. As of version 2.0, FLINT required GCC version 2.96 or later, either MPIR (2.6.0 or later) or GMP (5.1.1 or later), and MPFR 3.0.0 or later. It is also required that the platform provide a `uint64_t` type if a native 64 bit type is not available. Full C99 compliance is **not** required.

FLINT is supplied as a set of modules, `fmpz`, `fmpz_poly`, etc., each of which can be linked to a C program making use of their functionality.

All of the functions in FLINT have a corresponding test function provided in an appropriately named test file. For example, the function `fmpz_poly_add` located in `fmpz_poly/add.c` has test code in the file `fmpz_poly/test/t-add.c`.

§2. Configuring FLINT

The easiest way to use FLINT is to build a shared library. Simply download the FLINT tarball and untar it on your system.

FLINT requires either MPIR (version 2.6.0 or later) or GMP (version 5.1.1 or later). If MPIR is used, MPIR must be built with the `--enable-gmpcompat` option. FLINT also requires MPFR 3.0.0 or later and a pthread implementation. Some of the input/output tests require `fork` and `pipe`, however these are disabled on MinGW which does not provide a posix implementation.

To configure FLINT you must specify where GMP/MPIR and MPFR are on your system. FLINT can work with the libraries installed as usual, e.g. in `/usr/local` or it can work with the libraries built from source in their standard source trees.

In the case that a library is installed in say `/usr` in the `lib` and `include` directories as usual, simply specify the top level location, e.g. `/usr` when configuring FLINT. If a library is built in its source tree, specify the top level directory, e.g. `/home/user1/mpir/`.

To specify the directories where the libraries reside, you must pass the directories as parameters to FLINT's configure, e.g.

```
./configure --with-mpir=/usr --with-mpfr=/home/user1/mpfr/
```

If no directories are specified, FLINT assumes it will find the libraries it needs in `/usr/local`.

If you wish to use GMP, you can pass the configure option `--with-gmp` instead of `--with-mpir`.

Note that FLINT builds static and shared libraries by default, except on platforms where this is not supported. If you do not require either a shared or static library then you may pass `--disable-static` or `--disable-shared` to `configure`.

If you intend to install the FLINT library and header files, you can specify where they should be placed by passing `--prefix=path` to configure, where `path` is the directory under which the `lib` and `include` directories exist into which you wish to place the FLINT files when it is installed.

§3. TLS, reentrancy and single mode

If you wish to use FLINT on a single core machine then it can be configured for single mode. This mode can also be explicitly selected by passing the `--single` option to configure. Single mode is slightly faster, but by default uses thread local storage if threads are used, and this is not available on some machines.

FLINT uses thread local storage by default (`--enable-tls`). However, if reentrancy is required on systems that do not support this, one can pass `--disable-tls` and mutexes will be used instead (requires POSIX).

If you wish to build a threadsafe version of FLINT which uses a less complicated memory model (slower, but still works in the absence of TLS) you can pass the `--reentrant` option to configure.

§4. ABI and architecture support

On some systems, e.g. Sparc and some Macs, more than one ABI is available. FLINT chooses the ABI based on the CPU type available, however its default choice can be overridden by passing either `ABI=64` or `ABI=32` to configure.

To build on MinGW64 it is necessary to pass `ABI=64` to configure, as FLINT is otherwise unable to distinguish it from MinGW32.

In some cases, it is necessary to override the entire CPU/OS defaults. This can be done by passing `--build=cpu-os` to configure. The available choices for CPU include `x86_64`, `x86`, `ia64`, `sparc`, `sparc64`, `ppc`, `ppc64`. Other CPU types are unrecognised and FLINT will build with generic code on those machines. The choices for OS include `Linux`, `MINGW32`, `MINGW64`, `CYGWIN32`, `CYGWIN64`, `Darwin`, `FreeBSD`, `SunOS` and numerous other operating systems.

It is also possible to override the default CC, AR and CFLAGS used by FLINT by passing `CC=full_path_to_compiler`, etc., to FLINT's configure.

§5. C++ wrapper

If you wish to enable the test functions for the FLINT C++ wrapper `flintxx` you must pass `--enable-cxx` to configure. The `C++` wrapper is always available, but tests will only run if this option is selected. It is disabled by default (`--disable-cxx`) because some `C++` compilers internally segfault when compiling the tests, or exhaust memory due to the complexity of the `C++` code.

§6. Garbage collection

If building FLINT as part of an application that uses the Boehm-Demers-Weiser GC library, you may wish to pass the `--with-gc=<path/to/gc/` option to configure. This causes FLINT to use `GC_malloc` and friends for its memory allocation strategy. The user is responsible for calling `GC_init`.

The `--with-gc` option can be used with `--reentrant`, but not the `--enable-tls` option. In fact, configure will switch this off automatically and print a warning if it is selected in addition to `--with-gc`.

§7. Building, testing, installing and using FLINT

Once FLINT is configured, in the main directory of the FLINT directory tree simply type:

```
make
make check
```

GNU make is required to build FLINT. This is simply `make` on Linux, Darwin, MinGW and Cygwin systems. However, on some unixes the command is `gmake`.

If you wish to install FLINT, simply type:

```
make install
```

Now to use FLINT, simply include the appropriate header files for the FLINT modules you wish to use in your C program. Then compile your program, linking against the FLINT library, GMP/MPIR, MPFR and pthreads with the options `-lflint -lmpfr -lgmp -lpthread`.

Note that you may have to set `LD_LIBRARY_PATH` or equivalent for your system to let the linker know where to find these libraries. Please refer to your system documentation for how to do this.

If you have any difficulties with conflicts with system headers on your machine, you can do the following in your code:

```
#undef ulong
#define ulong ulongxx
#include <stdio.h>
// other system headers
#undef ulong
#define ulong mp_limb_t
```

This prevents FLINT's definition of `ulong` interfering with your system headers.

The FLINT make system responds to the standard commands

```
make
make library
make check
make clean
make distclean
make install
```

In addition, if you wish to simply check a single module of FLINT you can pass the option `MOD=modname` to `make check`. You can also pass a list of module names in inverted commas, e.g:

```
make check MOD=ulong_extras
```

FLINT has an assert system. If you want a debug build you can pass `--enable-assert` to configure. However, this will slow FLINT considerably, so asserts should not be enabled (`--disable-assert`, the default) for deployment.

If your system supports parallel builds, FLINT will build in parallel, e.g:

```
make -j4 check
```

Note that on some systems, most notably MinGW, parallel make is supported but can be problematic.

§8. FLINT extension modules

Numerous developers have been building libraries on top of FLINT, extending its functionality. These projects are known as FLINT extension modules.

To build a FLINT extension module as part of FLINT, do the following:

• Download the flint extension module and place it somewhere in your file system.

• Pass `--extensions=/path/to/extension` to FLINT's configure, or if more than one extension is desired use `--extensions="/path/to/extension1 /path/to/extension2"`, etc.

Now most of the options that are available for FLINT are also available for those extension modules.

Some examples of FLINT extension modules include:

• Arb (by Fredrik Johansson) – Arbitrary precision floating point ball arithmetic with rigorous error bounds, over the real and complex numbers (including polynomials, matrices, calculus and special functions). http://fredrikj.net/arb/

• ANTIC (by William Hart and Claus Fieker) – Algebraic Number Theory in C. Includes general number field arithmetic and class group computation. https://github.com/wbhart/antic

• Bland (by Fredrik Johansson) – Generic recursive rings over the basic FLINT types. https://github.com/fredrik-johansson/bland

See the FLINT website http://flintlib.org/ for a full list.

Writing extension modules is trivial. One should include a top level directory containing a single directory for each module one wishes to supply. FLINT expects to find, in the top level, a `.h` file with the same name as each module directory.

Inside each module directory there must be at least one `.c` file. There should also be a `test` subdirectory. Test files in the `test` subdirectories should be of the form `t-*.c` or `t-*.cpp`.

You may optionally add `doc`, `profile` and `tune` subdirectories to each module directory. These may be supported by some later version of FLINT.

§9. Test code

Each module of FLINT has an extensive associated test module. We strongly recommend running the test programs before relying on results from FLINT on your system.

To make and run the test programs, simply type:

```
make check
```

in the main FLINT directory after configuring FLINT.

§10. Reporting bugs

The maintainer wishes to be made aware of any and all bugs. Please send an email with your bug report to hart_wb@yahoo.com or report them on the FLINT devel list https://groups.google.com/group/flint-devel?hl=en.

If possible please include details of your system, the version of GCC, the versions of GMP/MPIR and MPFR as well as precise details of how to replicate the bug.

Note that FLINT needs to be linked against version 2.6.0 or later of MPIR (or version 5.1.1 or later of GMP), version 3.0.0 or later of MPFR and must be compiled with gcc version 2.96 or later.

§11. Contributors

FLINT has been developed since 2007 by a large number of people. Initially the library was started by David Harvey and William Hart. Later maintenance of the library was taken over solely by William Hart.

The authors of FLINT to date:

• William Hart – integer and polynomial arithmetic, factorisation and primality testing, general infrastructure (supported by EPSRC Grant EP/G004870/1 and DFG Priority programme SPP1489)

• Sebastian Pancratz – polynomial arithmetic over $\mathbf{Z}$, $\mathbf{Z}/n\mathbf{Z}$ and $\mathbf{Q}$, p-adic and q-adic arithmetic, including polynomials and matrices (supported by ERC Grant 204083)

• Andy Novocin – LLL, polynomial factorisation over Z, polynomial composition

• Fredrik Johansson – matrices, polynomial and power series arithmetic, special functions (supported by Austrian Science Fund FWF Grant Y464-N18)

• Tom Bachmann – `C++` expressions template wrapper, documentation parser (Google Summer of Code 2013)

• Mike Hansen – Finite fields (small and large $\mathbf{F}_q$), polynomials/matrices over $\mathbf{F}_q$, Finite fields with Zech logarithm representation, Fast factorisation of polynomials over $\mathbf{F}_q$ (supported by Macaulay2 developers NSF Grant 1002171)

• Martin Lee – Fast factorisation of polynomials over $\mathbf{Z}/n\mathbf{Z}$, faster Brent-Kung modular composition

• David Harvey – Fast Fourier Transform code, `zn_poly` for polynomial arithmetic over $\mathbf{Z}/n\mathbf{Z}$, `mpz_poly`, profiling and graphing code and many other parts of the FLINT library

• Jan Tuitman – helped with the p-adic interface

• Jason Papadopoulos – Block Lanczos code for quadratic sieve and multiprecision complex root finding code for polynomials.

• Gonzalo Tornaria – Theta function module, Montgomery multiplication and significant contributions to the $\mathbf{Z}[x]$ modular multiplication code.

• Burcin Erocal – wrote the primary FLINT wrapper in the SAGE system (Robert Bradshaw also wrote a preliminary version of this and Martin Albrecht and others have also contributed to it.) Burcin also contributed by writing grant applications via his Lmonade organisation to Google. (Supported by DFG Priority programme SPP1489.)

• Tom Boothby – Improved factoring of unsigned longs, detection of perfect powers

• Andres Goens – $\mathbf{F}_q$ module and polynomials over $\mathbf{F}_q$ (supported by DFG Priority program SPP1489)

• Lina Kulakova – factorisation for polynomials over $\mathbf{F}_p$ for large p (Google Summer of Code 2012)

• Thomas DuBuisson – logical ops for fmpz module, patches to the build system

• Jean-Pierre Flori – many build system patches and Sage integration

• Frithjof Schulze – some fmpz functions and various patches

• Curtis Bright – numerous patches including 32 bit support

• Daniel Woodhouse – Contributed an implementation of multivariate multiplication over $\mathbf{Z}/n\mathbf{Z}$ and used this to implement a fast "saturation" algorithm for Laurent polynomials. (Funded by Alessio Corti and Tom Coates at Imperial College)

• Tomasz Lechowski – Contributed some NTL and Pari polynomial profiling code and researched algorithms for polynomials over finite fields. (Funded by the Nuffield Foundation)

• Daniel Scott – Researched lazy and relaxed algorithms of Joris van der Hoeven. (Funded by Warwick University's Undergraduate Research Scholars Scheme)

• David Howden – Wrote code for computing Bernoulli numbers mod many primes, including fast polynomial multiplication over $\mathbf{Z}/p\mathbf{Z}$ specifically for the task. (Funded by Warwick University's Undergraduate Research Scholars Scheme)

• Daniel Ellam – Helped design a module for p-adic arithmetic for FLINT. (Funded by Warwick University's Undergraduate Research Scholars Scheme)

• Richard Howell-Peak – Wrote polynomial factorisation and irreducibility testing code for polynomials over $\mathbf{Z}/p\mathbf{Z}$. (Funded by Warwick University's Undergraduate Research Scholars Scheme)

• Peter Shrimpton – Wrote code for a basic prime sieve, Pocklington-Lehmer, Lucas, Fibonacci, BSPW and $n-1$ primality tests and a Weiferich prime search. (Funded by the Nuffield Foundation)

• Dana Jacobsen – test BPSW primality code up to 2^64 against Feitma's tables and sped up and corrected `n_is_prime` and `n_is_probabprime`.

• Patches and bug reports have been made by Michael Abshoff, Didier Deshommes, Craig Citro, Timothy Abbot, Carl Witty, Gonzalo Tornaria, Jaap Spies, Kiran Kedlaya, William Stein, Kate Minola, Didier Deshommes, Robert Bradshaw, Serge Torres, Dan Grayson, Martin Lee, Bob Smith, Antony Vennard, Frédéric Chyzak, Julien Puydt, Michael Jacobson Jr., Mike Stillman, Jan Englehardt, Jean-Pierre Flori, Jeroen Demeyer, Shi Bai, Qingwen Guan, Frithjof Schulze, Robert Baillie, Oleksandr Motsak, Hans Schoenemann, Janko Boehm, Ahmed Soliman, Francois Bissey and others.

• In addition Michael Abshoff, William Stein and Robert Bradshaw have contributed to the build system of FLINT.

• Michael Abshoff deserves special recognition for his help in resolving a number of difficult build issues which came to light as FLINT was incorporated into SAGE and for bringing numerous bugs to the attention of the FLINT maintainers. Michael regularly checked FLINT for memory leaks and corruption, which directly led to numerous issues being identified early! He also helped with setting up various pieces of infrastructure for the FLINT project.

• Numerous people have contributed to wrapping FLINT in Sage and debugging, including Mike Hansen, Jean-Pierre Flori, Burcin Erocal, Robert Bradshaw, Martin Albrecht, Sebastian Pancratz, Fredrik Johansson, Jeroen Demeyer and Leif Lionhardy, amongst others.

Some code (notably `longlong.h` and `clz_tab.c`) has been used from the GMP library, whose main author is Torbjorn Granlund.

FLINT 2 was a complete rewrite from scratch which began in about 2010.

§12. Tuning FLINT

FLINT uses a highly optimised Fast Fourier Transform routine for polynomial multiplication and some integer multiplication routines. This can be tuned by first typing `make tune` and then running the program `build/fft/tune/tune_fft`.

The output of the program can be pasted into `fft_tuning64.in` or `fft_tuning32.in` depending on the ABI of the current platform. FLINT must then be configured again and a clean build initiated.

Tuning is only necessary if you suspect that very large polynomial and integer operations (millions of bits) are taking longer than they should.

§13. Example programs

FLINT comes with example programs to demonstrate current and future FLINT features. To build the example programs, type:

```
make examples
```

The example programs are built in the `build/examples` directory. You must set your `LD_LIBRARY_PATH` or equivalent for the flint, mpir and mpfr libraries. See your operating system documentation to see how to set this.

The current example programs are:

`partitions` Demonstrates the partition counting code, e.g. `build/examples/partitions 1000000000` will compute the number of partitions of `10^9`.

`delta_qexp` Computes the n-th term of the delta function, e.g. `build/examples/delta_qexp 1000000` will compute the one million-th term of the q-expansion of delta.

`crt` Demonstrates the integer Chinese Remainder code, e.g. `build/examples/crt 10382788` will build up the given integer from its value mod various primes.

`multi_crt` Demonstrates the fast tree version of the integer Chinese Remainder code, e.g. `build/examples/multi_crt 100493287498239 13` will build up the given integer from its value mod the given number of primes.

`stirling_matrix` Generates Stirling number matrices of the first and second kind and computes their product, which should come out as the identity matrix. The matrices are printed to standard output. For example `build/examples/stirling_matrix 10` does this with 10 by 10 matrices.

`fmpz_poly_factor_zassenhaus` Demonstrates the factorisation of a small polynomial. A larger polynomials is also provided on disk and a small (obvious) change to the example program will read this file instead of using the hard coded polynomial.

`padic` Gives examples of the usage of many functions in the padic module.

`fmpz_poly_q` Gives a very simple example of the `fmpz_poly_q` module.

`fmpz_poly` Gives a very simple example of the `fmpz_poly` module.

`fmpq_poly` Gives a very simple example of the `fmpq_poly` module.

Some of the example programs have associated C++ versions.

§14. FLINT macros

The file `flint.h` contains various useful macros.

The macro constant `FLINT_BITS` is set at compile time to be the number of bits per limb on the machine. FLINT requires it to be either 32 or 64 bits. Other architectures are not currently supported.

The macro constant `FLINT_D_BITS` is set at compile time to be the number of bits per double on the machine or one less than the number of bits per limb, whichever is smaller. This will have the value 53 or 31 on currently supported architectures. Numerous internal functions using precomputed inverses only support operands up to `FLINT_D_BITS` bits, hence the macro.

The macro `FLINT_ABS(x)` returns the absolute value of `x` for primitive signed numerical types. It might fail for least negative values such as `INT_MIN` and `WORD_MIN`.

The macro `FLINT_MIN(x, y)` returns the minimum of `x` and `y` for primitive signed or unsigned numerical types. This macro is only safe to use when `x` and `y` are of the same type, to avoid problems with integer promotion.

Similar to the previous macro, `FLINT_MAX(x, y)` returns the maximum of `x` and `y`.

The function `FLINT_BIT_COUNT(x)` returns the number of binary bits required to represent an `ulong x`. If `x` is zero, returns 0.

Derived from this there are the two macros `FLINT_FLOG2(x)` and `FLINT_CLOG2(x)` which, for any $x \geq 1$, compute $\lfloor \log_2 x \rfloor$ and $\lceil \log_2 x \rceil$.

To determine the current FLINT version a number of macros are available. For example, if the current FLINT version is `2.4.0` then `__FLINT_VERSION` will have the value 2, `__FLINT_MINOR` will have the value 4 and `__FLINT_PATCHLEVEL` will have the value 0. The `__FLINT_RELEASE` macro will have the value 20400.

The `FLINT_VERSION` macro is a static text string giving the version number, e.g. “2.4” or “2.4.1”. Note that if the final digit is a zero it is suppressed.

§15. Memory management

The file `flint.h` defines functions `flint_malloc`, `flint_realloc`, `flint_calloc` and `flint_free`. They have the same interface as the standard library functions, but may perform additional error checking.

FLINT may cache some data (such as allocated integers and tables of prime numbers) to speed up various computations. If FLINT is built in threadsafe mode, cached data is kept in thread-local storage by default (unless configured otherwise). Cached data can be freed by calling the `flint_cleanup()` function. It is recommended to call `flint_cleanup()` right before exiting a thread, and at the end of the main program.

The user can register additional cleanup functions to be invoked by `flint_cleanup()` by passing a pointer to a function with signature `void cleanup_function(void)` to `flint_register_cleanup_function()`.

§16. Temporary allocation

FLINT allows for temporary allocation of memory using `alloca` to allocate on the stack if the allocation is small enough.

The following program demonstrates how to use this facility to allocate two different arrays.

```
#include <gmp.h>
#include "flint.h"

void myfun(void)
{
   /* other variable declarations */
   mp_ptr a, b;
   TMP_INIT;

   /* arbitrary code*/

   TMP_START; /* we are about to do some allocation */

   /* arbitrary code */

   a = TMP_ALLOC(32*sizeof(mp_limb_t));
   b = TMP_ALLOC(64*sizeof(mp_limb_t));

   /* arbitrary code */

   TMP_END; /* cleans up a and b */

   /* arbitrary code */
}
```

It is very important to note that temporary allocations should not be made in recursive functions, as many small allocations on the stack can exhaust the stack causing a stack overflow.

§17. Platform-safe types, format specifiers and constants

For platform independence, FLINT provides two types `ulong` and `slong` to replace `unsigned long` and `long` respectively. These are guaranteed to be the same size as GMP's `mp_limb_t` and `mp_limb_signed_t` types, respectively.

A full list of types provided by FLINT is available in `code_conventions.txt` in the top-level source tree.

As FLINT supports Windows 64 on which the FLINT `ulong` and `slong` types are 64 bits, whilst `unsigned long` and `long` are only 32 bits, it is necessary to have a special format specifier which is 64 bits on Windows 64 instead of the usual `"%lu"` and `"%ld"`.

For this purpose FLINT provides its own I/O functions, `flint_printf`, `flint_fprintf`, `flint_sprintf`, `flint_scanf`, `flint_fscanf` and `flint_sscanf`, which work exactly as the usual system versions, but which take the `"%wu"` and `"%wd"` format specifiers, which support FLINT `ulong` and `slong` types respectively.

Also, instead of using constants `123UL` and `123L`, FLINT provides the macros `UWORD(123)` and `WORD(123)` respectively for constants of type `ulong` and `slong` respectively.

The maximum and minimum values that can be represented by these types are given by `UWORD_MAX` and `WORD_MAX` respectively.

§18. fmpz

Arbitrary precision integers

18.1 Introduction

By default, an `fmpz_t` is implemented as an array of `fmpz`'s of length one to allow passing by reference as one can do with GMP/ MPIR's `mpz_t` type. The `fmpz_t` type is simply a single limb, though the user does not need to be aware of this except in one specific case outlined below.

In all respects, `fmpz_t`'s act precisely like GMP/ MPIR's `mpz_t`'s, with automatic memory management, however, in the first place only one limb is used to implement them. Once an `fmpz_t` overflows a limb then a multiprecision integer is automatically allocated and instead of storing the actual integer data the `slong` which implements the type becomes an index into a FLINT wide array of `mpz_t`'s.

These internal implementation details are not important for the user to understand, except for three important things.

Firstly, `fmpz_t`'s will be more efficient than `mpz_t`'s for single limb operations, or more precisely for signed quantities whose absolute value does not exceed `FLINT_BITS - 2` bits.

Secondly, for small integers that fit into `FLINT_BITS - 2` bits much less memory will be used than for an `mpz_t`. When very many `fmpz_t`'s are used, there can be important cache benefits on account of this.

Thirdly, it is important to understand how to deal with arrays of `fmpz_t`'s. As for `mpz_t`'s, there is an underlying type, an `fmpz`, which can be used to create the array, e.g.

```
fmpz myarr[100];
```

Now recall that an `fmpz_t` is an array of length one of `fmpz`'s. Thus, a pointer to an `fmpz` can be used in place of an `fmpz_t`. For example, to find the sign of the third integer in our array we would write

```
int sign = fmpz_sgn(myarr + 2);
```

The `fmpz` module provides routines for memory management, basic manipulation and basic arithmetic.

Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

18.2 Simple example

The following example computes the square of the integer 7 and prints the result.

```
#include "fmpz.h"
...
fmpz_t x, y;
fmpz_init(x);
fmpz_init(y);
fmpz_set_ui(x, 7);
fmpz_mul(y, x, x);
fmpz_print(x);
flint_printf("^2 = ");
fmpz_print(y);
flint_printf("\n");
fmpz_clear(x);
fmpz_clear(y);
```

The output is:

```
7^2 = 49
```

We now describe the functions available in the `fmpz` module.

18.3 Memory management

`void fmpz_init(fmpz_t f)`

A small `fmpz_t` is initialised, i.e. just a `slong`. The value is set to zero.

`void fmpz_init2(fmpz_t f, ulong limbs)`

Initialises the given `fmpz_t` to have space for the given number of limbs.

If `limbs` is zero then a small `fmpz_t` is allocated, i.e. just a `slong`. The value is also set to zero. It is not necessary to call this function except to save time. A call to `fmpz_init` will do just fine.

`void fmpz_clear(fmpz_t f)`

Clears the given `fmpz_t`, releasing any memory associated with it, either back to the stack or the OS, depending on whether the reentrant or non-reentrant version of FLINT is built.

`void fmpz_init_set(fmpz_t f, const fmpz_t g)`

Initialises f and sets it to the value of g.

`void fmpz_init_set_ui(fmpz_t f, ulong g)`

Initialises f and sets it to the value of g.

18.4 Random generation

For thread-safety, the randomisation methods take as one of their parameters an object of type `flint_rand_t`. Before calling any of the randomisation functions such an object first has to be initialised with a call to `flint_randinit()`. When one is finished generating random numbers, one should call `flint_randclear()` to clean up.

```
void fmpz_randbits(fmpz_t f, flint_rand_t state,
   mp_bitcnt_t bits)
```

Generates a random signed integer whose absolute value has the given number of bits.

```
void fmpz_randtest(fmpz_t f, flint_rand_t state,
   mp_bitcnt_t bits)
```

Generates a random signed integer whose absolute value has a number of bits which is random from 0 up to `bits` inclusive.

```
void fmpz_randtest_unsigned(fmpz_t f, flint_rand_t state,
   mp_bitcnt_t bits)
```

Generates a random unsigned integer whose value has a number of bits which is random from 0 up to `bits` inclusive.

```
void fmpz_randtest_not_zero(fmpz_t f, flint_rand_t state,
   mp_bitcnt_t bits)
```

As per `fmpz_randtest`, but the result will not be 0. If `bits` is set to 0, an exception will result.

```
void fmpz_randm(fmpz_t f, flint_rand_t state, const fmpz_t
   m)
```

Generates a random integer in the range 0 to $m - 1$ inclusive.

```
void fmpz_randtest_mod(fmpz_t f, flint_rand_t state, const
   fmpz_t m)
```

Generates a random integer in the range 0 to $m - 1$ inclusive, with an increased probability of generating values close to the endpoints.

```
void fmpz_randtest_mod_signed(fmpz_t f, flint_rand_t state,
   const fmpz_t m)
```

Generates a random integer in the range $(-m/2, m/2]$, with an increased probability of generating values close to the endpoints or close to zero.

18.5 Conversion

```
slong fmpz_get_si(const fmpz_t f)
```

Returns f as a `slong`. The result is undefined if f does not fit into a `slong`.

```
ulong fmpz_get_ui(const fmpz_t f)
```

Returns f as an `ulong`. The result is undefined if f does not fit into an `ulong` or is negative.

```
void fmpz_set_d(fmpz_t f, double c)
```

Sets f to the `double` c, rounding down towards zero if the value of c is fractional. The outcome is undefined if c is infinite, not-a-number, or subnormal.

```
double fmpz_get_d(const fmpz_t f)
```

Returns f as a `double`, rounding down towards zero if `f` cannot be represented exactly. The outcome is undefined if `f` is too large to fit in the normal range of a double.

`double fmpz_get_d_2exp(slong * exp, const fmpz_t f)`

Returns f as a normalized `double` along with a 2-exponent `exp`, i.e. if r is the return value then `f = r * 2^exp`, to within 1 ULP.

`void fmpz_get_mpz(mpz_t x, const fmpz_t f)`

Sets the `mpz_t` x to the same value as f.

`char * fmpz_get_str(char * str, int b, const fmpz_t f)`

Returns the representation of f in base b, which can vary between 2 and 62, inclusive.

If `str` is `NULL`, the result string is allocated by the function. Otherwise, it is up to the caller to ensure that the allocated block of memory is sufficiently large.

`void fmpz_set_si(fmpz_t f, slong val)`

Sets f to the given `slong` value.

`void fmpz_set_ui(fmpz_t f, ulong val)`

Sets f to the given `ulong` value.

`void fmpz_neg_ui(fmpz_t f, ulong val)`

Sets f to the given `ulong` value, and then negates f.

`void fmpz_set_uiui(fmpz_t f, mp_limb_t hi, mp_limb_t lo)`

Sets f to `lo`, plus `hi` shifted to the left by `FLINT_BITS`.

`void fmpz_neg_uiui(fmpz_t f, mp_limb_t hi, mp_limb_t lo)`

Sets f to `lo`, plus `hi` shifted to the left by `FLINT_BITS`, and then negates f.

`void fmpz_set_mpz(fmpz_t f, const mpz_t x)`

Sets f to the given `mpz_t` value.

`int fmpz_set_str(fmpz_t f, const char * str, int b)`

Sets f to the value given in the null-terminated string `str`, in base b. The base b can vary between 2 and 62, inclusive. Returns 0 if the string contains a valid input and -1 otherwise.

`void fmpz_set_ui_smod(fmpz_t f, mp_limb_t x, mp_limb_t m)`

Sets f to the signed remainder $y \equiv x \bmod m$ satisfying $-m/2 < y \leq m/2$, given x which is assumed to satisfy $0 \leq x < m$.

`void flint_mpz_init_set_readonly(mpz_t z, const fmpz_t f)`

Sets the unitialised `mpz_t` z to the value of the readonly `fmpz_t` f.

Note that it is assumed that f does not change during the lifetime of z.

The integer z has to be cleared by a call to `flint_mpz_clear_readonly()`.

The suggested use of the two functions is as follows:

```
fmpz_t f;
...
{
    mpz_t z;
```

```
    flint_mpz_init_set_readonly(z, f);
    foo(..., z);
    flint_mpz_clear_readonly(z);
}
```

This provides a convenient function for user code, only requiring to work with the types `fmpz_t` and `mpz_t`.

In critical code, the following approach may be favourable:

```
fmpz_t f;
...
{
    __mpz_struct *z;

    z = _fmpz_promote_val(f);
    foo(..., z);
    _fmpz_demote_val(f);
}
```

`void flint_mpz_clear_readonly(mpz_t z)`

Clears the readonly `mpz_t` z.

`void fmpz_init_set_readonly(fmpz_t f, const mpz_t z)`

Sets the uninitialised `fmpz_t` f to a readonly version of the integer z.

Note that the value of z is assumed to remain constant throughout the lifetime of f.

The `fmpz_t` f has to be cleared by calling the function `fmpz_clear_readonly()`.

The suggested use of the two functions is as follows:

```
mpz_t z;
...
{
    fmpz_t f;

    fmpz_init_set_readonly(f, z);
    foo(..., f);
    fmpz_clear_readonly(f);
}
```

`void fmpz_clear_readonly(fmpz_t f)`

Clears the readonly `fmpz_t` f.

18.6 Input and output

`int fmpz_read(fmpz_t f)`

Reads a multiprecision integer from `stdin`. The format is an optional minus sign, followed by one or more digits. The first digit should be non-zero unless it is the only digit.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

This convention is adopted in light of the return values of `scanf` from the standard library and `mpz_inp_str` from MPIR.

`int fmpz_fread(FILE * file, fmpz_t f)`

Reads a multiprecision integer from the stream `file`. The format is an optional minus sign, followed by one or more digits. The first digit should be non-zero unless it is the only digit.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

This convention is adopted in light of the return values of `scanf` from the standard library and `mpz_inp_str` from MPIR.

`size_t fmpz_inp_raw( fmpz_t x, FILE *fin )`

Reads a multiprecision integer from the stream `file`. The format is raw binary format write by `fmpz_out_raw`.

In case of success, return a posivitive number, indicating number of bytes read. In case of failure 0.

This function calls the `mpz_inp_raw` function in library gmp. So that it can read the raw data writen by `mpz_inp_raw` directly.

`int fmpz_print(fmpz_t x)`

Prints the value x to `stdout`, without a carriage return(CR). The value is printed as either 0, the decimal digits of a positive integer, or a minus sign followed by the digits of a negative integer.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

This convention is adopted in light of the return values of `flint_printf` from the standard library and `mpz_out_str` from MPIR.

`int fmpz_fprint(FILE * file, fmpz_t x)`

Prints the value x to `file`, without a carriage return(CR). The value is printed as either 0, the decimal digits of a positive integer, or a minus sign followed by the digits of a negative integer.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

This convention is adopted in light of the return values of `flint_printf` from the standard library and `mpz_out_str` from MPIR.

`size_t fmpz_out_raw( FILE *fout, const fmpz_t x )`

Writes the value x to `file`. The value is writen in raw binary format. The integer is written in portable format, with 4 bytes of size information, and that many bytes of linbs. Both the size and the limbs are written in decreasing significance order (i.e., in big-endian).

The output can be read with `fmpz_inp_raw`.

In case of success, return a positive number, indicating number of bytes written. In case of failure, return 0.

The output of this can also be read by `mpz_inp_raw` from GMP $>= 2$, Since this function calls the `mpz_inp_raw` function in library gmp.

18.7 Basic properties and manipulation

`size_t fmpz_sizeinbase(const fmpz_t f, int b)`

Returns the size of the absolute value of f in base b, measured in numbers of digits. The base b can be between 2 and 62, inclusive.

```
mp_bitcnt_t fmpz_bits(const fmpz_t f)
```

Returns the number of bits required to store the absolute value of f. If f is 0 then 0 is returned.

```
mp_size_t fmpz_size(const fmpz_t f)
```

Returns the number of limbs required to store the absolute value of f. If f is zero then 0 is returned.

```
int fmpz_sgn(const fmpz_t f)
```

Returns -1 if the sign of f is negative, $+1$ if it is positive, otherwise returns 0.

```
mp_bitcnt_t fmpz_val2(const fmpz_t f)
```

Returns the exponent of the largest power of two dividing f, or equivalently the number of trailing zeros in the binary expansion of f. If f is zero then 0 is returned.

```
void fmpz_swap(fmpz_t f, fmpz_t g)
```

Efficiently swaps f and g. No data is copied.

```
void fmpz_set(fmpz_t f, const fmpz_t g)
```

Sets f to the same value as g.

```
void fmpz_zero(fmpz_t f)
```

Sets f to zero.

```
void fmpz_one(fmpz_t f)
```

Sets f to one.

```
int fmpz_abs_fits_ui(const fmpz_t f)
```

Returns whether the absolute value of f fits into an `ulong`.

```
int fmpz_fits_si(const fmpz_t f)
```

Returns whether the value of f fits into a `slong`.

```
void fmpz_setbit(fmpz_t f, ulong i)
```

Sets bit index i of f.

```
int fmpz_tstbit(const fmpz_t f, ulong i)
```

Test bit index i of f and return 0 or 1, accordingly.

```
mp_limb_t fmpz_abs_lbound_ui_2exp(slong * exp, const fmpz_t
   x, int bits)
```

For nonzero x, returns a mantissa m with exactly `bits` bits and sets `exp` to an exponent e, such that $|x| \geq m2^e$. The number of bits must be between 1 and `FLINT_BITS` inclusive. The mantissa is guaranteed to be correctly rounded.

```
mp_limb_t fmpz_abs_ubound_ui_2exp(slong * exp, const fmpz_t
   x, int bits)
```

For nonzero x, returns a mantissa m with exactly `bits` bits and sets `exp` to an exponent e, such that $|x| \leq m2^e$. The number of bits must be between 1 and `FLINT_BITS` inclusive. The mantissa is either correctly rounded or one unit too large (possibly meaning that the exponent is one too large, if the mantissa is a power of two).

18.8 Comparison

`int fmpz_cmp(const fmpz_t f, const fmpz_t g)`

Returns a negative value if $f < g$, positive value if $g < f$, otherwise returns 0.

`int fmpz_cmp_ui(const fmpz_t f, ulong g)`

Returns a negative value if $f < g$, positive value if $g < f$, otherwise returns 0.

`int fmpz_cmp_si(const fmpz_t f, slong g)`

Returns a negative value if $f < g$, positive value if $g < f$, otherwise returns 0.

`int fmpz_cmpabs(const fmpz_t f, const fmpz_t g)`

Returns a negative value if $|f| < |g|$, positive value if $|g| < |f|$, otherwise returns 0.

`int fmpz_equal(const fmpz_t f, const fmpz_t g)`

Returns 1 if f is equal to g, otherwise returns 0.

`int fmpz_equal_ui(const fmpz_t f, ulong g)`

Returns 1 if f is equal to g, otherwise returns 0.

`int fmpz_equal_si(const fmpz_t f, slong g)`

Returns 1 if f is equal to g, otherwise returns 0.

`int fmpz_is_zero(const fmpz_t f)`

Returns 1 if f is 0, otherwise returns 0.

`int fmpz_is_one(const fmpz_t f)`

Returns 1 if f is equal to one, otherwise returns 0.

`int fmpz_is_pm1(const fmpz_t f)`

Returns 1 if f is equal to one or minus one, otherwise returns 0.

`int fmpz_is_even(const fmpz_t f)`

Returns whether the integer f is even.

`int fmpz_is_odd(const fmpz_t f)`

Returns whether the integer f is odd.

18.9 Basic arithmetic

`void fmpz_neg(fmpz_t f1, const fmpz_t f2)`

Sets f_1 to $-f_2$.

`void fmpz_abs(fmpz_t f1, const fmpz_t f2)`

Sets f_1 to the absolute value of f_2.

`void fmpz_add(fmpz_t f, const fmpz_t g, const fmpz_t h)`

Sets f to $g + h$.

`void fmpz_add_ui(fmpz_t f, const fmpz_t g, ulong x)`

Sets f to $g + x$ where x is an `ulong`.

`void fmpz_sub(fmpz_t f, const fmpz_t g, const fmpz_t h)`

Sets f to $g - h$.

`void fmpz_sub_ui(fmpz_t f, const fmpz_t g, ulong x)`

Sets f to $g - x$ where x is an `ulong`.

`void fmpz_mul(fmpz_t f, const fmpz_t g, const fmpz_t h)`

Sets f to $g \times h$.

`void fmpz_mul_si(fmpz_t f, const fmpz_t g, slong x)`

Sets f to $g \times x$ where x is a `slong`.

`void fmpz_mul_ui(fmpz_t f, const fmpz_t g, ulong x)`

Sets f to $g \times x$ where x is an `ulong`.

```
void fmpz_mul2_uiui(fmpz_t f, const fmpz_t g, ulong x,
   ulong y)
```

Sets f to $g \times x \times y$ where x and y are of type `ulong`.

`void fmpz_mul_2exp(fmpz_t f, const fmpz_t g, ulong e)`

Sets f to $g \times 2^e$.

`void fmpz_addmul(fmpz_t f, const fmpz_t g, const fmpz_t h)`

Sets f to $f + g \times h$.

`void fmpz_addmul_ui(fmpz_t f, const fmpz_t g, ulong x)`

Sets f to $f + g \times x$ where x is an `ulong`.

`void fmpz_submul(fmpz_t f, const fmpz_t g, const fmpz_t h)`

Sets f to $f - g \times h$.

`void fmpz_submul_ui(fmpz_t f, const fmpz_t g, ulong x)`

Sets f to $f - g \times x$ where x is an `ulong`.

`void fmpz_cdiv_q(fmpz_t f, const fmpz_t g, const fmpz_t h)`

Sets f to the quotient of g by h, rounding up towards infinity. If h is 0 an exception is raised.

`void fmpz_cdiv_q_si(fmpz_t f, const fmpz_t g, slong h)`

Sets f to the quotient of g by h, rounding up towards infinity. If h is 0 an exception is raised.

`void fmpz_cdiv_q_ui(fmpz_t f, const fmpz_t g, ulong h)`

Sets f to the quotient of g by h, rounding up towards infinity. If h is 0 an exception is raised.

`void fmpz_fdiv_q_2exp(fmpz_t f, const fmpz_t g, ulong exp)`

Sets f to g divided by `2^exp`, rounding down towards minus infinity.

`void fmpz_fdiv_q(fmpz_t f, const fmpz_t g, const fmpz_t h)`

Sets f to the quotient of g by h, rounding down towards minus infinity. If h is 0 an exception is raised.

`void fmpz_fdiv_q_si(fmpz_t f, const fmpz_t g, slong h)`

Set f to the quotient of g by h, rounding down towards minus infinity. If h is 0 an exception is raised.

`void fmpz_fdiv_q_ui(fmpz_t f, const fmpz_t g, ulong h)`

Set f to the quotient of g by h, rounding down towards minus infinity. If h is 0 an exception is raised.

`void fmpz_fdiv_qr(fmpz_t f, fmpz_t s, const fmpz_t g, const fmpz_t h)`

Sets f to the quotient of g by h, rounding down towards minus infinity and s to the remainder. If h is 0 an exception is raised.

`void fmpz_fdiv_r(fmpz_t f, const fmpz_t g, const fmpz_t h)`

Sets f to the remainder from dividing g by h and rounding the quotient down towards minus infinity. If h is 0 an exception is raised.

`void fmpz_fdiv_q_2exp(fmpz_t f, const fmpz_t g, ulong exp)`

Sets f to g divided by `2^exp`, rounding down towards minus infinity.

`void fmpz_fdiv_r_2exp(fmpz_t f, const fmpz_t g, ulong exp)`

Sets f to the remainder of g upon division by `2^exp`, where the remainder is non-negative.

`void fmpz_tdiv_q(fmpz_t f, const fmpz_t g, const fmpz_t h)`

Sets f to the quotient of g by h, rounding down towards zero. If h is 0 an exception is raised.

`void fmpz_tdiv_qr(fmpz_t f, fmpz_t s, const fmpz_t g, const fmpz_t h)`

Sets f to the quotient of g by h, rounding down towards zero and s to the remainder. If h is 0 an exception is raised.

`void fmpz_tdiv_q_si(fmpz_t f, const fmpz_t g, slong h)`

Set f to the quotient of g by h, rounding down towards zero. If h is 0 an exception is raised.

`void fmpz_tdiv_q_ui(fmpz_t f, const fmpz_t g, ulong h)`

Set f to the quotient of g by h, rounding down towards zero. If h is 0 an exception is raised.

`ulong fmpz_tdiv_ui(const fmpz_t g, ulong h)`

Returns the absolute value of the remainder from dividing g by h, rounding towards zero. If h is 0 an exception is raised.

`void fmpz_tdiv_q_2exp(fmpz_t f, const fmpz_t g, ulong exp)`

Sets f to g divided by `2^exp`, rounding down towards zero.

`void fmpz_divexact(fmpz_t f, const fmpz_t g, const fmpz_t h)`

Sets f to the quotient of g and h, assuming that the division is exact, i.e. g is a multiple of h. If h is 0 an exception is raised.

`void fmpz_divexact_si(fmpz_t f, const fmpz_t g, slong h)`

Sets f to the quotient of g and h, assuming that the division is exact, i.e. g is a multiple of h. If h is 0 an exception is raised.

`void fmpz_divexact_ui(fmpz_t f, const fmpz_t g, ulong h)`

Sets f to the quotient of g and h, assuming that the division is exact, i.e. g is a multiple of h. If h is 0 an exception is raised.

```
void fmpz_divexact2_uiui(fmpz_t f, const fmpz_t g, ulong x,
   ulong y)
```

Sets f to the quotient of g and $h = x \times y$, assuming that the division is exact, i.e. g is a multiple of h. If x or y is 0 an exception is raised.

`int fmpz_divisible(const fmpz_t f, const fmpz_t g)`

Returns whether f is divisible by $g > 0$.

`int fmpz_divisible_si(const fmpz_t f, slong g)`

Returns whether f is divisible by $g > 0$.

`void fmpz_mod(fmpz_t f, const fmpz_t g, const fmpz_t h)`

Sets f to the remainder of g divided by h. The remainder is always taken to be positive.

`ulong fmpz_mod_ui(fmpz_t f, const fmpz_t g, ulong x)`

Sets f to g reduced modulo x where x is an `ulong`. If x is 0 an exception will result.

`ulong fmpz_fdiv_ui(const fmpz_t g, ulong x)`

Returns the remainder of g modulo x where x is an `ulong`, without changing g. If x is 0 an exception will result.

`void fmpz_preinvn_init(fmpz_preinvn_t inv, fmpz_t f)`

Compute a precomputed inverse `inv` of `f` for use in the `preinvn` functions listed below.

`void fmpz_preinvn_clear(fmpz_preinvn_t inv)`

Clean up the resources used by a precomputed inverse created with the `fmpz_preinvn_init` function.

```
void fmpz_fdiv_qr_preinvn(fmpz_t f, fmpz_t s, const fmpz_t
   g, const fmpz_t h, const fmpz_preinvn_t hinv)
```

As per `fmpz_fdiv_qr`, but takes a precomputed inverse `hinv` of h constructed using `fmpz_preinvn`.

This function will be faster than `fmpz_fdiv_qr_preinvn` when the number of limbs of h is at least `PREINVN_CUTOFF`.

```
void fmpz_pow_ui(fmpz_t f, const fmpz_t g, ulong x)
```

Sets f to g^x where x is an `ulong`. If x is 0 and g is 0, then f will be set to 1.

```
void fmpz_powm_ui(fmpz_t f, const fmpz_t g, ulong e, const
    fmpz_t m)
```

Sets f to $g^e \bmod m$. If $e = 0$, sets f to 1.

Assumes that $m \neq 0$, raises an `abort` signal otherwise.

```
void fmpz_powm(fmpz_t f, const fmpz_t g, const fmpz_t e,
    const fmpz_t m)
```

Sets f to $g^e \bmod m$. If $e = 0$, sets f to 1.

Assumes that $m \neq 0$, raises an `abort` signal otherwise.

```
slong fmpz_clog(const fmpz_t x, const fmpz_t b)

slong fmpz_clog_ui(const fmpz_t x, ulong b)
```

Returns $\lceil \log_b x \rceil$.

Assumes that $x \geq 1$ and $b \geq 2$ and that the return value fits into a signed `slong`.

```
slong fmpz_flog(const fmpz_t x, const fmpz_t b)

slong fmpz_flog_ui(const fmpz_t x, ulong b)
```

Returns $\lfloor \log_b x \rfloor$.

Assumes that $x \geq 1$ and $b \geq 2$ and that the return value fits into a signed `slong`.

```
double fmpz_dlog(const fmpz_t x)
```

Returns a double precision approximation of the natural logarithm of x.

The accuracy depends on the implementation of the floating-point logarithm provided by the C standard library. The result can typically be expected to have a relative error no greater than 1-2 bits.

```
int fmpz_sqrtmod(fmpz_t b, const fmpz_t a, const fmpz_t p)
```

Returns whether a is a quadratic residue or zero modulo p and sets b to a square root of a if this is the case.

```
void fmpz_sqrt(fmpz_t f, const fmpz_t g)
```

Sets f to the integer part of the square root of g, where g is assumed to be non-negative. If g is negative, an exception is raised.

```
void fmpz_sqrtrem(fmpz_t f, fmpz_t r, const fmpz_t g)
```

Sets f to the integer part of the square root of g, where g is assumed to be non-negative, and sets r to the remainder, that is, the difference $g - f^2$. If g is negative, an exception is raised. The behaviour is undefined if f and r are aliases.

```
int fmpz_is_square(const fmpz_t f)
```

Returns nonzero if f is a perfect square and zero otherwise.

```
void fmpz_root(fmpz_t r, const fmpz_t f, slong n)
```

Set r to the integer part of the n-th root of f. Requires that $n > 0$ and that if n is even then f be non-negative, otherwise an exception is raised.

```
void fmpz_fac_ui(fmpz_t f, ulong n)
```

Sets f to the factorial $n!$ where n is an `ulong`.

```
void fmpz_fib_ui(fmpz_t f, ulong n)
```

Sets f to the Fibonacci number F_n where n is an `ulong`.

```
void fmpz_bin_uiui(fmpz_t f, ulong n, ulong k)
```

Sets f to the binomial coefficient $\binom{n}{k}$.

```
void fmpz_rfac_ui(fmpz_t r, const fmpz_t x, ulong k)
```

Sets r to the rising factorial $x(x+1)(x+2)\cdots(x+k-1)$.

```
void fmpz_rfac_uiui(fmpz_t r, ulong x, ulong k)
```

Sets r to the rising factorial $x(x+1)(x+2)\cdots(x+k-1)$.

```
void fmpz_mul_tdiv_q_2exp(fmpz_t f, const fmpz_t g, const
   fmpz_t h, ulong exp)
```

Sets f to the product g and h divided by `2^exp`, rounding down towards zero.

```
void fmpz_mul_si_tdiv_q_2exp(fmpz_t f, const fmpz_t g,
   slong x, ulong exp)
```

Sets f to the product g and x divided by `2^exp`, rounding down towards zero.

18.10 Greatest common divisor

```
void fmpz_gcd(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to the greatest common divisor of g and h. The result is always positive, even if one of g and h is negative.

```
void fmpz_lcm(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to the least common multiple of g and h. The result is always nonnegative, even if one of g and h is negative.

```
void fmpz_gcdinv(fmpz_t d, fmpz_t a, const fmpz_t f, const
   fmpz_t g)
```

Given integers f, g with $0 \leq f < g$, computes the greatest common divisor $d = \gcd(f, g)$ and the modular inverse $a = f^{-1} \pmod{g}$, whenever $f \neq 0$.

Assumes that d and a are not aliased.

```
void fmpz_xgcd(fmpz_t d, fmpz_t a, fmpz_t b, const fmpz_t
   f, const fmpz_t g)
```

Computes the extended GCD of f and g, i.e. values a and b such that $af + bg = d$, where $d = \gcd(f, g)$.

Assumes that d is not aliased with a or b and that a and b are not aliased.

```
void fmpz_xgcd_partial(fmpz_t co2, fmpz_t co1, fmpz_t r2,
   fmpz_t r1, const fmpz_t L)
```

This function is an implementation of Lehmer extended GCD with early termination, as used in the `qfb` module. It terminates early when remainders fall below the specified bound. The initial values `r1` and `r2` are treated as successive remainders in the Euclidean algorithm and are replaced with the last two remainders computed. The values `co1` and `co2` are the last two cofactors and satisfy the identity `co2*r1 - co1*r2 == +/- r2_orig` upon termination, where `r2_orig` is the starting value of `r2` supplied, and `r1` and `r2` are the final values.

Aliasing of inputs is not allowed. Similarly aliasing of inputs and outputs is not allowed.

18.11 Modular arithmetic

```
slong _fmpz_remove(fmpz_t x, const fmpz_t f, double finv)
```

Removes all factors f from x and returns the number of such.

Assumes that x is non-zero, that $f > 1$ and that `finv` is the precomputed `double` inverse of f whenever f is a small integer and 0 otherwise.

Does not support aliasing.

```
slong fmpz_remove(fmpz_t rop, const fmpz_t op, const fmpz_t
   f)
```

Remove all occurrences of the factor $f > 1$ from the integer `op` and sets `rop` to the resulting integer.

If `op` is zero, sets `rop` to `op` and returns 0.

Returns an `abort` signal if any of the assumptions are violated.

```
int fmpz_invmod(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to the inverse of g modulo h. The value of h may not be 0 otherwise an exception results. If the inverse exists the return value will be non-zero, otherwise the return value will be 0 and the value of f undefined.

```
void fmpz_negmod(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to $-g \pmod{h}$, assuming g is reduced modulo h.

```
int fmpz_jacobi(const fmpz_t a, const fmpz_t p);
```

Computes the Jacobi symbol of a modulo p, where p is a prime and a is reduced modulo p.

18.12 Bit packing and unpacking

```
int fmpz_bit_pack(mp_limb_t * arr, mp_bitcnt_t shift,
   mp_bitcnt_t bits, fmpz_t coeff, int negate, int borrow)
```

Shifts the given coefficient to the left by `shift` bits and adds it to the integer in `arr` in a field of the given number of bits.

```
shift  bits  --------------

X X X C C C C 0 0 0 0 0 0 0
```

An optional borrow of 1 can be subtracted from `coeff` before it is packed. If `coeff` is negative after the borrow, then a borrow will be returned by the function.

The value of `shift` is assumed to be less than `FLINT_BITS`. All but the first `shift` bits of `arr` are assumed to be zero on entry to the function.

The value of `coeff` may also be optionally (and notionally) negated before it is used, by setting the `negate` parameter to -1.

```
int fmpz_bit_unpack(fmpz_t coeff, mp_limb_t * arr,
   mp_bitcnt_t shift, mp_bitcnt_t bits, int negate, int
   borrow)
```

A bit field of the given number of bits is extracted from `arr`, starting after `shift` bits, and placed into `coeff`. An optional borrow of 1 may be added to the coefficient. If the result is negative, a borrow of 1 is returned. Finally, the resulting `coeff` may be negated by setting the `negate` parameter to -1.

The value of `shift` is expected to be less than `FLINT_BITS`.

```
void fmpz_bit_unpack_unsigned(fmpz_t coeff, const mp_limb_t
   * arr, mp_bitcnt_t shift, mp_bitcnt_t bits)
```

A bit field of the given number of bits is extracted from `arr`, starting after `shift` bits, and placed into `coeff`.

The value of `shift` is expected to be less than `FLINT_BITS`.

18.13 Logic Operations

```
void fmpz_complement(fmpz_t r, const fmpz_t f)
```

The variable `r` is set to the ones-complement of `f`.

```
void fmpz_clrbit(fmpz_t f, ulong i)
```

Sets the `i`th bit in `f` to zero.

```
void fmpz_combit(fmpz_t f, ulong i)
```

Complements the `i`th bit in `f`.

```
void fmpz_and(fmpz_t r, const fmpz_t a, const fmpz_t b)
```

Sets `r` to the bit-wise logical `and` of `a` and `b`.

```
void fmpz_or(fmpz_t r, const fmpz_t a, const fmpz_t b)
```

Sets `r` to the bit-wise logical (inclusive) `or` of `a` and `b`.

```
void fmpz_xor(fmpz_t r, const fmpz_t a, const fmpz_t b)
```

Sets `r` to the bit-wise logical exclusive `or` of `a` and `b`.

```
int fmpz_popcnt(const fmpz_t a)
```

Returns the number of '1' bits in the given Z (aka Hamming weight or population count). The return value is undefined if the input is negative.

18.14 Chinese remaindering

The following functions can be used to reconstruct an integer from its residues modulo a set of small (word-size) prime numbers. The first two functions, `fmpz_CRT_ui` and `fmpz_CRT_ui_unsigned`, are easy to use and allow building the result one residue at a time, which is useful when the number of needed primes is not known in advance.

The remaining functions support performing the modular reductions and reconstruction using balanced subdivision. This greatly improves efficiency for large integers but assumes that the basis of primes is known in advance. The user must precompute a `comb` structure and temporary working space with `fmpz_comb_init` and `fmpz_comb_temp_init`, and free this data afterwards.

For simple demonstration programs showing how to use the CRT functions, see `crt.c` and `multi_crt.c` in the `examples` directory.

```
void fmpz_CRT_ui(fmpz_t out, fmpz_t r1, fmpz_t m1, ulong
   r2, ulong m2, int sign)
```

Uses the Chinese Remainder Theorem to compute the unique integer $0 \leq x < M$ (if sign $= 0$) or $-M/2 < x \leq M/2$ (if sign $= 1$) congruent to r_1 modulo m_1 and r_2 modulo m_2, where where $M = m_1 \times m_2$. The result x is stored in `out`.

It is assumed that m_1 and m_2 are positive integers greater than 1 and coprime.

If sign $= 0$, it is assumed that $0 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$. Otherwise, it is assumed that $-m_1 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$.

```
void fmpz_multi_mod_ui(mp_limb_t * out, const fmpz_t in,
   const fmpz_comb_t comb, fmpz_comb_temp_t temp)
```

Reduces the multiprecision integer `in` modulo each of the primes stored in the `comb` structure. The array `out` will be filled with the residues modulo these primes. The structure `temp` is temporary space which must be provided by `fmpz_comb_temp_init` and cleared by `fmpz_comb_temp_clear`.

```
void fmpz_multi_CRT_ui_unsigned(fmpz_t output, const
   mp_limb_t * residues, const fmpz_comb_t comb,
   fmpz_comb_temp_t temp)
```

This function takes a set of residues modulo the list of primes contained in the `comb` structure and reconstructs the unique unsigned multiprecision integer modulo the product of the primes which has these residues modulo the corresponding primes. The structure `temp` is temporary space which must be provided by `fmpz_comb_temp_init` and cleared by `fmpz_comb_temp_clear`.

```
void fmpz_multi_CRT_ui(fmpz_t output, mp_srcptr residues,
   const fmpz_comb_t comb, fmpz_comb_temp_t temp)
```

This function takes a set of residues modulo the list of primes contained in the `comb` structure and reconstructs a multiprecision integer modulo the product of the primes which has these residues modulo the corresponding primes.

If N is the product of all the primes then `out` is normalised to be in the range $[0, N)$ if sign $= 0$ and the range $[-(N-1)/2, N/2]$ if sign $= 1$. The array `temp` is temporary space which must be provided by `fmpz_comb_temp_init` and cleared by `fmpz_comb_temp_clear`.

```
void fmpz_comb_init(fmpz_comb_t comb, mp_srcptr primes,
    slong num_primes)
```

Initialises a `comb` structure for multimodular reduction and recombination. The array `primes` is assumed to contain `num_primes` primes each of `FLINT_BITS - 1` bits. Modular reductions and recombinations will be done modulo this list of primes. The `primes` array must not be `free`'d until the `comb` structure is no longer required and must be cleared by the user.

```
void fmpz_comb_temp_init(fmpz_comb_temp_t temp, const
    fmpz_comb_t comb)
```

Creates temporary space to be used by multimodular and CRT functions based on an initialised `comb` structure.

```
void fmpz_comb_clear(fmpz_comb_t comb)
```

Clears the given `comb` structure, releasing any memory it uses.

```
void fmpz_comb_temp_clear(fmpz_comb_temp_t temp)
```

Clears temporary space `temp` used by multimodular and CRT functions using the given `comb` structure.

18.15 Primality testing

```
int fmpz_is_probabprime(const fmpz_t p)
```

Performs some trial division and then some probabilistic primality tests. If p is definitely composite, the function returns 0, otherwise it is declared probably prime, i.e. prime for most practical purposes, and the function returns 1. The chance of declaring a composite prime is very small.

Subsequent calls to the same function do not increase the probability of the number being prime.

```
int fmpz_is_prime_pseudosquare(const fmpz_t n)
```

Return 0 is n is composite. If n is too large (greater than about 94 bits) the function fails silently and returns -1, otherwise, if n is proven prime by the pseudosquares method, return 1.

Tests if n is a prime according to [21, Theorem 2.7].

We first factor N using trial division up to some limit B. In fact, the number of primes used in the trial factoring is at most `FLINT_PSEUDOSQUARES_CUTOFF`.

Next we compute N/B and find the next pseudosquare L_p above this value, using a static table as per `http://research.att.com/~njas/sequences/b002189.txt`.

As noted in the text, if p is prime then Step 3 will pass. This test rejects many composites, and so by this time we suspect that p is prime. If N is 3 or 7 modulo 8, we are done, and N is prime.

We now run a probable prime test, for which no known counterexamples are known, to reject any composites. We then proceed to prove N prime by executing Step 4. In the case that N is 1 modulo 8, if Step 4 fails, we extend the number of primes p_i at Step 3 and hope to find one which passes Step 4. We take the test one past the largest p for which we have pseudosquares L_p tabulated, as this already corresponds to the next L_p which is bigger than 2^{64} and hence larger than any prime we might be testing.

As explained in the text, Condition 4 cannot fail if N is prime.

The possibility exists that the probable prime test declares a composite prime. However in that case an error is printed, as that would be of independent interest.

§19. fmpz_vec

Vectors over **Z**

19.1 Memory management

`fmpz * _fmpz_vec_init(slong len)`

Returns an initialised vector of `fmpz`'s of given length.

`void _fmpz_vec_clear(fmpz * vec, slong len)`

Clears the entries of `(vec, len)` and frees the space allocated for `vec`.

19.2 Randomisation

`void _fmpz_vec_randtest(fmpz * f, flint_rand_t state, slong len, mp_bitcnt_t bits)`

Sets the entries of a vector of the given length to random integers with up to the given number of bits per entry.

`void _fmpz_vec_randtest_unsigned(fmpz * f, flint_rand_t state, slong len, mp_bitcnt_t bits)`

Sets the entries of a vector of the given length to random unsigned integers with up to the given number of bits per entry.

19.3 Bit sizes and norms

`slong _fmpz_vec_max_bits(const fmpz * vec, slong len)`

If b is the maximum number of bits of the absolute value of any coefficient of `vec`, then if any coefficient of `vec` is negative, $-b$ is returned, else b is returned.

`slong _fmpz_vec_max_bits_ref(const fmpz * vec, slong len)`

If b is the maximum number of bits of the absolute value of any coefficient of `vec`, then if any coefficient of `vec` is negative, $-b$ is returned, else b is returned. This is a slower reference implementation of `_fmpz_vec_max_bits`.

`ulong _fmpz_vec_max_limbs(const fmpz * vec, slong len)`

Returns the maximum number of limbs needed to store the absolute value of any entry in `(vec, len)`. If all entries are zero, returns zero.

```
void _fmpz_vec_height(fmpz_t height, const fmpz * vec,
    slong len)
```

Computes the height of `(vec, len)`, defined as the largest of the absolute values the coefficients. Equivalently, this gives the infinity norm of the vector. If `len` is zero, the height is 0.

```
slong _fmpz_vec_height_index(const fmpz * vec, slong len)
```

Returns the index of an entry of maximum absolute value in the vector. The the length must be at least 1.

19.4 Input and output

```
int _fmpz_vec_fread(FILE * file, fmpz ** vec, slong * len)
```

Reads a vector from the stream `file` and stores it at `*vec`. The format is the same as the output format of `_fmpz_vec_fprint()`, followed by either any character or the end of the file.

The interpretation of the various input arguments depends on whether or not `*vec` is `NULL`:

If `*vec == NULL`, the value of `*len` on input is ignored. Once the length has been read from `file`, `*len` is set to that value and a vector of this length is allocated at `*vec`. Finally, `*len` coefficients are read from the input stream. In case of a file or parsing error, clears the vector and sets `*vec` and `*len` to `NULL` and `0`, respectively.

Otherwise, if `*vec != NULL`, it is assumed that `(*vec, *len)` is a properly initialised vector. If the length on the input stream does not match `*len`, a parsing error is raised. Attempts to read the right number of coefficients from the input stream. In case of a file or parsing error, leaves the vector `(*vec, *len)` in its current state.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fmpz_vec_read(fmpz ** vec, slong * len)
```

Reads a vector from `stdin` and stores it at `*vec`.

For further details, see `_fmpz_vec_fread()`.

```
int _fmpz_vec_fprint(FILE * file, const fmpz * vec, slong
    len)
```

Prints the vector of given length to the stream `file`. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fmpz_vec_print(const fmpz * vec, slong len)
```

Prints the vector of given length to `stdout`.

For further details, see `_fmpz_vec_fprint()`.

19.5 Conversions

```
void _fmpz_vec_get_nmod_vec(mp_ptr res, const fmpz * poly,
    slong len, nmod_t mod)
```

Reduce the coefficients of `(poly, len)` modulo the given modulus and set `(res, len)` to the result.

```
void _fmpz_vec_set_nmod_vec(fmpz * res, mp_srcptr poly,
    slong len, nmod_t mod)
```

Set the coefficients of `(res, len)` to the symmetric modulus of the coefficients of `(poly, len)`, i.e. convert the given coefficients modulo the given modulus n to their signed integer representatives in the range $[-n/2, n/2)$.

```
slong _fmpz_vec_get_fft(mp_limb_t ** coeffs_f, const fmpz *
    coeffs_m, slong l, slong length)
```

Convert the vector of coeffs `coeffs_m` to an fft vector `coeffs_f` of the given `length` with `l` limbs per coefficient with an additional limb for overflow.

```
void _fmpz_vec_set_fft(fmpz * coeffs_m, slong length, const
    mp_ptr * coeffs_f, slong limbs, slong sign)
```

Convert an fft vector `coeffs_f` of the given `length` to a vector of `fmpz`'s. Each is assumed to be the given number of limbs in length with an additional limb for overflow. If the output coefficients are to be signed then set `sign`, otherwise clear it.

19.6 Assignment and basic manipulation

```
void _fmpz_vec_set(fmpz * vec1, const fmpz * vec2, slong
    len2)
```

Makes a copy of `(vec2, len2)` into `vec1`.

```
void _fmpz_vec_swap(fmpz * vec1, fmpz * vec2, slong len2)
```

Swaps the integers in `(vec1, len2)` and `(vec2, len2)`.

```
void _fmpz_vec_zero(fmpz * vec, slong len)
```

Zeros the entries of `(vec, len)`.

```
void _fmpz_vec_neg(fmpz * vec1, const fmpz * vec2, slong
    len2)
```

Negates `(vec2, len2)` and places it into `vec1`.

19.7 Comparison

```
int _fmpz_vec_equal(const fmpz * vec1, const fmpz * vec2,
    slong len)
```

Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

```
int _fmpz_vec_is_zero(const fmpz * vec, slong len)
```

Returns 1 if `(vec, len)` is zero, and 0 otherwise.

19.8 Sorting

```
void _fmpz_vec_sort(fmpz * vec, slong len)
```

Sorts the coefficients of `vec` in ascending order.

19.9 Addition and subtraction

```
void _fmpz_vec_add(fmpz * res, const fmpz * vec1, const
   fmpz * vec2, slong len2)
```

Sets `(res, len2)` to the sum of `(vec1, len2)` and `(vec2, len2)`.

```
void _fmpz_vec_sub(fmpz * res, const fmpz * vec1, const
   fmpz * vec2, slong len2)
```

Sets `(res, len2)` to `(vec1, len2)` minus `(vec2, len2)`.

19.10 Scalar multiplication and division

```
void _fmpz_vec_scalar_mul_fmpz(fmpz * vec1, const fmpz *
   vec2, slong len2, const fmpz_t x)
```

Sets `(vec1, len2)` to `(vec2, len2)` multiplied by c, where c is an `fmpz_t`.

```
id _fmpz_vec_scalar_mul_si(fmpz * vec1, const fmpz * vec2,
   slong len2, slong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` multiplied by c, where c is a `slong`.

```
void _fmpz_vec_scalar_mul_ui(fmpz * vec1, const fmpz *
   vec2, slong len2, ulong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` multiplied by c, where c is an `ulong`.

```
void _fmpz_vec_scalar_mul_2exp(fmpz * vec1, const fmpz *
   vec2, slong len2, ulong exp)
```

Sets `(vec1, len2)` to `(vec2, len2)` multiplied by `2^exp`.

```
void _fmpz_vec_scalar_divexact_fmpz(fmpz * vec1, const fmpz
   * vec2, slong len2, const fmpz_t x)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by x, where the division is assumed to be exact for every entry in `vec2`.

```
void _fmpz_vec_scalar_divexact_si(fmpz * vec1, const fmpz *
   vec2, slong len2, slong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by x, where the division is assumed to be exact for every entry in `vec2`.

```
void _fmpz_vec_scalar_divexact_ui(fmpz * vec1, const fmpz *
   vec2, ulong len2, ulong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by x, where the division is assumed to be exact for every entry in `vec2`.

```
void _fmpz_vec_scalar_fdiv_q_fmpz(fmpz * vec1, const fmpz *
   vec2, slong len2, const fmpz_t c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by c, rounding down towards minus infinity whenever the division is not exact.

```
void _fmpz_vec_scalar_fdiv_q_si(fmpz * vec1, const fmpz *
   vec2, slong len2, slong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by c, rounding down towards minus infinity whenever the division is not exact.

```
void _fmpz_vec_scalar_fdiv_q_ui(fmpz * vec1, const fmpz *
   vec2, slong len2, ulong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by c, rounding down towards minus infinity whenever the division is not exact.

```
void _fmpz_vec_scalar_fdiv_q_2exp(fmpz * vec1, const fmpz *
   vec2, slong len2, ulong exp)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by `2^exp`, rounding down towards minus infinity whenever the division is not exact.

```
void _fmpz_vec_scalar_fdiv_r_2exp(fmpz * vec1, const fmpz *
   vec2, slong len2, ulong exp)
```

Sets `(vec1, len2)` to the remainder of `(vec2, len2)` divided by `2^exp`, rounding down the quotient towards minus infinity whenever the division is not exact.

```
void _fmpz_vec_scalar_tdiv_q_fmpz(fmpz * vec1, const fmpz *
   vec2, slong len2, const fmpz_t c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by c, rounding towards zero whenever the division is not exact.

```
void _fmpz_vec_scalar_tdiv_q_si(fmpz * vec1, const fmpz *
   vec2, slong len2, slong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by c, rounding towards zero whenever the division is not exact.

```
void _fmpz_vec_scalar_tdiv_q_ui(fmpz * vec1, const fmpz *
   vec2, slong len2, ulong c)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by c, rounding towards zero whenever the division is not exact.

```
void _fmpz_vec_scalar_tdiv_q_2exp(fmpz * vec1, const fmpz *
   vec2, slong len2, ulong exp)
```

Sets `(vec1, len2)` to `(vec2, len2)` divided by `2^exp`, rounding down towards zero whenever the division is not exact.

```
void _fmpz_vec_scalar_addmul_fmpz(fmpz * vec1, const fmpz *
   vec2, slong len2, const fmpz_t c)
```

Adds `(vec2, len2)` times c to `(vec1, len2)`, where c is a `fmpz_t`.

```
void _fmpz_vec_scalar_addmul_si(fmpz * vec1, const fmpz *
   vec2, slong len2, slong c)
```

Adds `(vec2, len2)` times c to `(vec1, len2)`, where c is a `slong`.

```
void _fmpz_vec_scalar_addmul_si_2exp(fmpz * vec1, const
   fmpz * vec2, slong len2, slong c, ulong exp)
```

Adds `(vec2, len2)` times `c * 2^exp` to `(vec1, len2)`, where c is a `slong`.

```
void _fmpz_vec_scalar_submul_fmpz(fmpz * vec1, const fmpz *
   vec2, slong len2, const fmpz_t x)
```

Subtracts `(vec2, len2)` times c from `(vec1, len2)`, where c is a `fmpz_t`.

```
void _fmpz_vec_scalar_submul_si(fmpz * vec1, const fmpz *
   vec2, slong len2, slong c)
```

Subtracts `(vec2, len2)` times c from `(vec1, len2)`, where c is a `slong`.

```
void _fmpz_vec_scalar_submul_si_2exp(fmpz * vec1, const
   fmpz * vec2, slong len2, slong c, ulong e)
```

Subtracts `(vec2, len2)` times $c \times 2^e$ from `(vec1, len2)`, where c is a `slong`.

19.11 Sums and products

```
void _fmpz_vec_sum(fmpz_t res, const fmpz * vec, slong len)
```

Sets `res` to the sum of the entries in `(vec, len)`. Aliasing of `res` with the entries in `vec` is not permitted.

```
void _fmpz_vec_prod(fmpz_t res, const fmpz * vec, slong len)
```

Sets `res` to the product of the entries in `(vec, len)`. Aliasing of `res` with the entries in `vec` is not permitted. Uses binary splitting.

19.12 Reduction mod p

```
void _fmpz_vec_scalar_mod_fmpz(fmpz *res, const fmpz *vec,
   slong len, const fmpz_t p)
```

Reduces all entries in `(vec, len)` modulo $p > 0$.

```
void _fmpz_vec_scalar_smod_fmpz(fmpz *res, const fmpz *vec,
   slong len, const fmpz_t p)
```

Reduces all entries in `(vec, len)` modulo $p > 0$, choosing the unique representative in $(-p/2, p/2]$.

19.13 Gaussian content

```
void _fmpz_vec_content(fmpz_t res, const fmpz * vec, slong
   len)
```

Sets `res` to the non-negative content of the entries in `vec`. The content of a zero vector, including the case when the length is zero, is defined to be zero.

```
void _fmpz_vec_lcm(fmpz_t res, const fmpz * vec, slong len)
```

Sets `res` to the nonnegative least common multiple of the entries in `vec`. The least common multiple is zero if any entry in the vector is zero. The least common multiple of a length zero vector is defined to be one.

§20. fmpz_factor

Factorisation in **Z**

The `fmpz_factor` module is included automatically with `fmpz.h`. One should not try to include `fmpz_factor.h` directly.

20.1 Factoring integers

An integer may be represented in factored form using the `fmpz_factor_t` data structure. This consists of two `fmpz` vectors representing bases and exponents, respectively. Canonically, the bases will be prime numbers sorted in ascending order and the exponents will be positive.

A separate `int` field holds the sign, which may be -1, 0 or 1.

```
void fmpz_factor_init(fmpz_factor_t factor)
```

Initialises an `fmpz_factor_t` structure.

```
void fmpz_factor_clear(fmpz_factor_t factor)
```

Clears an `fmpz_factor_t` structure.

```
void fmpz_factor(fmpz_factor_t factor, const fmpz_t n)
```

Factors n into prime numbers. If n is zero or negative, the sign field of the `factor` object will be set accordingly.

This currently only uses trial division, falling back to `n_factor()` as soon as the number shrinks to a single limb.

```
void fmpz_factor_si(fmpz_factor_t factor, slong n)
```

Like `fmpz_factor`, but takes a machine integer n as input.

```
int fmpz_factor_trial_range(fmpz_factor_t factor, const
    fmpz_t n, ulong start, ulong num_primes)
```

Factors n into prime factors using trial division. If n is zero or negative, the sign field of the `factor` object will be set accordingly.

The algorithm starts with the given start index in the `flint_primes` table and uses at most `num_primes` primes from that point.

The function returns 1 if n is completely factored, otherwise it returns 0.

```
void fmpz_factor_expand_iterative(fmpz_t n, const
   fmpz_factor_t factor)
```

Evaluates an integer in factored form back to an `fmpz_t`.

This currently exponentiates the bases separately and multiplies them together one by one, although much more efficient algorithms exist.

```
int fmpz_factor_pp1(fmpz_t factor, const fmpz_t n, ulong
   B1, ulong B2_sqrt, ulong c)
```

Use Williams' $p + 1$ method to factor n, using a prime bound in stage 1 of `B1` and a prime limit in stage 2 of at least the square of `B2_sqrt`. If a factor is found, the function returns 1 and `factor` is set to the factor that is found. Otherwise, the function returns 0.

The value c should be a random value greater than 2. Successive calls to the function with different values of c give additional chances to factor n with roughly exponentially decaying probability of finding a factor which has been missed (if $p + 1$ or $p - 1$ is not smooth for any prime factors p of n then the function will not ever succeed).

§21. fmpz_mat

Matrices over **Z**

21.1 Introduction

The `fmpz_mat_t` data type represents dense matrices of multiprecision integers, implemented using `fmpz` vectors.

No automatic resizing is performed: in general, the user must provide matrices of correct dimensions for both input and output variables. Output variables are *not* allowed to be aliased with input variables unless otherwise noted.

Matrices are indexed from zero: an $m \times n$ matrix has rows of index $0, 1, \ldots, m-1$ and columns of index $0, 1, \ldots, n-1$. One or both of m and n may be zero.

Elements of a matrix can be read or written using the `fmpz_mat_entry` macro, which returns a reference to the entry at a given row and column index. This reference can be passed as an input or output `fmpz_t` variable to any function in the `fmpz` module for direct manipulation.

21.2 Simple example

The following example creates the 2×2 matrix A with value $2i+j$ at row i and column j, computes $B = A^2$, and prints both matrices.

```
#include "fmpz.h"
#include "fmpz_mat.h"
...
long i, j;
fmpz_mat_t A;
fmpz_mat_t B;
fmpz_mat_init(A, 2, 2);
fmpz_mat_init(B, 2, 2);
for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++)
        fmpz_set_ui(fmpz_mat_entry(A, i, j), 2*i+j);
fmpz_mat_mul(B, A, A);
flint_printf("A = \n");
fmpz_mat_print_pretty(A);
flint_printf("A^2 = \n");
fmpz_mat_print_pretty(B);
```

```
fmpz_mat_clear(A);
fmpz_mat_clear(B);
```

The output is:

```
A =
[[0 1]
[2 3]]
A^2 =
[[2 3]
[6 11]]
```

21.3 Memory management

`void fmpz_mat_init(fmpz_mat_t mat, slong rows, slong cols)`

Initialises a matrix with the given number of rows and columns for use.

`void fmpz_mat_clear(fmpz_mat_t mat)`

Clears the given matrix.

21.4 Basic assignment and manipulation

`void fmpz_mat_set(fmpz_mat_t mat1, const fmpz_mat_t mat2)`

Sets `mat1` to a copy of `mat2`. The dimensions of `mat1` and `mat2` must be the same.

`void fmpz_mat_init_set(fmpz_mat_t mat, const fmpz_mat_t src)`

Initialises the matrix `mat` to the same size as `src` and sets it to a copy of `src`.

`void fmpz_mat_swap(fmpz_mat_t mat1, fmpz_mat_t mat2)`

Swaps two matrices. The dimensions of `mat1` and `mat2` are allowed to be different.

`fmpz * fmpz_mat_entry(fmpz_mat_t mat, slong i, slong j)`

Returns a reference to the entry of `mat` at row i and column j. This reference can be passed as an input or output variable to any function in the `fmpz` module for direct manipulation.

Both i and j must not exceed the dimensions of the matrix.

This function is implemented as a macro.

`void fmpz_mat_zero(fmpz_mat_t mat)`

Sets all entries of `mat` to 0.

`void fmpz_mat_one(fmpz_mat_t mat)`

Sets `mat` to the unit matrix, having ones on the main diagonal and zeroes elsewhere. If `mat` is nonsquare, it is set to the truncation of a unit matrix.

21.5 Random matrix generation

`void fmpz_mat_randbits(fmpz_mat_t mat, flint_rand_t state, mp_bitcnt_t bits)`

Sets the entries of `mat` to random signed integers whose absolute values have the given number of binary bits.

```
void fmpz_mat_randtest(fmpz_mat_t mat, flint_rand_t state,
    mp_bitcnt_t bits)
```

Sets the entries of `mat` to random signed integers whose absolute values have a random number of bits up to the given number of bits inclusive.

```
void fmpz_mat_randintrel(fmpz_mat_t mat, flint_rand_t
    state, mp_bitcnt_t bits)
```

Sets `mat` to be a random *integer relations* matrix, with signed entries up to the given number of bits.

The number of columns of `mat` must be equal to one more than the number of rows. The format of the matrix is a set of random integers in the left hand column and an identity matrix in the remaining square submatrix.

```
void fmpz_mat_randsimdioph(fmpz_mat_t mat, flint_rand_t
    state, mp_bitcnt_t bits, mp_bitcnt_t bits2)
```

Sets `mat` to a random *simultaneous diophantine* matrix.

The matrix must be square. The top left entry is set to `2^bits2`. The remainder of that row is then set to signed random integers of the given number of binary bits. The remainder of the first column is zero. Running down the rest of the diagonal are the values `2^bits` with all remaining entries zero.

```
void fmpz_mat_randntrulike(fmpz_mat_t mat, flint_rand_t
    state, mp_bitcnt_t bits, ulong q)
```

Sets a square matrix `mat` of even dimension to a random *NTRU like* matrix.

The matrix is broken into four square submatrices. The top left submatrix is set to the identity. The bottom left submatrix is set to the zero matrix. The bottom right submatrix is set to q times the identity matrix. Finally the top right submatrix has the following format. A random vector h of length $r/2$ is created, with random signed entries of the given number of bits. Then entry (i, j) of the submatrix is set to $h[i + j \bmod r/2]$.

```
void fmpz_mat_randntrulike2(fmpz_mat_t mat, flint_rand_t
    state, mp_bitcnt_t bits, ulong q)
```

Sets a square matrix `mat` of even dimension to a random *NTRU like* matrix.

The matrix is broken into four square submatrices. The top left submatrix is set to q times the identity matrix. The top right submatrix is set to the zero matrix. The bottom right submatrix is set to the identity matrix. Finally the bottom left submatrix has the following format. A random vector h of length $r/2$ is created, with random signed entries of the given number of bits. Then entry (i, j) of the submatrix is set to $h[i + j \bmod r/2]$.

```
void fmpz_mat_randajtai(fmpz_mat_t mat, flint_rand_t state,
    double alpha)
```

Sets a square matrix `mat` to a random *ajtai* matrix. The diagonal entries (i, i) are set to a random entry in the range $[1, 2^{b-1}]$ inclusive where $b = \lfloor (2r - i)^\alpha \rfloor$ for some double parameter α. The entries below the diagonal in column i are set to a random entry in the range $(-2^b + 1, 2^b - 1)$ whilst the entries to the right of the diagonal in row i are set to zero.

```
int fmpz_mat_randpermdiag(fmpz_mat_t mat, flint_rand_t
    state, const fmpz * diag, slong n)
```

Sets `mat` to a random permutation of the rows and columns of a given diagonal matrix. The diagonal matrix is specified in the form of an array of the n initial entries on the main diagonal.

The return value is 0 or 1 depending on whether the permutation is even or odd.

```
void fmpz_mat_randrank(fmpz_mat_t mat, flint_rand_t state,
    slong rank, mp_bitcnt_t bits)
```

Sets `mat` to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the nonzero elements being random integers of the given bit size.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling `fmpz_mat_randops()`.

```
void fmpz_mat_randdet(fmpz_mat_t mat, flint_rand_t state,
    const fmpz_t det)
```

Sets `mat` to a random sparse matrix with minimal number of nonzero entries such that its determinant has the given value.

Note that the matrix will be zero if `det` is zero. In order to generate a non-zero singular matrix, the function `fmpz_mat_randrank()` can be used.

The matrix can be transformed into a dense matrix with unchanged determinant by subsequently calling `fmpz_mat_randops()`.

```
void fmpz_mat_randops(fmpz_mat_t mat, flint_rand_t state,
    slong count)
```

Randomises `mat` by performing elementary row or column operations. More precisely, at most `count` random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, the determinant) unchanged.

21.6 Input and output

```
int fmpz_mat_fprint(FILE * file, const fmpz_mat_t mat)
```

Prints the given matrix to the stream `file`. The format is the number of rows, a space, the number of columns, two spaces, then a space separated list of coefficients, one row after the other.

In case of success, returns a positive value; otherwise, returns a non-positive value.

```
int fmpz_mat_fprint_pretty(FILE * file, const fmpz_mat_t
    mat)
```

Prints the given matrix to the stream `file`. The format is an opening square bracket then on each line a row of the matrix, followed by a closing square bracket. Each row is written as an opening square bracket followed by a space separated list of coefficients followed by a closing square bracket.

In case of success, returns a positive value; otherwise, returns a non-positive value.

```
int fmpz_mat_print(const fmpz_mat_t mat)
```

Prints the given matrix to the stream `stdout`. For further details, see `fmpz_mat_fprint()`.

```
int fmpz_mat_print_pretty(const fmpz_mat_t mat)
```

Prints the given matrix to `stdout`. For further details, see `fmpz_mat_fprint_pretty()`.

```
int fmpz_mat_fread(FILE* file, fmpz_mat_t mat)
```

Reads a matrix from the stream `file`, storing the result in `mat`. The expected format is the number of rows, a space, the number of columns, two spaces, then a space separated list of coefficients, one row after the other.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

```
int fmpz_mat_read(fmpz_mat_t mat)
```

Reads a matrix from `stdin`, storing the result in `mat`.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

21.7 Comparison

```
int fmpz_mat_equal(const fmpz_mat_t mat1, const fmpz_mat_t
    mat2)
```

Returns a non-zero value if `mat1` and `mat2` have the same dimensions and entries, and zero otherwise.

```
int fmpz_mat_is_zero(const fmpz_mat_t mat)
```

Returns a non-zero value if all entries `mat` are zero, and otherwise returns zero.

```
int fmpz_mat_is_empty(const fmpz_mat_t mat)
```

Returns a non-zero value if the number of rows or the number of columns in `mat` is zero, and otherwise returns zero.

```
int fmpz_mat_is_square(const fmpz_mat_t mat)
```

Returns a non-zero value if the number of rows is equal to the number of columns in `mat`, and otherwise returns zero.

21.8 Transpose

```
void fmpz_mat_transpose(fmpz_mat_t B, const fmpz_mat_t A)
```

Sets B to A^T, the transpose of A. Dimensions must be compatible. A and B are allowed to be the same object if A is a square matrix.

21.9 Modular reduction and reconstruction

```
void fmpz_mat_get_nmod_mat(nmod_mat_t Amod, const
    fmpz_mat_t A)
```

Sets the entries of `Amod` to the entries of `A` reduced by the modulus of `Amod`.

```
void fmpz_mat_set_nmod_mat(fmpz_mat_t A, const nmod_mat_t
    Amod)
```

Sets the entries of `Amod` to the residues in `Amod`, normalised to the interval $-m/2 <= r < m/2$ where m is the modulus.

```
void fmpz_mat_set_nmod_mat_unsigned(fmpz_mat_t A, const
    nmod_mat_t Amod)
```

Sets the entries of `Amod` to the residues in `Amod`, normalised to the interval $0 <= r < m$ where m is the modulus.

```
void fmpz_mat_CRT_ui(fmpz_mat_t res, const fmpz_mat_t mat1,
    const fmpz_t m1, const nmod_mat_t mat2, int sign)
```

Given `mat1` with entries modulo `m` and `mat2` with modulus n, sets `res` to the CRT reconstruction modulo mn with entries satisfying $-mn/2 <= c < mn/2$ (if sign = 1) or $0 <= c < mn$ (if sign = 0).

```
void fmpz_mat_multi_mod_ui_precomp(nmod_mat_t * residues,
    slong nres, const fmpz_mat_t mat, fmpz_comb_t comb,
    fmpz_comb_temp_t temp)
```

Sets each of the `nres` matrices in `residues` to `mat` reduced modulo the modulus of the respective matrix, given precomputed `comb` and `comb_temp` structures.

```
void fmpz_mat_multi_mod_ui(nmod_mat_t * residues, slong
    nres, const fmpz_mat_t mat)
```

Sets each of the `nres` matrices in `residues` to `mat` reduced modulo the modulus of the respective matrix.

This function is provided for convenience purposes. For reducing or reconstructing multiple integer matrices over the same set of moduli, it is faster to use `fmpz_mat_multi_mod_precomp`.

```
void fmpz_mat_multi_CRT_ui_precomp(fmpz_mat_t mat,
    nmod_mat_t * const residues, slong nres, fmpz_comb_t
    comb, fmpz_comb_temp_t temp, int sign)
```

Reconstructs `mat` from its images modulo the `nres` matrices in `residues`, given precomputed `comb` and `comb_temp` structures.

```
void fmpz_mat_multi_CRT_ui(fmpz_mat_t mat, nmod_mat_t *
    const residues, slong nres, int sign)
```

Reconstructs `mat` from its images modulo the `nres` matrices in `residues`.

This function is provided for convenience purposes. For reducing or reconstructing multiple integer matrices over the same set of moduli, it is faster to use `fmpz_mat_multi_CRT_ui_precomp`.

21.10 Addition and subtraction

```
void fmpz_mat_add(fmpz_mat_t C, const fmpz_mat_t A, const
    fmpz_mat_t B)
```

Sets `C` to the elementwise sum $A + B$. All inputs must be of the same size. Aliasing is allowed.

```
void fmpz_mat_sub(fmpz_mat_t C, const fmpz_mat_t A, const
    fmpz_mat_t B)
```

Sets `C` to the elementwise difference $A - B$. All inputs must be of the same size. Aliasing is allowed.

```
void fmpz_mat_neg(fmpz_mat_t B, const fmpz_mat_t A)
```

Sets `B` to the elementwise negation of `A`. Both inputs must be of the same size. Aliasing is allowed.

21.11 Matrix-scalar arithmetic

```
void fmpz_mat_scalar_mul_si(fmpz_mat_t B, const fmpz_mat_t
   A, slong c)

void fmpz_mat_scalar_mul_ui(fmpz_mat_t B, const fmpz_mat_t
   A, ulong c)

void fmpz_mat_scalar_mul_fmpz(fmpz_mat_t B, const
   fmpz_mat_t A, const fmpz_t c)
```

Set `A = B*c` where `B` is an `fmpz_mat_t` and `c` is a scalar respectively of type `slong`, `ulong`, or `fmpz_t`. The dimensions of `A` and `B` must be compatible.

```
void fmpz_mat_scalar_addmul_si(fmpz_mat_t B, const
   fmpz_mat_t A, slong c)

void fmpz_mat_scalar_addmul_ui(fmpz_mat_t B, const
   fmpz_mat_t A, ulong c)

void fmpz_mat_scalar_addmul_fmpz(fmpz_mat_t B, const
   fmpz_mat_t A, const fmpz_t c)
```

Set `A = A + B*c` where `B` is an `fmpz_mat_t` and `c` is a scalar respectively of type `slong`, `ulong`, or `fmpz_t`. The dimensions of `A` and `B` must be compatible.

```
void fmpz_mat_scalar_submul_si(fmpz_mat_t B, const
   fmpz_mat_t A, slong c)

void fmpz_mat_scalar_submul_ui(fmpz_mat_t B, const
   fmpz_mat_t A, ulong c)

void fmpz_mat_scalar_submul_fmpz(fmpz_mat_t B, const
   fmpz_mat_t A, const fmpz_t c)
```

Set `A = A - B*c` where `B` is an `fmpz_mat_t` and `c` is a scalar respectively of type `slong`, `ulong`, or `fmpz_t`. The dimensions of `A` and `B` must be compatible.

```
void fmpz_mat_scalar_addmul_nmod_mat_ui(fmpz_mat_t B, const
   nmod_mat_t A, ulong c)

void fmpz_mat_scalar_addmul_nmod_mat_fmpz(fmpz_mat_t B,
   const nmod_mat_t A, const fmpz_t c)
```

Set `A = A + B*c` where `B` is an `nmod_mat_t` and `c` is a scalar respectively of type `ulong` or `fmpz_t`. The dimensions of `A` and `B` must be compatible.

```
void fmpz_mat_scalar_divexact_si(fmpz_mat_t B, const
   fmpz_mat_t A, slong c)

void fmpz_mat_scalar_divexact_ui(fmpz_mat_t B, const
   fmpz_mat_t A, ulong c)

void fmpz_mat_scalar_divexact_fmpz(fmpz_mat_t B, const
   fmpz_mat_t A, const fmpz_t c)
```

Set `A = B / c`, where `B` is an `fmpz_mat_t` and `c` is a scalar respectively of type `slong`, `ulong`, or `fmpz_t`, which is assumed to divide all elements of `B` exactly.

21.12 Matrix multiplication

```
void fmpz_mat_mul(fmpz_mat_t C, const fmpz_mat_t A, const
    fmpz_mat_t B)
```

Sets `C` to the matrix product $C = AB$. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

This function automatically switches between classical and multimodular multiplication, based on a heuristic comparison of the dimensions and entry sizes.

```
void fmpz_mat_mul_classical(fmpz_mat_t C, const fmpz_mat_t
    A, const fmpz_mat_t B)
```

Sets `C` to the matrix product $C = AB$ computed using classical matrix algorithm.

The matrices must have compatible dimensions for matrix multiplication. No aliasing is allowed.

```
void _fmpz_mat_mul_multi_mod(fmpz_mat_t C, const fmpz_mat_t
    A, const fmpz_mat_t B, mp_bitcnt_t bits)

void fmpz_mat_mul_multi_mod(fmpz_mat_t C, const fmpz_mat_t
    A, const fmpz_mat_t B)
```

Sets `C` to the matrix product $C = AB$ computed using a multimodular algorithm. C is computed modulo several small prime numbers and reconstructed using the Chinese Remainder Theorem. This generally becomes more efficient than classical multiplication for large matrices.

The `bits` parameter is a bound for the bit size of largest element of C, or twice the absolute value of the largest element if any elements of C are negative. The function `fmpz_mat_mul_multi_mod` calculates a rigorous bound automatically. If the default bound is too pessimistic, `_fmpz_mat_mul_multi_mod` can be used with a custom bound.

The matrices must have compatible dimensions for matrix multiplication. No aliasing is allowed.

```
void fmpz_mat_sqr(fmpz_mat_t B, const fmpz_mat_t A)
```

Sets `B` to the square of the matrix `A`, which must be a square matrix. Aliasing is allowed.

```
void fmpz_mat_pow(fmpz_mat_t B, const fmpz_mat_t A, ulong e)
```

Sets `B` to the matrix `A` raised to the power `e`, where `A` must be a square matrix. Aliasing is allowed.

21.13 Inverse

```
int fmpz_mat_inv(fmpz_mat_t Ainv, fmpz_t den, const
    fmpz_mat_t A)
```

Sets (`Ainv`, `den`) to the inverse matrix of `A`. Returns 1 if `A` is nonsingular and 0 if `A` is singular. Aliasing of `Ainv` and `A` is allowed.

The denominator is not guaranteed to be minimal, but is guaranteed to be a divisor of the determinant of `A`.

This function uses a direct formula for matrices of size two or less, and otherwise solves for the identity matrix using fraction-free LU decomposition.

21.14 Trace

```
void fmpz_mat_trace(fmpz_t trace, const fmpz_mat_t mat)
```

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

21.15 Determinant

```
void fmpz_mat_det(fmpz_t det, const fmpz_mat_t A)
```

Sets `det` to the determinant of the square matrix A. The matrix of dimension 0×0 is defined to have determinant 1.

This function automatically chooses between `fmpz_mat_det_cofactor`, `fmpz_mat_det_bareiss`, `fmpz_mat_det_modular` and `fmpz_mat_det_modular_accelerated` (with `proved` = 1), depending on the size of the matrix and its entries.

```
void fmpz_mat_det_cofactor(fmpz_t det, const fmpz_mat_t A)
```

Sets `det` to the determinant of the square matrix A computed using direct cofactor expansion. This function only supports matrices up to size 4×4.

```
void fmpz_mat_det_bareiss(fmpz_t det, const fmpz_mat_t A)
```

Sets `det` to the determinant of the square matrix A computed using the Bareiss algorithm. A copy of the input matrix is row reduced using fraction-free Gaussian elimination, and the determinant is read off from the last element on the main diagonal.

```
void fmpz_mat_det_modular(fmpz_t det, const fmpz_mat_t A,
    int proved)
```

Sets `det` to the determinant of the square matrix A (if `proved` = 1), or a probabilistic value for the determinant (`proved` = 0), computed using a multimodular algorithm.

The determinant is computed modulo several small primes and reconstructed using the Chinese Remainder Theorem. With `proved` = 1, sufficiently many primes are chosen to satisfy the bound computed by `fmpz_mat_det_bound`. With `proved` = 0, the determinant is considered determined if it remains unchanged modulo several consecutive primes (currently if their product exceeds 2^{100}).

```
void fmpz_mat_det_modular_accelerated(fmpz_t det, const
    fmpz_mat_t A, int proved)
```

Sets `det` to the determinant of the square matrix A (if `proved` = 1), or a probabilistic value for the determinant (`proved` = 0), computed using a multimodular algorithm.

This function uses the same basic algorithm as `fmpz_mat_det_modular`, but instead of computing $\det(A)$ directly, it generates a divisor d of $\det(A)$ and then computes $x = \det(A)/d$ modulo several small primes not dividing d. This typically accelerates the computation by requiring fewer primes for large matrices, since d with high probability will be nearly as large as the determinant. This trick is described in [1].

```
void fmpz_mat_det_modular_given_divisor(fmpz_t det, const
    fmpz_mat_t A, const fmpz_t d, int proved)
```

Given a positive divisor d of $\det(A)$, sets `det` to the determinant of the square matrix A (if `proved` $= 1$), or a probabilistic value for the determinant (`proved` $= 0$), computed using a multimodular algorithm.

```
void fmpz_mat_det_bound(fmpz_t bound, const fmpz_mat_t A)
```

Sets `bound` to a nonnegative integer B such that $|\det(A)| \leq B$. Assumes A to be a square matrix. The bound is computed from the Hadamard inequality $|\det(A)| \leq \prod \|a_i\|_2$ where the product is taken over the rows a_i of A.

```
void fmpz_mat_det_divisor(fmpz_t d, const fmpz_mat_t A)
```

Sets d to some positive divisor of the determinant of the given square matrix A, if the determinant is nonzero. If $|\det(A)| = 0$, d will always be set to zero.

A divisor is obtained by solving $Ax = b$ for an arbitrarily chosen right-hand side b using Dixon's algorithm and computing the least common multiple of the denominators in x. This yields a divisor d such that $|\det(A)|/d$ is tiny with very high probability.

21.16 Characteristic polynomial

```
void _fmpz_mat_charpoly(fmpz * cp, const fmpz_mat_t mat)
```

Sets `(cp, n+1)` to the characteristic polynomial of an $n \times n$ square matrix.

```
void fmpz_mat_charpoly(fmpz_poly_t cp, const fmpz_mat_t mat)
```

Computes the characteristic polynomial of length $n + 1$ of an $n \times n$ square matrix.

21.17 Rank

```
slong fmpz_mat_rank(const fmpz_mat_t A)
```

Returns the rank, that is, the number of linearly independent columns (equivalently, rows), of A. The rank is computed by row reducing a copy of A.

21.18 Nonsingular solving

The following functions allow solving matrix-matrix equations $AX = B$ where the system matrix A is square and has full rank. The solving is implicitly done over the field of rational numbers: except where otherwise noted, an integer matrix $\hat{X}$ and a separate denominator d (`den`) are computed such that $A(\hat{X}/d) = b$, equivalently such that $A\hat{X} = bd$ holds over the integers.

No guarantee is made that the numerators and denominator are reduced to lowest terms, but the denominator is always guaranteed to be a divisor of the determinant of A. If A is singular, `den` will be set to zero and the elements of the solution vector or matrix will have undefined values. No aliasing is allowed between arguments.

```
int fmpz_mat_solve(fmpz_mat_t X, fmpz_t den, const
    fmpz_mat_t A, const fmpz_mat_t B)
```

Solves the equation $AX = B$ for nonsingular A. More precisely, computes (`X`, `den`) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

This function uses Cramer's rule for small systems and fraction-free LU decomposition followed by fraction-free forward and back substitution for larger systems.

Note that for very large systems, it is faster to compute a modular solution using `fmpz_mat_solve_dixon`.

```
int fmpz_mat_solve_fflu(fmpz_mat_t X, fmpz_t den, const
    fmpz_mat_t A, const fmpz_mat_t B)
```

Solves the equation $AX = B$ for nonsingular A. More precisely, computes (`X`, `den`) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

```
void fmpz_mat_solve_fflu_precomp(fmpz_mat_t X, const slong
    * perm, const fmpz_mat_t FFLU, const fmpz_mat_t B)
```

Performs fraction-free forward and back substitution given a precomputed fraction-free LU decomposition and corresponding permutation.

```
int fmpz_mat_solve_cramer(fmpz_mat_t X, fmpz_t den, const
    fmpz_mat_t A, const fmpz_mat_t B)
```

Solves the equation $AX = B$ for nonsingular A. More precisely, computes (`X`, `den`) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular.

Uses Cramer's rule. Only systems of size up to 3×3 are allowed.

```
void fmpz_mat_solve_bound(fmpz_t N, fmpz_t D, const
    fmpz_mat_t A, const fmpz_mat_t B)
```

Assuming that A is nonsingular, computes integers N and D such that the reduced numerators and denominators n/d in $A^{-1}B$ satisfy the bounds $0 \le |n| \le N$ and $0 \le d \le D$.

```
int fmpz_mat_solve_dixon(fmpz_mat_t X, fmpz_t M, const
    fmpz_mat_t A, const fmpz_mat_t B)
```

Solves $AX = B$ given a nonsingular square matrix A and a matrix B of compatible dimensions, using a modular algorithm. In particular, Dixon's p-adic lifting algorithm is used (currently a non-adaptive version). This is generally the preferred method for large dimensions.

More precisely, this function computes an integer M and an integer matrix X such that $AX = B \bmod M$ and such that all the reduced numerators and denominators of the elements $x = p/q$ in the full solution satisfy $2|p|q < B$. As such, the explicit rational solution matrix can be recovered uniquely by passing the output of this function to `fmpq_mat_set_fmpz_mat_mod`.

A nonzero value is returned if A is nonsingular. If A is singular, zero is returned and the values of the output variables will be undefined.

Aliasing between input and output matrices is allowed.

21.19 Row reduction

```
slong fmpz_mat_find_pivot_any(const fmpz_mat_t mat, slong
    start_row, slong end_row, slong c)
```

Attempts to find a pivot entry for row reduction. Returns a row index r between `start_row` (inclusive) and `stop_row` (exclusive) such that column c in `mat` has a nonzero entry on row r, or returns -1 if no such entry exists.

This implementation simply chooses the first nonzero entry from it encounters. This is likely to be a nearly optimal choice if all entries in the matrix have roughly the same size, but can lead to unnecessary coefficient growth if the entries vary in size.

```
slong fmpz_mat_fflu(fmpz_mat_t B, fmpz_poly_t den, slong *
    perm, const fmpz_mat_t A, int rank_check)
```

Uses fraction-free Gaussian elimination to set (`B`, `den`) to a fraction-free LU decomposition of `A` and returns the rank of `A`. Aliasing of `A` and `B` is allowed.

Pivot elements are chosen with `fmpz_mat_find_pivot_any`. If `perm` is non-`NULL`, the permutation of rows in the matrix will also be applied to `perm`.

If `rank_check` is set, the function aborts and returns 0 if the matrix is detected not to have full rank without completing the elimination.

The denominator `den` is set to $\pm \det(S)$ where S is an appropriate submatrix of A ($S = A$ if A is square) and the sign is decided by the parity of the permutation. Note that the determinant is not generally the minimal denominator.

The fraction-free LU decomposition is defined in [24].

```
slong fmpz_mat_rref(fmpz_mat_t B, fmpz_t den, const
    fmpz_mat_t A)
```

Sets (`B`, `den`) to the reduced row echelon form of `A` and returns the rank of `A`. Aliasing of `A` and `B` is allowed.

The algorithm proceeds by first computing a row echelon form using `fmpz_mat_fflu`. Letting the upper part of this matrix be $(U|V)P$ where U is full rank upper triangular and P is a permutation matrix, we obtain the rref by setting V to $U^{-1}V$ using back substitution. Scaling each completed row in the back substitution to the denominator `den`, we avoid introducing new fractions. This strategy is equivalent to the fraction-free Gauss-Jordan elimination in [24], but faster since only the part V corresponding to the null space has to be updated.

The denominator `den` is set to $\pm \det(S)$ where S is an appropriate submatrix of A ($S = A$ if A is square). Note that the determinant is not generally the minimal denominator.

21.20 Modular gaussian elimination

```
slong fmpz_mat_rref_mod(slong * perm, fmpz_mat_t A, const
    fmpz_t p)
```

Uses fraction-free Gauss-Jordan elimination to set `A` to its reduced row echelon form and returns the rank of `A`. All computations are done modulo p.

Pivot elements are chosen with `fmpz_mat_find_pivot_any`. If `perm` is non-`NULL`, the permutation of rows in the matrix will also be applied to `perm`.

21.21 Nullspace

```
slong fmpz_mat_nullspace(fmpz_mat_t B, const fmpz_mat_t A)
```

Computes a basis for the right rational nullspace of A and returns the dimension of the nullspace (or nullity). B is set to a matrix with linearly independent columns and maximal rank such that $AB = 0$ (i.e. $Ab = 0$ for each column b in B), and the rank of B is returned.

In general, the entries in B will not be minimal: in particular, the pivot entries in B will generally differ from unity. B must be allocated with sufficient space to represent the result (at most $n \times n$ where n is the number of column of A).

21.22 Echelon form

```
slong fmpz_mat_rref_fraction_free(slong * perm, fmpz_mat_t
   B, fmpz_t den, const fmpz_mat_t A)
```

Computes an integer matrix `B` and an integer `den` such that `B / den` is the unique row reduced echelon form (RREF) of `A` and returns the rank, i.e. the number of nonzero rows in `B`.

Aliasing of `B` and `A` is allowed, with an in-place computation being more efficient. The size of `B` must be the same as that of `A`.

The permutation order will be written to `perm` unless this argument is `NULL`. That is, row `i` of the output matrix will correspond to row `perm[i]` of the input matrix.

The denominator will always be a divisor of the determinant of (some submatrix of) A, but is not guaranteed to be minimal or canonical in any other sense.

§22. fmpz_poly

Polynomials over **Z**

22.1 Introduction

The `fmpz_poly_t` data type represents elements of $\mathbf{Z}[x]$. The `fmpz_poly` module provides routines for memory management, basic arithmetic, and conversions from or to other types.

Each coefficient of an `fmpz_poly_t` is an integer of the FLINT `fmpz_t` type. There are two advantages of this model. Firstly, the `fmpz_t` type is memory managed, so the user can manipulate individual coefficients of a polynomial without having to deal with tedious memory management. Secondly, a coefficient of an `fmpz_poly_t` can be changed without changing the size of any of the other coefficients.

Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

22.2 Simple example

The following example computes the square of the polynomial $5x^3 - 1$.

```
#include "fmpz_poly.h"
...
fmpz_poly_t x, y;
fmpz_poly_init(x);
fmpz_poly_init(y);
fmpz_poly_set_coeff_ui(x, 3, 5);
fmpz_poly_set_coeff_si(x, 0, -1);
fmpz_poly_mul(y, x, x);
fmpz_poly_print(x); flint_printf("\n");
fmpz_poly_print(y); flint_printf("\n");
fmpz_poly_clear(x);
fmpz_poly_clear(y);
```

The output is:

```
4  -1 0 0 5
7  1 0 0 -10 0 0 25
```

22.3 Definition of the fmpz_poly_t type

The `fmpz_poly_t` type is a typedef for an array of length 1 of `fmpz_poly_struct`'s. This permits passing parameters of type `fmpz_poly_t` by reference in a manner similar to the way GMP integers of type `mpz_t` can be passed by reference.

In reality one never deals directly with the `struct` and simply deals with objects of type `fmpz_poly_t`. For simplicity we will think of an `fmpz_poly_t` as a `struct`, though in practice to access fields of this `struct`, one needs to dereference first, e.g. to access the `length` field of an `fmpz_poly_t` called `poly1` one writes `poly1->length`.

An `fmpz_poly_t` is said to be *normalised* if either `length` is zero, or if the leading coefficient of the polynomial is non-zero. All `fmpz_poly` functions expect their inputs to be normalised, and unless otherwise specified they produce output that is normalised.

It is recommended that users do not access the fields of an `fmpz_poly_t` or its coefficient data directly, but make use of the functions designed for this purpose, detailed below.

Functions in `fmpz_poly` do all the memory management for the user. One does not need to specify the maximum length or number of limbs per coefficient in advance before using a polynomial object. FLINT reallocates space automatically as the computation proceeds, if more space is required. Each coefficient is also managed separately, being resized as needed, independently of the other coefficients.

We now describe the functions available in `fmpz_poly`.

22.4 Memory management

```
void fmpz_poly_init(fmpz_poly_t poly)
```

Initialises `poly` for use, setting its length to zero. A corresponding call to `fmpz_poly_clear()` must be made after finishing with the `fmpz_poly_t` to free the memory used by the polynomial.

```
void fmpz_poly_init2(fmpz_poly_t poly, slong alloc)
```

Initialises `poly` with space for at least `alloc` coefficients and sets the length to zero. The allocated coefficients are all set to zero.

```
void fmpz_poly_realloc(fmpz_poly_t poly, slong alloc)
```

Reallocates the given polynomial to have space for `alloc` coefficients. If `alloc` is zero the polynomial is cleared and then reinitialised. If the current length is greater than `alloc` the polynomial is first truncated to length `alloc`.

```
void fmpz_poly_fit_length(fmpz_poly_t poly, slong len)
```

If `len` is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least `len` coefficients. No data is lost when calling this function.

The function efficiently deals with the case where `fit_length` is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

```
void fmpz_poly_clear(fmpz_poly_t poly)
```

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

```
void _fmpz_poly_normalise(fmpz_poly_t poly)
```

Sets the length of `poly` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

```
void _fmpz_poly_set_length(fmpz_poly_t poly, slong newlen)
```

Demotes the coefficients of `poly` beyond `newlen` and sets the length of `poly` to `newlen`.

22.5 Polynomial parameters

```
slong fmpz_poly_length(const fmpz_poly_t poly)
```

Returns the length of `poly`. The zero polynomial has length zero.

```
slong fmpz_poly_degree(const fmpz_poly_t poly)
```

Returns the degree of `poly`, which is one less than its length.

22.6 Assignment and basic manipulation

```
void fmpz_poly_set(fmpz_poly_t poly1, const fmpz_poly_t
   poly2)
```

Sets `poly1` to equal `poly2`.

```
void fmpz_poly_set_si(fmpz_poly_t poly, slong c)
```

Sets `poly` to the signed integer `c`.

```
void fmpz_poly_set_ui(fmpz_poly_t poly, ulong c)
```

Sets `poly` to the unsigned integer `c`.

```
void fmpz_poly_set_fmpz(fmpz_poly_t poly, const fmpz_t c)
```

Sets `poly` to the integer `c`.

```
void fmpz_poly_set_mpz(fmpz_poly_t poly, const mpz_t c)
```

Sets `poly` to the integer `c`.

```
int _fmpz_poly_set_str(fmpz * poly, const char * str)
```

Sets `poly` to the polynomial encoded in the null-terminated string `str`. Assumes that `poly` is allocated as a sufficiently large array suitable for the number of coefficients present in `str`.

Returns 0 if no error occurred. Otherwise, returns a non-zero value, in which case the resulting value of `poly` is undefined. If `str` is not null-terminated, calling this method might result in a segmentation fault.

```
int fmpz_poly_set_str(fmpz_poly_t poly, const char * str)
```

Imports a polynomial from a null-terminated string. If the string `str` represents a valid polynomial returns 1, otherwise returns 0.

Returns 0 if no error occurred. Otherwise, returns a non-zero value, in which case the resulting value of `poly` is undefined. If `str` is not null-terminated, calling this method might result in a segmentation fault.

```
char * _fmpz_poly_get_str(const fmpz * poly, slong len)
```

Returns the plain FLINT string representation of the polynomial `(poly, len)`.

```
char * fmpz_poly_get_str(const fmpz_poly_t poly)
```

Returns the plain FLINT string representation of the polynomial `poly`.

```
char * _fmpz_poly_get_str_pretty(const fmpz * poly, slong
   len, const char * x)
```

Returns a pretty representation of the polynomial `(poly, len)` using the null-terminated string `x` as the variable name.

```
char * fmpz_poly_get_str_pretty(const fmpz_poly_t poly,
   const char * x)
```

Returns a pretty representation of the polynomial `poly` using the null-terminated string `x` as the variable name.

```
void fmpz_poly_zero(fmpz_poly_t poly)
```

Sets `poly` to the zero polynomial.

```
void fmpz_poly_one(fmpz_poly_t poly)
```

Sets `poly` to the constant polynomial one.

```
void fmpz_poly_zero_coeffs(fmpz_poly_t poly, slong i, slong
   j)
```

Sets the coefficients of $x^i, \ldots, x^{j-1}$ to zero.

```
void fmpz_poly_swap(fmpz_poly_t poly1, fmpz_poly_t poly2)
```

Swaps `poly1` and `poly2`. This is done efficiently without copying data by swapping pointers, etc.

```
void _fmpz_poly_reverse(fmpz * res, const fmpz * poly,
   slong len, slong n)
```

Sets `(res, n)` to the reverse of `(poly, n)`, where `poly` is in fact an array of length `len`. Assumes that `0 < len <= n`. Supports aliasing of `res` and `poly`, but the behaviour is undefined in case of partial overlap.

```
void fmpz_poly_reverse(fmpz_poly_t res, const fmpz_poly_t
   poly, slong n)
```

This function considers the polynomial `poly` to be of length n, notionally truncating and zero padding if required, and reverses the result. Since the function normalises its result `res` may be of length less than n.

```
void fmpz_poly_truncate(fmpz_poly_t poly, slong newlen)
```

If the current length of `poly` is greater than `newlen`, it is truncated to have the given length. Discarded coefficients are not necessarily set to zero.

22.7 Randomisation

```
void fmpz_poly_randtest(fmpz_poly_t f, flint_rand_t state,
   slong len, mp_bitcnt_t bits)
```

Sets f to a random polynomial with up to the given length and where each coefficient has up to the given number of bits. The coefficients are signed randomly. One must call `flint_randinit()` before calling this function.

```
void fmpz_poly_randtest_unsigned(fmpz_poly_t f,
   flint_rand_t state, slong len, mp_bitcnt_t bits)
```

Sets f to a random polynomial with up to the given length and where each coefficient has up to the given number of bits. One must call `flint_randinit()` before calling this function.

```
void fmpz_poly_randtest_not_zero(fmpz_poly_t f,
   flint_rand_t state, slong len, mp_bitcnt_t bits)
```

As for `fmpz_poly_randtest()` except that `len` and bits may not be zero and the polynomial generated is guaranteed not to be the zero polynomial. One must call `flint_randinit()` before calling this function.

22.8 Getting and setting coefficients

```
void fmpz_poly_get_coeff_fmpz(fmpz_t x, const fmpz_poly_t
   poly, slong n)
```

Sets x to the nth coefficient of `poly`. Coefficient numbering is from zero and if n is set to a value beyond the end of the polynomial, zero is returned.

```
slong fmpz_poly_get_coeff_si(const fmpz_poly_t poly, slong
   n)
```

Returns coefficient n of `poly` as a `slong`. The result is undefined if the value does not fit into a `slong`. Coefficient numbering is from zero and if n is set to a value beyond the end of the polynomial, zero is returned.

```
ulong fmpz_poly_get_coeff_ui(const fmpz_poly_t poly, slong
   n)
```

Returns coefficient n of `poly` as a `ulong`. The result is undefined if the value does not fit into a `ulong`. Coefficient numbering is from zero and if n is set to a value beyond the end of the polynomial, zero is returned.

```
fmpz * fmpz_poly_get_coeff_ptr(const fmpz_poly_t poly,
   slong n)
```

Returns a reference to the coefficient of x^n in the polynomial, as an `fmpz *`. This function is provided so that individual coefficients can be accessed and operated on by functions in the `fmpz` module. This function does not make a copy of the data, but returns a reference to the actual coefficient.

Returns `NULL` when n exceeds the degree of the polynomial.

This function is implemented as a macro.

```
fmpz * fmpz_poly_lead(const fmpz_poly_t poly)
```

Returns a reference to the leading coefficient of the polynomial, as an `fmpz *`. This function is provided so that the leading coefficient can be easily accessed and operated on by functions in the `fmpz` module. This function does not make a copy of the data, but returns a reference to the actual coefficient.

Returns `NULL` when the polynomial is zero.

This function is implemented as a macro.

```
void fmpz_poly_set_coeff_fmpz(fmpz_poly_t poly, slong n,
    const fmpz_t x)
```

Sets coefficient n of `poly` to the `fmpz` value `x`. Coefficient numbering starts from zero and if n is beyond the current length of `poly` then the polynomial is extended and zero coefficients inserted if necessary.

```
void fmpz_poly_set_coeff_si(fmpz_poly_t poly, slong n,
    slong x)
```

Sets coefficient n of `poly` to the `slong` value `x`. Coefficient numbering starts from zero and if n is beyond the current length of `poly` then the polynomial is extended and zero coefficients inserted if necessary.

```
void fmpz_poly_set_coeff_ui(fmpz_poly_t poly, slong n,
    ulong x)
```

Sets coefficient n of `poly` to the `ulong` value `x`. Coefficient numbering starts from zero and if n is beyond the current length of `poly` then the polynomial is extended and zero coefficients inserted if necessary.

22.9 Comparison

```
int fmpz_poly_equal(const fmpz_poly_t poly1, const
    fmpz_poly_t poly2)
```

Returns 1 if `poly1` is equal to `poly2`, otherwise returns 0. The polynomials are assumed to be normalised.

```
int fmpz_poly_is_zero(const fmpz_poly_t poly)
```

Returns 1 if the polynomial is zero and 0 otherwise.

This function is implemented as a macro.

```
int fmpz_poly_is_one(const fmpz_poly_t poly)
```

Returns 1 if the polynomial is one and 0 otherwise.

```
int fmpz_poly_is_unit(const fmpz_poly_t poly)
```

Returns 1 is the polynomial is the constant polynomial ± 1, and 0 otherwise.

22.10 Addition and subtraction

```
void _fmpz_poly_add(fmpz * res, const fmpz * poly1, slong
    len1, const fmpz * poly2, slong len2)
```

Sets `res` to the sum of `(poly1, len1)` and `(poly2, len2)`. It is assumed that `res` has sufficient space for the longer of the two polynomials.

```
void fmpz_poly_add(fmpz_poly_t res, const fmpz_poly_t
    poly1, const fmpz_poly_t poly2)
```

Sets `res` to the sum of `poly1` and `poly2`.

```
void _fmpz_poly_sub(fmpz * res, const fmpz * poly1, slong
    len1, const fmpz * poly2, slong len2)
```

Sets `res` to `(poly1, len1)` minus `(poly2, len2)`. It is assumed that `res` has sufficient space for the longer of the two polynomials.

```
void fmpz_poly_sub(fmpz_poly_t res, const fmpz_poly_t
   poly1, const fmpz_poly_t poly2)
```

Sets `res` to `poly1` minus `poly2`.

```
void fmpz_poly_neg(fmpz_poly_t res, const fmpz_poly_t poly)
```

Sets `res` to `-poly`.

22.11 Scalar multiplication and division

```
void fmpz_poly_scalar_mul_fmpz(fmpz_poly_t poly1, const
   fmpz_poly_t poly2, const fmpz_t x)
```

Sets `poly1` to `poly2` times x.

```
void fmpz_poly_scalar_mul_mpz(fmpz_poly_t poly1, const
   fmpz_poly_t poly2, const mpz_t x)
```

Sets `poly1` to `poly2` times the `mpz_t` x.

```
void fmpz_poly_scalar_mul_si(fmpz_poly_t poly1, fmpz_poly_t
   poly2, slong x)
```

Sets `poly1` to `poly2` times the signed `slong x`.

```
void fmpz_poly_scalar_mul_ui(fmpz_poly_t poly1, fmpz_poly_t
   poly2, ulong x)
```

Sets `poly1` to `poly2` times the `ulong x`.

```
void fmpz_poly_scalar_mul_2exp(fmpz_poly_t poly1,
   fmpz_poly_t poly2, ulong exp)
```

Sets `poly1` to `poly2` times `2^exp`.

```
void fmpz_poly_scalar_addmul_fmpz(fmpz_poly_t poly1, const
   fmpz_poly_t poly2, const fmpz_t x)
```

Sets `poly` to `poly1 + x * poly2`.

```
void fmpz_poly_scalar_submul_fmpz(fmpz_poly_t poly1, const
   fmpz_poly_t poly2, const fmpz_t x)
```

Sets `poly` to `poly1 - x * poly2`.

```
void fmpz_poly_scalar_fdiv_fmpz(fmpz_poly_t poly1, const
   fmpz_poly_t poly2, const fmpz_t x)
```

Sets `poly1` to `poly2` divided by the `fmpz_t x`, rounding coefficients down toward $-\infty$.

```
void fmpz_poly_scalar_fdiv_mpz(fmpz_poly_t poly1, const
   fmpz_poly_t poly2, const mpz_t x)
```

Sets `poly1` to `poly2` divided by the `mpz_t x`, rounding coefficients down toward $-\infty$.

```
void fmpz_poly_scalar_fdiv_si(fmpz_poly_t poly1,
   fmpz_poly_t poly2, slong x)
```

Sets `poly1` to `poly2` divided by the `slong x`, rounding coefficients down toward $-\infty$.

```
void fmpz_poly_scalar_fdiv_ui(fmpz_poly_t poly1,
   fmpz_poly_t poly2, ulong x)
```

Sets `poly1` to `poly2` divided by the `ulong x`, rounding coefficients down toward $-\infty$.

```
void fmpz_poly_scalar_fdiv_2exp(fmpz_poly_t poly1,
   fmpz_poly_t poly2, ulong x)
```

Sets `poly1` to `poly2` divided by `2^x`, rounding coefficients down toward $-\infty$.

```
void fmpz_poly_scalar_tdiv_fmpz(fmpz_poly_t poly1, const
   fmpz_poly_t poly2, const fmpz_t x)
```

Sets `poly1` to `poly2` divided by the `fmpz_t x`, rounding coefficients toward 0.

```
void fmpz_poly_scalar_tdiv_si(fmpz_poly_t poly1,
   fmpz_poly_t poly2, slong x)
```

Sets `poly1` to `poly2` divided by the `slong x`, rounding coefficients toward 0.

```
void fmpz_poly_scalar_tdiv_ui(fmpz_poly_t poly1,
   fmpz_poly_t poly2, ulong x)
```

Sets `poly1` to `poly2` divided by the `ulong x`, rounding coefficients toward 0.

```
void fmpz_poly_scalar_tdiv_2exp(fmpz_poly_t poly1,
   fmpz_poly_t poly2, ulong x)
```

Sets `poly1` to `poly2` divided by `2^x`, rounding coefficients toward 0.

```
void fmpz_poly_scalar_divexact_fmpz(fmpz_poly_t poly1,
   const fmpz_poly_t poly2, const fmpz_t x)
```

Sets `poly1` to `poly2` divided by the `fmpz_t x`, assuming the division is exact for every coefficient.

```
void fmpz_poly_scalar_divexact_mpz(fmpz_poly_t poly1, const
   fmpz_poly_t poly2, const mpz_t x)
```

Sets `poly1` to `poly2` divided by the `mpz_t x`, assuming the coefficient is exact for every coefficient.

```
id fmpz_poly_scalar_divexact_si(fmpz_poly_t poly1,
   fmpz_poly_t poly2, slong x)
```

Sets `poly1` to `poly2` divided by the `slong x`, assuming the coefficient is exact for every coefficient.

```
void fmpz_poly_scalar_divexact_ui(fmpz_poly_t poly1,
   fmpz_poly_t poly2, ulong x)
```

Sets `poly1` to `poly2` divided by the `ulong x`, assuming the coefficient is exact for every coefficient.

```
void fmpz_poly_scalar_mod_fmpz(fmpz_poly_t poly1, const
   fmpz_poly_t poly2, const fmpz_t p)
```

Sets `poly1` to `poly2`, reducing each coefficient modulo $p > 0$.

```
void fmpz_poly_scalar_smod_fmpz(fmpz_poly_t poly1, const
   fmpz_poly_t poly2, const fmpz_t p)
```

Sets `poly1` to `poly2`, symmetrically reducing each coefficient modulo $p > 0$, that is, choosing the unique representative in the interval $(-p/2, p/2]$.

22.12 Bit packing

```
void _fmpz_poly_bit_pack(mp_ptr arr, const fmpz * poly,
    slong len, mp_bitcnt_t bit_size, int negate)
```

Packs the coefficients of `poly` into bitfields of the given `bit_size`, negating the coefficients before packing if `negate` is set to -1.

```
int _fmpz_poly_bit_unpack(fmpz * poly, slong len, mp_srcptr
    arr, mp_bitcnt_t bit_size, int negate)
```

Unpacks the polynomial of given length from the array as packed into fields of the given `bit_size`, finally negating the coefficients if `negate` is set to -1. Returns borrow, which is nonzero if a leading term with coefficient ± 1 should be added at position `len` of `poly`.

```
void _fmpz_poly_bit_unpack_unsigned(fmpz * poly, slong len,
    mp_srcptr_t arr, mp_bitcnt_t bit_size)
```

Unpacks the polynomial of given length from the array as packed into fields of the given `bit_size`. The coefficients are assumed to be unsigned.

```
void fmpz_poly_bit_pack(fmpz_t f, const fmpz_poly_t poly,
    mp_bitcnt_t bit_size)
```

Packs `poly` into bitfields of size `bit_size`, writing the result to `f`. The sign of `f` will be the same as that of the leading coefficient of `poly`.

```
void fmpz_poly_bit_unpack(fmpz_poly_t poly, const fmpz_t f,
    mp_bitcnt_t bit_size)
```

Unpacks the polynomial with signed coefficients packed into fields of size `bit_size` as represented by the integer `f`.

```
void fmpz_poly_bit_unpack_unsigned(fmpz_poly_t poly, const
    fmpz_t f, mp_bitcnt_t bit_size)
```

Unpacks the polynomial with unsigned coefficients packed into fields of size `bit_size` as represented by the integer `f`. It is required that `f` is nonnegative.

22.13 Multiplication

```
void _fmpz_poly_mul_classical(fmpz * res, const fmpz *
    poly1, slong len1, const fmpz * poly2, slong len2)
```

Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes `len1` and `len2` are positive. Allows zero-padding of the two input polynomials. No aliasing of inputs with outputs is allowed.

```
void fmpz_poly_mul_classical(fmpz_poly_t res, const
    fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`, computed using the classical or schoolbook method.

```
void _fmpz_poly_mullow_classical(fmpz * res, const fmpz *
    poly1, slong len1, const fmpz * poly2, slong len2, slong
    n)
```

Sets `(res, n)` to the first n coefficients of `(poly1, len1)` multiplied by `(poly2, len2)`.

Assumes `0 < n <= len1 + len2 - 1`. Assumes neither `len1` nor `len2` is zero.

```
void fmpz_poly_mullow_classical(fmpz_poly_t res, const
    fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)
```

Sets `res` to the first n coefficients of `poly1 * poly2`.

```
void _fmpz_poly_mulhigh_classical(fmpz * res, const fmpz *
   poly1, slong len1, const fmpz * poly2, slong len2, slong
   start)
```

Sets the first `start` coefficients of `res` to zero and the remainder to the corresponding coefficients of `(poly1, len1)* (poly2, len2)`.

Assumes `start <= len1 + len2 - 1`. Assumes neither `len1` nor `len2` is zero.

```
void fmpz_poly_mulhigh_classical(fmpz_poly_t res, const
   fmpz_poly_t poly1, const fmpz_poly_t poly2, slong start)
```

Sets the first `start` coefficients of `res` to zero and the remainder to the corresponding coefficients of the product of `poly1` and `poly2`.

```
void _fmpz_poly_mulmid_classical(fmpz * res, const fmpz *
   poly1, slong len1, const fmpz * poly2, slong len2)
```

Sets `res` to the middle `len1 - len2 + 1` coefficients of the product of `(poly1, len1)` and `(poly2, len2)`, i.e. the coefficients from degree `len2 - 1` to `len1 - 1` inclusive. Assumes that `len1 >= len2 > 0`.

```
void fmpz_poly_mulmid_classical(fmpz_poly_t res, const
   fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Sets `res` to the middle `len(poly1)- len(poly2)+ 1` coefficients of `poly1 * poly2`, i.e. the coefficient from degree `len2 - 1` to `len1 - 1` inclusive. Assumes that `len1 >= len2`.

```
void _fmpz_poly_mul_karatsuba(fmpz * res, const fmpz *
   poly1, slong len1, const fmpz * poly2, slong len2)
```

Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2)`. Assumes `len1 >= len2 > 0`. Allows zero-padding of the two input polynomials. No aliasing of inputs with outputs is allowed.

```
void fmpz_poly_mul_karatsuba(fmpz_poly_t res, const
   fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _fmpz_poly_mullow_karatsuba_n(fmpz * res, const fmpz *
   poly1, const fmpz * poly2, slong n)
```

Sets `res` to the product of `poly1` and `poly2` and truncates to the given length. It is assumed that `poly1` and `poly2` are precisely the given length, possibly zero padded. Assumes n is not zero.

```
void fmpz_poly_mullow_karatsuba_n(fmpz_poly_t res, const
   fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)
```

Sets `res` to the product of `poly1` and `poly2` and truncates to the given length.

```
void _fmpz_poly_mulhigh_karatsuba_n(fmpz * res, const fmpz
   * poly1, const fmpz * poly2, slong len)
```

Sets `res` to the product of `poly1` and `poly2` and truncates at the top to the given length. The first `len - 1` coefficients are set to zero. It is assumed that `poly1` and `poly2` are precisely the given length, possibly zero padded. Assumes `len` is not zero.

```
void fmpz_poly_mulhigh_karatsuba_n(fmpz_poly_t res, const
   fmpz_poly_t poly1, const fmpz_poly_t poly2, slong len)
```

Sets the first `len - 1` coefficients of the result to zero and the remaining coefficients to the corresponding coefficients of the product of `poly1` and `poly2`. Assumes `poly1` and `poly2` are at most of the given length.

```
void _fmpz_poly_mul_KS(fmpz * res, const fmpz * poly1,
   slong len1, const fmpz * poly2, slong len2)
```

Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2)`.

Places no assumptions on `len1` and `len2`. Allows zero-padding of the two input polynomials. Supports aliasing of inputs and outputs.

```
void fmpz_poly_mul_KS(fmpz_poly_t res, const fmpz_poly_t
   poly1, const fmpz_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _fmpz_poly_mullow_KS(fmpz * res, const fmpz * poly1,
   slong len1, const fmpz * poly2, slong len2, slong n)
```

Sets `(res, n)` to the lowest n coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `len1` and `len2` are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between `res`, `poly1` and `poly2`.

```
void fmpz_poly_mullow_KS(fmpz_poly_t res, const fmpz_poly_t
   poly1, const fmpz_poly_t poly2, slong n)
```

Sets `res` to the lowest n coefficients of the product of `poly1` and `poly2`.

```
void _fmpz_poly_mul_SS(fmpz * output, const fmpz * input1,
   slong length1, const fmpz * input2, slong length2)
```

Sets `(output, length1 + length2 - 1)` to the product of `(input1, length1)` and `(input2, length2)`.

We must have `len1 > 1` and `len2 > 1`. Allows zero-padding of the two input polynomials. Supports aliasing of inputs and outputs.

```
void fmpz_poly_mul_SS(fmpz_poly_t res, const fmpz_poly_t
   poly1, const fmpz_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`. Uses the Schönhage-Strassen algorithm.

```
void _fmpz_poly_mullow_SS(fmpz * output, const fmpz *
   input1, slong length1, const fmpz * input2, slong
   length2, slong n)
```

Sets `(res, n)` to the lowest n coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `len1` and `len2` are positive, but does allow for the polynomials to be zero-padded. We must have `len1 > 1` and `len2 > 1`. Assumes n is positive. Supports aliasing between `res`, `poly1` and `poly2`.

```
void fmpz_poly_mullow_SS(fmpz_poly_t res, const fmpz_poly_t
   poly1, const fmpz_poly_t poly2, slong n)
```

Sets `res` to the lowest n coefficients of the product of `poly1` and `poly2`.

```
void _fmpz_poly_mul(fmpz * res, const fmpz * poly1, slong
    len1, const fmpz * poly2, slong len2)
```

Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2)`. Assumes `len1 >= len2 > 0`. Allows zero-padding of the two input polynomials. Does not support aliasing between the inputs and the output.

```
void fmpz_poly_mul(fmpz_poly_t res, const fmpz_poly_t
    poly1, const fmpz_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`. Chooses an optimal algorithm from the choices above.

```
void _fmpz_poly_mullow(fmpz * res, const fmpz * poly1,
    slong len1, const fmpz * poly2, slong len2, slong n)
```

Sets `(res, n)` to the lowest n coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes `len1 >= len2 > 0` and `0 < n <= len1 + len2 - 1`. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

```
void fmpz_poly_mullow(fmpz_poly_t res, const fmpz_poly_t
    poly1, const fmpz_poly_t poly2, slong n)
```

Sets `res` to the lowest n coefficients of the product of `poly1` and `poly2`.

```
void fmpz_poly_mulhigh_n(fmpz_poly_t res, const fmpz_poly_t
    poly1, const fmpz_poly_t poly2, slong n)
```

Sets the high n coefficients of `res` to the high n coefficients of the product of `poly1` and `poly2`, assuming the latter are precisely n coefficients in length, zero padded if necessary. The remaining $n - 1$ coefficients may be arbitrary.

22.14 Squaring

```
void _fmpz_poly_sqr_KS(fmpz * rop, const fmpz * op, slong
    len)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `len > 0`.

Supports zero-padding in `(op, len)`. Does not support aliasing.

```
void fmpz_poly_sqr_KS(fmpz_poly_t rop, const fmpz_poly_t op)
```

Sets `rop` to the square of the polynomial `op` using Kronecker segmentation.

```
void _fmpz_poly_sqr_karatsuba(fmpz * rop, const fmpz * op,
    slong len)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `len > 0`.

Supports zero-padding in `(op, len)`. Does not support aliasing.

```
void fmpz_poly_sqr_karatsuba(fmpz_poly_t rop, const
    fmpz_poly_t op)
```

Sets `rop` to the square of the polynomial `op` using the Karatsuba multiplication algorithm.

```
void _fmpz_poly_sqr_classical(fmpz * rop, const fmpz * op,
    slong len)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `len > 0`.

Supports zero-padding in `(op, len)`. Does not support aliasing.

```
void fmpz_poly_sqr_classical(fmpz_poly_t rop, const
    fmpz_poly_t op)
```

Sets `rop` to the square of the polynomial `op` using the classical or schoolbook method.

```
void _fmpz_poly_sqr(fmpz * rop, const fmpz * op, slong len)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `len > 0`.

Supports zero-padding in `(op, len)`. Does not support aliasing.

```
void fmpz_poly_sqr(fmpz_poly_t rop, const fmpz_poly_t op)
```

Sets `rop` to the square of the polynomial `op`.

```
void _fmpz_poly_sqrlow_KS(fmpz * res, const fmpz * poly,
    slong len, slong n)
```

Sets `(res, n)` to the lowest n coefficients of the square of `(poly, len)`.

Assumes that `len` is positive, but does allow for the polynomial to be zero-padded. The polynomial may be zero, too. Assumes n is positive. Supports aliasing between `res` and `poly`.

```
void fmpz_poly_sqrlow_KS(fmpz_poly_t res, const fmpz_poly_t
    poly, slong n)
```

Sets `res` to the lowest n coefficients of the square of `poly`.

```
void _fmpz_poly_sqrlow_karatsuba_n(fmpz * res, const fmpz *
    poly, slong n)
```

Sets `(res, n)` to the square of `(poly, n)` truncated to length n, which is assumed to be positive. Allows for `poly` to be zero-oadded.

```
void fmpz_poly_sqrlow_karatsuba_n(fmpz_poly_t res, const
    fmpz_poly_t poly, slong n)
```

Sets `res` to the square of `poly` and truncates to the given length.

```
void _fmpz_poly_sqrlow_classical(fmpz * res, const fmpz *
    poly, slong len, slong n)
```

Sets `(res, n)` to the first n coefficients of the square of `(poly, len)`.

Assumes that `0 < n <= 2 * len - 1`.

```
void fmpz_poly_sqrlow_classical(fmpz_poly_t res, const
    fmpz_poly_t poly, slong n)
```

Sets `res` to the first n coefficients of the square of `poly`.

```
void _fmpz_poly_sqrlow(fmpz * res, const fmpz * poly, slong
    len, slong n)
```

Sets `(res, n)` to the lowest n coefficients of the square of `(poly, len)`.

Assumes `len1 >= len2 > 0` and `0 < n <= 2 * len - 1`. Allows for zero-padding in the input. Does not support aliasing between the input and the output.

```
void fmpz_poly_sqrlow(fmpz_poly_t res, const fmpz_poly_t
    poly, slong n)
```

Sets `res` to the lowest n coefficients of the square of `poly`.

22.15 Powering

```
void _fmpz_poly_pow_multinomial(fmpz * res, const fmpz *
    poly, slong len, ulong e)
```

Computes `res = poly^e`. This uses the J.C.P. Miller pure recurrence as follows:

If ℓ is the index of the lowest non-zero coefficient in `poly`, as a first step this method zeros out the lowest $e\ell$ coefficients of `res`. The recurrence above is then used to compute the remaining coefficients.

Assumes `len > 0`, `e > 0`. Does not support aliasing.

```
void fmpz_poly_pow_multinomial(fmpz_poly_t res, const
    fmpz_poly_t poly, ulong e)
```

Computes `res = poly^e` using a generalisation of binomial expansion called the J.C.P. Miller pure recurrence [20, 30]. If e is zero, returns one, so that in particular `0^0 = 1`.

The formal statement of the recurrence is as follows. Write the input polynomial as $P(x) = p_0 + p_1 x + \cdots + p_m x^m$ with $p_0 \neq 0$ and let

$$P(x)^n = a(n,0) + a(n,1)x + \cdots + a(n,mn)x^{mn}.$$

Then $a(n,0) = p_0^n$ and, for all $1 \leq k \leq mn$,

$$a(n,k) = (kp_0)^{-1} \sum_{i=1}^{m} p_i\big((n+1)i - k\big)a(n,k-i).$$

```
void _fmpz_poly_pow_binomial(fmpz * res, const fmpz * poly,
    ulong e)
```

Computes `res = poly^e` when poly is of length 2, using binomial expansion.

Assumes $e > 0$. Does not support aliasing.

```
void fmpz_poly_pow_binomial(fmpz_poly_t res, const
    fmpz_poly_t poly, ulong e)
```

Computes `res = poly^e` when `poly` is of length 2, using binomial expansion.

If the length of `poly` is not 2, raises an exception and aborts.

```
void _fmpz_poly_pow_addchains(fmpz * res, const fmpz *
    poly, slong len, const int * a, int n)
```

Given a star chain $1 = a_0 < a_1 < \cdots < a_n = e$ computes `res = poly^e`.

A star chain is an addition chain $1 = a_0 < a_1 < \cdots < a_n$ such that, for all $i > 0$, $a_i = a_{i-1} + a_j$ for some $j < i$.

Assumes that $e > 2$, or equivalently $n > 1$, and `len > 0`. Does not support aliasing.

```
void fmpz_poly_pow_addchains(fmpz_poly_t res, const
    fmpz_poly_t poly, ulong e)
```

Computes `res = poly^e` using addition chains whenever $0 \leq e \leq 148$.

If $e > 148$, raises an exception and aborts.

```
void _fmpz_poly_pow_binexp(fmpz * res, const fmpz * poly,
    slong len, ulong e)
```

Sets `res = poly^e` using left-to-right binary exponentiation as described in [20, p. 461].

Assumes that `len > 0`, `e > 1`. Assumes that `res` is an array of length at least `e*(len - 1)+ 1`. Does not support aliasing.

```
void fmpz_poly_pow_binexp(fmpz_poly_t res, const
    fmpz_poly_t poly, ulong e)
```

Computes `res = poly^e` using the binary exponentiation algorithm. If e is zero, returns one, so that in particular `0^0 = 1`.

```
void _fmpz_poly_pow_small(fmpz * res, const fmpz * poly,
    slong len, ulong e)
```

Sets `res = poly^e` whenever $0 \leq e \leq 4$.

Assumes that `len > 0` and that `res` is an array of length at least `e*(len - 1)+ 1`. Does not support aliasing.

```
void _fmpz_poly_pow(fmpz * res, const fmpz * poly, slong
    len, ulong e)
```

Sets `res = poly^e`, assuming that `e, len > 0` and that `res` has space for `e*(len - 1)+ 1` coefficients. Does not support aliasing.

```
void fmpz_poly_pow(fmpz_poly_t res, const fmpz_poly_t poly,
    ulong e)
```

Computes `res = poly^e`. If e is zero, returns one, so that in particular `0^0 = 1`.

```
void _fmpz_poly_pow_trunc(fmpz * res, const fmpz * poly,
    ulong e, slong n)
```

Sets `(res, n)` to `(poly, n)` raised to the power e and truncated to length n.

Assumes that $e, n > 0$. Allows zero-padding of `(poly, n)`. Does not support aliasing of any inputs and outputs.

```
void fmpz_poly_pow_trunc(fmpz_poly_t res, const fmpz_poly_t
    poly, ulong e, slong n)
```

Notationally raises `poly` to the power e, truncates the result to length n and writes the result in `res`. This is computed much more efficiently than simply powering the polynomial and truncating.

Thus, if $n = 0$ the result is zero. Otherwise, whenever $e = 0$ the result will be the constant polynomial equal to 1.

This function can be used to raise power series to a power in an efficient way.

22.16 Shifting

```
void _fmpz_poly_shift_left(fmpz * res, const fmpz * poly,
    slong len, slong n)
```

Sets `(res, len + n)` to `(poly, len)` shifted left by n coefficients.

Inserts zero coefficients at the lower end. Assumes that `len` and n are positive, and that `res` fits `len + n` elements. Supports aliasing between `res` and `poly`.

```
void fmpz_poly_shift_left(fmpz_poly_t res, const
    fmpz_poly_t poly, slong n)
```

Sets `res` to `poly` shifted left by n coeffs. Zero coefficients are inserted.

```
void _fmpz_poly_shift_right(fmpz * res, const fmpz * poly,
    slong len, slong n)
```

Sets `(res, len - n)` to `(poly, len)` shifted right by n coefficients.

Assumes that `len` and n are positive, that `len > n`, and that `res` fits `len - n` elements. Supports aliasing between `res` and `poly`, although in this case the top coefficients of `poly` are not set to zero.

```
void fmpz_poly_shift_right(fmpz_poly_t res, const
    fmpz_poly_t poly, slong n)
```

Sets `res` to `poly` shifted right by n coefficients. If n is equal to or greater than the current length of `poly`, `res` is set to the zero polynomial.

22.17 Bit sizes and norms

```
ulong fmpz_poly_max_limbs(const fmpz_poly_t poly)
```

Returns the maximum number of limbs required to store the absolute value of coefficients of `poly`. If `poly` is zero, returns 0.

```
slong fmpz_poly_max_bits(const fmpz_poly_t poly)
```

Computes the maximum number of bits b required to store the absolute value of coefficients of `poly`. If all the coefficients of `poly` are non-negative, b is returned, otherwise $-b$ is returned.

```
void fmpz_poly_height(fmpz_t height, const fmpz_poly_t poly)
```

Computes the height of `poly`, defined as the largest of the absolute values the coefficients of `poly`. Equivalently, this gives the infinity norm of the coefficients. If `poly` is zero, the height is 0.

```
void _fmpz_poly_2norm(fmpz_t res, const fmpz * poly, slong
    len)
```

Sets `res` to the Euclidean norm of `(poly, len)`, that is, the integer square root of the sum of the squares of the coefficients of `poly`.

```
void fmpz_poly_2norm(fmpz_t res, const fmpz_poly_t poly)
```

Sets `res` to the Euclidean norm of `poly`, that is, the integer square root of the sum of the squares of the coefficients of `poly`.

```
mp_limb_t _fmpz_poly_2norm_normalised_bits(const fmpz *
    poly, slong len)
```

Returns an upper bound on the number of bits of the normalised Euclidean norm of `(poly, len)`, i.e. the number of bits of the Euclidean norm divided by the absolute value of the leading coefficient. The returned value will be no more than 1 bit too large.

This is used in the computation of the Landau-Mignotte bound.

It is assumed that `len > 0`. The result only makes sense if the leading coefficient is nonzero.

22.18 Greatest common divisor

```
void _fmpz_poly_gcd_subresultant(fmpz * res, const fmpz *
    poly1, slong len1, const fmpz * poly2, slong len2)
```

Computes the greatest common divisor `(res, len2)` of `(poly1, len1)` and `(poly2, len2)`, assuming `len1 >= len2 > 0`. The result is normalised to have positive leading coefficient. Aliasing between `res`, `poly1` and `poly2` is supported.

```
void fmpz_poly_gcd_subresultant(fmpz_poly_t res, const
    fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Computes the greatest common divisor `res` of `poly1` and `poly2`, normalised to have non-negative leading coefficient.

This function uses the subresultant algorithm as described in [8, Algorithm 3.3.1].

```
int _fmpz_poly_gcd_heuristic(fmpz * res, const fmpz *
    poly1, slong len1, const fmpz * poly2, slong len2)
```

Computes the greatest common divisor `(res, len2)` of `(poly1, len1)` and `(poly2, len2)`, assuming `len1 >= len2 > 0`. The result is normalised to have positive leading coefficient. Aliasing between `res`, `poly1` and `poly2` is not supported. The function may not always succeed in finding the GCD. If it fails, the function returns 0, otherwise it returns 1.

```
int fmpz_poly_gcd_heuristic(fmpz_poly_t res, const
    fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Computes the greatest common divisor `res` of `poly1` and `poly2`, normalised to have non-negative leading coefficient.

The function may not always succeed in finding the GCD. If it fails, the function returns 0, otherwise it returns 1.

This function uses the heuristic GCD algorithm (GCDHEU). The basic strategy is to remove the content of the polynomials, pack them using Kronecker segmentation (given a bound on the size of the coefficients of the GCD) and take the integer GCD. Unpack the result and test divisibility.

```
void _fmpz_poly_gcd_modular(fmpz * res, const fmpz * poly1,
    slong len1, const fmpz * poly2, slong len2)
```

Computes the greatest common divisor `(res, len2)` of `(poly1, len1)` and `(poly2, len2)`, assuming `len1 >= len2 > 0`. The result is normalised to have positive leading coefficient. Aliasing between `res`, `poly1` and `poly2` is not supported.

```
void fmpz_poly_gcd_modular(fmpz_poly_t res, const
    fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Computes the greatest common divisor `res` of `poly1` and `poly2`, normalised to have non-negative leading coefficient.

This function uses the modular GCD algorithm. The basic strategy is to remove the content of the polynomials, reduce them modulo sufficiently many primes and do CRT reconstruction until some bound is reached (or we can prove with trial division that we have the GCD).

```
void _fmpz_poly_gcd(fmpz * res, const fmpz * poly1, slong
    len1, const fmpz * poly2, slong len2)
```

Computes the greatest common divisor `res` of `(poly1, len1)` and `(poly2, len2)`, assuming `len1 >= len2 > 0`. The result is normalised to have positive leading coefficient.

Assumes that `res` has space for `len2` coefficients. Aliasing between `res`, `poly1` and `poly2` is not supported.

```
void fmpz_poly_gcd(fmpz_poly_t res, const fmpz_poly_t
   poly1, const fmpz_poly_t poly2)
```

Computes the greatest common divisor `res` of `poly1` and `poly2`, normalised to have non-negative leading coefficient.

```
void _fmpz_poly_xgcd_modular(fmpz_t r, fmpz * s, fmpz * t,
   const fmpz * f, slong len1, const fmpz * g, slong len2)
```

Set r to the resultant of `(f, len1)` and `(g, len2)`. If the resultant is zero, the function returns immediately. Otherwise it finds polynomials s and t such that `s*f + t*g = r`. The length of s will be no greater than `len2` and the length of t will be no greater than `len1` (both are zero padded if necessary).

It is assumed that `len1 >= len2 > 0`. No aliasing of inputs and outputs is permitted.

The function assumes that f and g are primitive (have Gaussian content equal to 1). The result is undefined otherwise.

Uses a multimodular algorithm. The resultant is first computed and extended GCD's modulo various primes p are computed and combined using CRT. When the CRT stabilises the resulting polynomials are simply reduced modulo further primes until a proven bound is reached.

```
void fmpz_poly_xgcd_modular(fmpz_t r, fmpz_poly_t s,
   fmpz_poly_t t, const fmpz_poly_t f, const fmpz_poly_t g)
```

Set r to the resultant of f and g. If the resultant is zero, the function then returns immediately, otherwise s and t are found such that `s*f + t*g = r`.

The function assumes that f and g are primitive (have Gaussian content equal to 1). The result is undefined otherwise.

Uses the multimodular algorithm.

```
void _fmpz_poly_xgcd(fmpz_t r, fmpz * s, fmpz * t, const
   fmpz * f, slong len1, const fmpz * g, slong len2)
```

Set r to the resultant of `(f, len1)` and `(g, len2)`. If the resultant is zero, the function returns immediately. Otherwise it finds polynomials s and t such that `s*f + t*g = r`. The length of s will be no greater than `len2` and the length of t will be no greater than `len1` (both are zero padded if necessary).

The function assumes that f and g are primitive (have Gaussian content equal to 1). The result is undefined otherwise.

It is assumed that `len1 >= len2 > 0`. No aliasing of inputs and outputs is permitted.

```
void fmpz_poly_xgcd(fmpz_t r, fmpz_poly_t s, fmpz_poly_t t,
   const fmpz_poly_t f, const fmpz_poly_t g)
```

Set r to the resultant of f and g. If the resultant is zero, the function then returns immediately, otherwise s and t are found such that `s*f + t*g = r`.

The function assumes that f and g are primitive (have Gaussian content equal to 1). The result is undefined otherwise.

```
void _fmpz_poly_lcm(fmpz * res, const fmpz * poly1, slong
   len1, const fmpz * poly2, slong len2)
```

Sets `(res, len1 + len2 - 1)` to the least common multiple of the two polynomials `(poly1, len1)` and `(poly2, len2)`, normalised to have non-negative leading coefficient.

Assumes that `len1 >= len2 > 0`.

Does not support aliasing.

```
void fmpz_poly_lcm(fmpz_poly_t res, const fmpz_poly_t
   poly1, const fmpz_poly_t poly2)
```

Sets `res` to the least common multiple of the two polynomials `poly1` and `poly2`, normalised to have non-negative leading coefficient.

If either of the two polynomials is zero, sets `res` to zero.

This ensures that the equality

$$fg = \gcd(f, g) \operatorname{lcm}(f, g)$$

holds up to sign.

```
void _fmpz_poly_resultant(fmpz_t res, const fmpz * poly1,
   slong len1, const fmpz * poly2, slong len2)
```

Sets `res` to the resultant of `(poly1, len1)` and `(poly2, len2)`, assuming that `len1 >= len2 > 0`.

```
void fmpz_poly_resultant(fmpz_t res, const fmpz_poly_t
   poly1, const fmpz_poly_t poly2)
```

Computes the resultant of `poly1` and `poly2`.

For two non-zero polynomials $f(x) = a_m x^m + \cdots + a_0$ and $g(x) = b_n x^n + \cdots + b_0$ of degrees m and n, the resultant is defined to be

$$a_m^n b_n^m \prod_{(x,y):f(x)=g(y)=0} (x - y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

This function uses the algorithm described in [8, Algorithm 3.3.7].

22.19 Gaussian content

```
void _fmpz_poly_content(fmpz_t res, const fmpz * poly,
   slong len)
```

Sets `res` to the non-negative content of `(poly, len)`. Aliasing between `res` and the coefficients of `poly` is not supported.

```
void fmpz_poly_content(fmpz_t res, const fmpz_poly_t poly)
```

Sets `res` to the non-negative content of `poly`. The content of the zero polynomial is defined to be zero. Supports aliasing, that is, `res` is allowed to be one of the coefficients of `poly`.

```
void _fmpz_poly_primitive_part(fmpz * res, const fmpz *
   poly, slong len)
```

Sets `(res, len)` to `(poly, len)` divided by the content of `(poly, len)`, and normalises the result to have non-negative leading coefficient.

Assumes that `(poly, len)` is non-zero. Supports aliasing of `res` and `poly`.

```
void fmpz_poly_primitive_part(fmpz_poly_t res, const
   fmpz_poly_t poly)
```

Sets `res` to `poly` divided by the content of `poly`, and normalises the result to have non-negative leading coefficient. If `poly` is zero, sets `res` to zero.

22.20 Square-free

```
int _fmpz_poly_is_squarefree(const fmpz * poly, slong len)
```

Returns whether the polynomial `(poly, len)` is square-free.

```
int fmpz_poly_is_squarefree(const fmpz_poly_t poly)
```

Returns whether the polynomial `poly` is square-free. A non-zero polynomial is defined to be square-free if it has no non-unit square factors. We also define the zero polynomial to be square-free.

Returns 1 if the length of `poly` is at most 2. Returns whether the discriminant is zero for quadratic polynomials. Otherwise, returns whether the greatest common divisor of `poly` and its derivative has length 1.

22.21 Euclidean division

```
void _fmpz_poly_divrem_basecase(fmpz * Q, fmpz * R, const
   fmpz * A, slong lenA, const fmpz * B, slong lenB)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ + R$ and each coefficient of R beyond `lenB` is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same thing as division over $\mathbf{Q}$.

Assumes that $\operatorname{len}(A), \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fmpz_poly_divrem_basecase(fmpz_poly_t Q, fmpz_poly_t
   R, const fmpz_poly_t A, const fmpz_poly_t B)
```

Computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same thing as division over $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_divrem_divconquer_recursive(fmpz * Q, fmpz
   * BQ, fmpz * W, const fmpz * A, const fmpz * B, slong
   lenB)
```

Computes `(Q, lenB)`, `(BQ, 2 lenB - 1)` such that $BQ = B \times Q$ and $A = BQ + R$ where each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. We assume that $\operatorname{len}(A) = 2\operatorname{len}(B) - 1$. If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$.

Assumes $\operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. Requires a temporary array `(W, 2 lenB - 1)`. No aliasing of input and output operands is allowed.

This function does not read the bottom $\operatorname{len}(B) - 1$ coefficients from A, which means that they might not even need to exist in allocated memory.

```
void _fmpz_poly_divrem_divconquer(fmpz * Q, fmpz * R, const
   fmpz * A, slong lenA, const fmpz * B, slong lenB)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$.

Assumes $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. No aliasing of input and output operands is allowed.

```
void fmpz_poly_divrem_divconquer(fmpz_poly_t Q, fmpz_poly_t
   R, const fmpz_poly_t A, const fmpz_poly_t B)
```

Computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_divrem(fmpz * Q, fmpz * R, const fmpz * A,
   slong lenA, const fmpz * B, slong lenB)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same thing as division over $\mathbf{Q}$.

Assumes $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. No aliasing of input and output operands is allowed.

```
void fmpz_poly_divrem(fmpz_poly_t Q, fmpz_poly_t R, const
   fmpz_poly_t A, const fmpz_poly_t B)
```

Computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_div_basecase(fmpz * Q, fmpz * R, const fmpz
   * A, slong lenA, const fmpz * B, slong lenB)
```

Computes the quotient `(Q, lenA - lenB + 1)` of `(A, lenA)` divided by `(B, lenB)`.

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B.

If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$.

Assumes $\operatorname{len}(A), \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. Requires a temporary array R of size at least the (actual) length of A. For convenience, R may be `NULL`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fmpz_poly_div_basecase(fmpz_poly_t Q, const
   fmpz_poly_t A, const fmpz_poly_t B)
```

Computes the quotient Q of A divided by Q.

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B.

If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_divremlow_divconquer_recursive(fmpz * Q,
   fmpz * BQ, const fmpz * A, const fmpz * B, slong lenB)
```

Divide and conquer division of `(A, 2 lenB - 1)` by `(B, lenB)`, computing only the bottom $\operatorname{len}(B) - 1$ coefficients of BQ.

Assumes $\operatorname{len}(B) > 0$. Requires BQ to have length at least $2\operatorname{len}(B) - 1$, although only the bottom $\operatorname{len}(B) - 1$ coefficients will carry meaningful output. Does not support any aliasing. Allows zero-padding in A, but not in B.

```
void _fmpz_poly_div_divconquer_recursive(fmpz * Q, fmpz *
    temp, const fmpz * A, const fmpz * B, slong lenB)
```

Recursive short division in the balanced case.

Computes the quotient `(Q, lenB)` of `(A, 2 lenB - 1)` upon division by `(B, lenB)`. Requires $\operatorname{len}(B) > 0$. Needs a temporary array `temp` of length $2\operatorname{len}(B) - 1$. Does not support any aliasing.

For further details, see [23].

```
void _fmpz_poly_div_divconquer(fmpz * Q, const fmpz * A,
    slong lenA, const fmpz * B, slong lenB)
```

Computes the quotient `(Q, lenA - lenB + 1)` of `(A, lenA)` upon division by `(B, lenB)`. Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Does not support aliasing.

```
fmpz_poly_div_divconquer(fmpz_poly_t Q, const fmpz_poly_t
    A, const fmpz_poly_t B)
```

Computes the quotient Q of A divided by B.

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B.

If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_div(fmpz * Q, const fmpz * A, slong lenA,
    const fmpz * B, slong lenB)
```

Computes the quotient `(Q, lenA - lenB + 1)` of `(A, lenA)` divided by `(B, lenB)`.

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$.

Assumes $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. Aliasing of input and output operands is not allowed.

```
void fmpz_poly_div(fmpz_poly_t Q, const fmpz_poly_t A,
    const fmpz_poly_t B)
```

Computes the quotient Q of A divided by B.

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over Q. An exception is raised if B is zero.

```
void _fmpz_poly_rem_basecase(fmpz * R, const fmpz * A,
    slong lenA, const fmpz * B, slong lenB)
```

Computes the remainder `(R, lenA)` of `(A, lenA)` upon division by `(B, lenB)`.

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same thing as division over $\mathbf{Q}$.

Assumes that $\operatorname{len}(A), \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fmpz_poly_rem_basecase(fmpz_poly_t R, const
    fmpz_poly_t A, const fmpz_poly_t B)
```

Computes the remainder R of A upon division by B.

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_rem(fmpz * R, const fmpz * A, slong lenA,
   const fmpz * B, slong lenB)
```

Computes the remainder `(R, lenA)` of `(A, lenA)` upon division by `(B, lenB)`.

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same thing as division over $\mathbf{Q}$.

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. Aliasing of input and output operands is not allowed.

```
void fmpz_poly_rem(fmpz_poly_t R, const fmpz_poly_t A,
   const fmpz_poly_t B)
```

Computes the remainder R of A upon division by B.

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\operatorname{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbf{Q}$. An exception is raised if B is zero.

```
void _fmpz_poly_div_root(fmpz * Q, const fmpz * A, slong
   len, const fmpz_t c)
```

Computes the quotient `(Q, len-1)` of `(A, len)` upon division by $x - c$.

Supports aliasing of `Q` and `A`, but the result is undefined in case of partial overlap.

```
void fmpz_poly_div_root(fmpz_poly_t Q, const fmpz_poly_t A,
   const fmpz_t c)
```

Computes the quotient `(Q, len-1)` of `(A, len)` upon division by $x - c$.

22.22 Division with precomputed inverse

```
void _fmpz_poly_preinvert(fmpz * B_inv, const fmpz * B,
   slong n)
```

Given a monic polynomial `B` of length `n`, compute a precomputed inverse `B_inv` of length `n` for use in the functions below. No aliasing of B and B_inv is permitted. We assume `n` is not zero.

```
void fmpz_poly_preinvert(fmpz_poly_t B_inv, const
   fmpz_poly_t B)
```

Given a monic polynomial `B`, compute a precomputed inverse `B_inv` for use in the functions below. An exception is raised if `B` is zero.

```
void _fmpz_poly_div_preinv(fmpz * Q, const fmpz * A, slong
   len1, const fmpz * B, const fmpz * B_inv, slong len2)
```

Given a precomputed inverse `B_inv` of the polynomial `B` of length `len2`, compute the quotient `Q` of `A` by `B`. We assume the length `len1` of `A` is at least `len2`. The polynomial `Q` must have space for `len1 - len2 + 1` coefficients. No aliasing of operands is permitted.

```
void fmpz_poly_div_preinv(fmpz_poly_t Q, const fmpz_poly_t
   A, const fmpz_poly_t B, const fmpz_poly_t B_inv)
```

Given a precomputed inverse `B_inv` of the polynomial `B`, compute the quotient `Q` of `A` by `B`. Aliasing of `B` and `B_inv` is not permitted.

```
void _fmpz_poly_divrem_preinv(fmpz * Q, fmpz * A, slong
   len1, const fmpz * B, const fmpz * B_inv, slong len2)
```

Given a precomputed inverse `B_inv` of the polynomial `B` of length `len2`, compute the quotient `Q` of `A` by `B`. The remainder is then placed in `A`. We assume the length `len1` of `A` is at least `len2`. The polynomial `Q` must have space for `len1 - len2 + 1` coefficients. No aliasing of operands is permitted.

```
void fmpz_poly_divrem_preinv(fmpz_poly_t Q, fmpz_poly_t R,
   const fmpz_poly_t A, const fmpz_poly_t B, const
   fmpz_poly_t B_inv)
```

Given a precomputed inverse `B_inv` of the polynomial `B`, compute the quotient `Q` of `A` by `B` and the remainder `R`. Aliasing of `B` and `B_inv` is not permitted.

```
fmpz ** _fmpz_poly_powers_precompute(const fmpz * B, slong
   len)
```

Computes `2*len - 1` powers of x modulo the polynomial B of the given length. This is used as a kind of precomputed inverse in the remainder routine below.

```
void fmpz_poly_powers_precompute(fmpz_poly_powers_precomp_t
   pinv, fmpz_poly_t poly)
```

Computes `2*len - 1` powers of x modulo the polynomial B of the given length. This is used as a kind of precomputed inverse in the remainder routine below.

```
void _fmpz_poly_powers_clear(fmpz ** powers, slong len)
```

Clean up resources used by precomputed powers which have been computed by `_fmpz_poly_powers_precompute`.

```
void fmpz_poly_powers_clear(fmpz_poly_powers_precomp_t pinv)
```

Clean up resources used by precomputed powers which have been computed by `fmpz_poly_powers_precompute`.

```
void _fmpz_poly_rem_powers_precomp(fmpz * A, slong m, const
   fmpz * B, slong n, fmpz ** const powers)
```

Set A to the remainder of A divide B given precomputed powers mod B provided by `_fmpz_poly_powers_precompute`. No aliasing is allowed.

```
void fmpz_poly_rem_powers_precomp(fmpz_poly_t R, const
   fmpz_poly_t A, const fmpz_poly_t B, const
   fmpz_poly_powers_precomp_t B_inv)
```

Set R to the remainder of A divide B given precomputed powers mod B provided by `fmpz_poly_powers_precompute`.

22.23 Divisibility testing

```
int _fmpz_poly_divides(fmpz * Q, const fmpz * A, slong
   lenA, const fmpz * B, slong lenB)
```

Returns 1 if `(B, lenB)` divides `(A, lenA)` exactly and sets Q to the quotient, otherwise returns 0.

It is assumed that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$ and that Q has space for $\operatorname{len}(A) - \operatorname{len}(B) + 1$ coefficients.

Aliasing of Q with either of the inputs is not permitted.

This function is currently unoptimised and provided for convenience only.

```
int fmpz_poly_divides(fmpz_poly_t Q, const fmpz_poly_t A,
    const fmpz_poly_t B)
```

Returns 1 if B divides A exactly and sets Q to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.

22.24 Power series division

```
void _fmpz_poly_inv_series_newton(fmpz * Qinv, const fmpz *
    Q, slong n)
```

Computes the first n terms of the inverse power series of Q using Newton iteration.

Assumes that $n \geq 1$, that Q has length at least n and constant term ± 1. Does not support aliasing.

```
id fmpz_poly_inv_series_newton(fmpz_poly_t Qinv, const
    fmpz_poly_t Q, slong n)
```

Computes the first n terms of the inverse power series of Q using Newton iteration, assuming that Q has constant term ± 1 and $n \geq 1$.

```
void _fmpz_poly_inv_series(fmpz * Qinv, const fmpz * Q,
    slong n)
```

Computes the first n terms of the inverse power series of Q.

Assumes that $n \geq 1$, that Q has length at least n and constant term 1. Does not support aliasing.

```
void fmpz_poly_inv_series(fmpz_poly_t Qinv, const
    fmpz_poly_t Q, slong n)
```

Computes the first n terms of the inverse power series of Q, assuming Q has constant term 1 and $n \geq 1$.

```
void _fmpz_poly_div_series(fmpz * Q, const fmpz * A, const
    fmpz * B)
```

Divides `(A, n)` by `(B, n)` as power series over $\mathbf{Z}$, assuming B has constant term 1 and $n \geq 1$.

Only supports aliasing of `(Q, n)` and `(B, n)`.

```
void fmpz_poly_div_series(fmpz_poly_t Q, const fmpz_poly_t
    A, const fmpz_poly_t B, slong n)
```

Performs power series division in $\mathbf{Z}[[x]]/(x^n)$. The function considers the polynomials A and B as power series of length n starting with the constant terms. The function assumes that B has constant term 1 and $n \geq 1$.

22.25 Pseudo division

```
void _fmpz_poly_pseudo_divrem_basecase(fmpz * Q, fmpz * R,
   ulong * d, const fmpz * A, slong lenA, const fmpz * B,
   slong lenB, const fmpz_preinvn_t inv)
```

If ℓ is the leading coefficient of B, then computes Q, R such that $\ell^d A = QB + R$. This function is used for simulating division over $\mathbf{Q}$.

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Assumes that Q can fit $\operatorname{len}(A) - \operatorname{len}(B) + 1$ coefficients, and that R can fit $\operatorname{len}(A)$ coefficients. Supports aliasing of `(R, lenA)` and `(A, lenA)`. But other than this, no aliasing of the inputs and outputs is suppported.

An optional precomputed inverse of the leading coefficient of B from `fmpz_preinvn_init` can be supplied. Otherwise `inv` should be `NULL`.

```
void fmpz_poly_pseudo_divrem_basecase(fmpz_poly_t Q,
   fmpz_poly_t R, ulong * d, const fmpz_poly_t A, const
   fmpz_poly_t B)
```

If ℓ is the leading coefficient of B, then computes Q, R such that $\ell^d A = QB + R$. This function is used for simulating division over $\mathbf{Q}$.

```
void _fmpz_poly_pseudo_divrem_divconquer(fmpz * Q, fmpz *
   R, ulong * d, const fmpz * A, slong lenB, const fmpz *
   B, slong lenB, const fmpz_preinvn_t inv)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $\ell^d A = BQ + R$, only setting the bottom $\operatorname{len}(B) - 1$ coefficients of R to their correct values. The remaining top coefficients of `(R, lenA)` may be arbitrary.

Assumes $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. No aliasing of input and output operands is allowed.

An optional precomputed inverse of the leading coefficient of B from `fmpz_preinvn_init` can be supplied. Otherwise `inv` should be `NULL`.

```
void fmpz_poly_pseudo_divrem_divconquer(fmpz_poly_t Q,
   fmpz_poly_t R, ulong * d, const fmpz_poly_t A, const
   fmpz_poly_t B)
```

Computes Q, R, and d such that $\ell^d A = BQ + R$, where R has length less than the length of B and ℓ is the leading coefficient of B. An exception is raised if B is zero.

```
void _fmpz_poly_pseudo_divrem_cohen(fmpz * Q, fmpz * R,
   const fmpz * A, slong lenA, const fmpz * B, slong lenB)
```

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Assumes that Q can fit $\operatorname{len}(A) - \operatorname{len}(B) + 1$ coefficients, and that R can fit $\operatorname{len}(A)$ coefficients. Supports aliasing of `(R, lenA)` and `(A, lenA)`. But other than this, no aliasing of the inputs and outputs is supported.

```
void fmpz_poly_pseudo_divrem_cohen(fmpz_poly_t Q,
   fmpz_poly_t R, const fmpz_poly_t A, const fmpz_poly_t B)
```

This is a variant of `fmpz_poly_pseudo_divrem` which computes polynomials Q and R such that $\ell^d A = BQ + R$. However, the value of d is fixed at $\max\{0, \operatorname{len}(A) - \operatorname{len}(B) + 1\}$.

This function is faster when the remainder is not well behaved, i.e. where it is not expected to be close to zero. Note that this function is not asymptotically fast. It is efficient only for short polynomials, e.g. when $\operatorname{len}(B) < 32$.

```
void _fmpz_poly_pseudo_rem_cohen(fmpz * R, const fmpz * A,
   slong lenA, const fmpz * B, slong lenB)
```

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Assumes that R can fit $\operatorname{len}(A)$ coefficients. Supports aliasing of `(R, lenA)` and `(A, lenA)`. But other than this, no aliasing of the inputs and outputs is supported.

```
void fmpz_poly_pseudo_rem_cohen(fmpz_poly_t R, const
    fmpz_poly_t A, const fmpz_poly_t B)
```

This is a variant of `fmpz_poly_pseudo_rem()` which computes polynomials Q and R such that $\ell^d A = BQ + R$, but only returns R. However, the value of d is fixed at $\max\{0, \operatorname{len}(A) - \operatorname{len}(B) + 1\}$.

This function is faster when the remainder is not well behaved, i.e. where it is not expected to be close to zero. Note that this function is not asymptotically fast. It is efficient only for short polynomials, e.g. when $\operatorname{len}(B) < 32$.

This function uses the algorithm described in [8, Algorithm 3.1.2].

```
void _fmpz_poly_pseudo_divrem(fmpz * Q, fmpz * R, ulong *
    d, const fmpz * A, slong lenA, const fmpz * B, slong
    lenB, const fmpz_preinvn_t inv)
```

If ℓ is the leading coefficient of B, then computes `(Q, lenA - lenB + 1)`, `(R, lenB - 1)` and d such that $\ell^d A = BQ + R$. This function is used for simulating division over $\mathbf{Q}$.

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Assumes that Q can fit $\operatorname{len}(A) - \operatorname{len}(B) + 1$ coefficients, and that R can fit $\operatorname{len}(A)$ coefficients, although on exit only the bottom $\operatorname{len}(B)$ coefficients will carry meaningful data.

Supports aliasing of `(R, lenA)` and `(A, lenA)`. But other than this, no aliasing of the inputs and outputs is suppported.

An optional precomputed inverse of the leading coefficient of B from `fmpz_preinvn_init` can be supplied. Otherwise `inv` should be `NULL`.

```
void fmpz_poly_pseudo_divrem(fmpz_poly_t Q, fmpz_poly_t R,
    ulong * d, const fmpz_poly_t A, const fmpz_poly_t B)
```

Computes Q, R, and d such that $\ell^d A = BQ + R$.

```
void _fmpz_poly_pseudo_div(fmpz * Q, ulong * d, const fmpz
    * A, slong lenA, const fmpz * B, slong lenB, const
    fmpz_preinvn_t inv)
```

Pseudo-division, only returning the quotient.

```
void fmpz_poly_pseudo_div(fmpz_poly_t Q, ulong * d, const
    fmpz_poly_t A, const fmpz_poly_t B)
```

Pseudo-division, only returning the quotient.

```
void _fmpz_poly_pseudo_rem(fmpz * R, ulong * d, const fmpz
    * A, slong lenA, const fmpz * B, slong lenB, const
    fmpz_preinvn_t inv)
```

Pseudo-division, only returning the remainder.

```
void fmpz_poly_pseudo_rem(fmpz_poly_t R, ulong * d, const
    fmpz_poly_t A, const fmpz_poly_t B)
```

Pseudo-division, only returning the remainder.

22.26 Derivative

```
void _fmpz_poly_derivative(fmpz * rpoly, const fmpz * poly,
   slong len)
```

Sets `(rpoly, len - 1)` to the derivative of `(poly, len)`. Also handles the cases where `len` is 0 or 1 correctly. Supports aliasing of `rpoly` and `poly`.

```
void fmpz_poly_derivative(fmpz_poly_t res, const
   fmpz_poly_t poly)
```

Sets `res` to the derivative of `poly`.

22.27 Evaluation

```
void _fmpz_poly_evaluate_divconquer_fmpz(fmpz_t res, const
   fmpz * poly, slong len, const fmpz_t a)
```

Evaluates the polynomial `(poly, len)` at the integer a using a divide and conquer approach. Assumes that the length of the polynomial is at least one. Allows zero padding. Does not allow aliasing between `res` and `x`.

```
void fmpz_poly_evaluate_divconquer_fmpz(fmpz_t res, const
   fmpz_poly_t poly, const fmpz_t a)
```

Evaluates the polynomial `poly` at the integer a using a divide and conquer approach.

Aliasing between `res` and `a` is supported, however, `res` may not be part of `poly`.

```
void _fmpz_poly_evaluate_horner_fmpz(fmpz_t res, const fmpz
   * f, slong len, const fmpz_t a)
```

Evaluates the polynomial `(f, len)` at the integer a using Horner's rule, and sets `res` to the result. Aliasing between `res` and a or any of the coefficients of f is not supported.

```
void fmpz_poly_evaluate_horner_fmpz(fmpz_t res, const
   fmpz_poly_t f, const fmpz_t a)
```

Evaluates the polynomial f at the integer a using Horner's rule, and sets `res` to the result.

As expected, aliasing between `res` and `a` is supported. However, `res` may not be aliased with a coefficient of f.

```
void _fmpz_poly_evaluate_fmpz(fmpz_t res, const fmpz * f,
   slong len, const fmpz_t a)
```

Evaluates the polynomial `(f, len)` at the integer a and sets `res` to the result. Aliasing between `res` and a or any of the coefficients of f is not supported.

```
void fmpz_poly_evaluate_fmpz(fmpz_t res, const fmpz_poly_t
   f, const fmpz_t a)
```

Evaluates the polynomial f at the integer a and sets `res` to the result.

As expected, aliasing between `res` and a is supported. However, `res` may not be aliased with a coefficient of f.

```
void _fmpz_poly_evaluate_horner_mpq(fmpz_t rnum, fmpz_t
   rden, const fmpz * f, slong len, const fmpz_t anum,
   const fmpz_t aden)
```

Evaluates the polynomial `(f, len)` at the rational `(anum, aden)` using Horner's rule, and sets `(rnum, rden)` to the result in lowest terms.

Aliasing between `(rnum, rden)` and `(anum, aden)` or any of the coefficients of f is not supported.

```
void fmpz_poly_evaluate_horner_mpq(mpq_t res, const
   fmpz_poly_t f, const mpq_t a)
```

Evaluates the polynomial f at the rational a using Horner's rule, and sets `res` to the result.

```
void _fmpz_poly_evaluate_mpq(fmpz_t rnum, fmpz_t rden,
   const fmpz * f, slong len, const fmpz_t anum, const
   fmpz_t aden)
```

Evaluates the polynomial `(f, len)` at the rational `(anum, aden)` and sets `(rnum, rden)` to the result in lowest terms.

Aliasing between `(rnum, rden)` and `(anum, aden)` or any of the coefficients of f is not supported.

```
void fmpz_poly_evaluate_mpq(mpq_t res, const fmpz_poly_t f,
   const mpq_t a)
```

Evaluates the polynomial f at the rational a and sets `res` to the result.

```
mp_limb_t _fmpz_poly_evaluate_mod(const fmpz * poly, slong
   len, mp_limb_t a, mp_limb_t n, mp_limb_t ninv)
```

Evaluates `(poly, len)` at the value a modulo n and returns the result. The last argument `ninv` must be set to the precomputed inverse of n, which can be obtained using the function `n_preinvert_limb()`.

```
mp_limb_t fmpz_poly_evaluate_mod(const fmpz_poly_t poly,
   mp_limb_t a, mp_limb_t n)
```

Evaluates `poly` at the value a modulo n and returns the result.

```
void fmpz_poly_evaluate_fmpz_vec(fmpz * res, const
   fmpz_poly_t f, const fmpz * a, slong n)
```

Evaluates `f` at the n values given in the vector `f`, writing the results to `res`.

22.28 Newton basis

```
void _fmpz_poly_monomial_to_newton(fmpz * poly, const fmpz
   * roots, slong n)
```

Converts `(poly, n)` in-place from its coefficients given in the standard monomial basis to the Newton basis for the roots $r_0, r_1, \ldots, r_{n-2}$. In other words, this determines output coefficients c_i such that

$$c_0 + c_1(x - r_0) + c_2(x - r_0)(x - r_1) + \ldots + c_{n-1}(x - r_0)(x - r_1)\cdots(x - r_{n-2})$$

is equal to the input polynomial. Uses repeated polynomial division.

```
void _fmpz_poly_newton_to_monomial(fmpz * poly, const fmpz
   * roots, slong n)
```

Converts `(poly, n)` in-place from its coefficients given in the Newton basis for the roots $r_0, r_1, \ldots, r_{n-2}$ to the standard monomial basis. In other words, this evaluates

$$c_0 + c_1(x - r_0) + c_2(x - r_0)(x - r_1) + \ldots + c_{n-1}(x - r_0)(x - r_1)\cdots(x - r_{n-2})$$

where c_i are the input coefficients for `poly`. Uses Horner's rule.

22.29 Interpolation

```
void fmpz_poly_interpolate_fmpz_vec(fmpz_poly_t poly, const
   fmpz * xs, const fmpz * ys, slong n)
```

Sets `poly` to the unique interpolating polynomial of degree at most $n-1$ satisfying $f(x_i) = y_i$ for every pair x_i, y_u in `xs` and `ys`, assuming that this polynomial has integer coefficients.

If an interpolating polynomial with integer coefficients does not exist, the result is undefined.

It is assumed that the x values are distinct.

22.30 Composition

```
void _fmpz_poly_compose_horner(fmpz * res, const fmpz *
   poly1, slong len1, const fmpz * poly2, slong len2)
```

Sets `res` to the composition of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `res` has space for `(len1-1)*(len2-1)+ 1` coefficients. Assumes that `poly1` and `poly2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fmpz_poly_compose_horner(fmpz_poly_t res, const
   fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Sets `res` to the composition of `poly1` and `poly2`. To be more precise, denoting `res`, `poly1`, and `poly2` by f, g, and h, sets $f(t) = g(h(t))$.

This implementation uses Horner's method.

```
void _fmpz_poly_compose_divconquer(fmpz * res, const fmpz *
   poly1, slong len1, const fmpz * poly2, slong len2)
```

Computes the composition of `(poly1, len1)` and `(poly2, len2)` using a divide and conquer approach and places the result into `res`, assuming `res` can hold the output of length `(len1 - 1)* (len2 - 1)+ 1`.

Assumes `len1, len2 > 0`. Does not support aliasing between `res` and any of `(poly1, len1)` and `(poly2, len2)`.

```
void fmpz_poly_compose_divconquer(fmpz_poly_t res, const
   fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Sets `res` to the composition of `poly1` and `poly2`. To be precise about the order of composition, denoting `res`, `poly1`, and `poly2` by f, g, and h, respectively, sets $f(t) = g(h(t))$.

```
void _fmpz_poly_compose(fmpz * res, const fmpz * poly1,
   slong len1, const fmpz * poly2, slong len2)
```

Sets `res` to the composition of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `res` has space for `(len1-1)*(len2-1)+ 1` coefficients. Assumes that `poly1` and `poly2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fmpz_poly_compose(fmpz_poly_t res, const fmpz_poly_t
   poly1, const fmpz_poly_t poly2)
```

Sets `res` to the composition of `poly1` and `poly2`. To be precise about the order of composition, denoting `res`, `poly1`, and `poly2` by f, g, and h, respectively, sets $f(t) = g(h(t))$.

22.31 Taylor shift

```
void _fmpz_poly_taylor_shift_horner(fmpz * poly, const
   fmpz_t c, slong n)
```

Performs the Taylor shift composing `poly` by $x + c$ in-place. Uses an efficient version Horner's rule.

```
void fmpz_poly_taylor_shift_horner(fmpz_poly_t g, const
   fmpz_poly_t f, const fmpz_t c)
```

Performs the Taylor shift composing `f` by $x + c$.

```
void _fmpz_poly_taylor_shift_divconquer(fmpz * poly, const
   fmpz_t c, slong n)
```

Performs the Taylor shift composing `poly` by $x+c$ in-place. Uses the divide-and-conquer polynomial composition algorithm.

```
void fmpz_poly_taylor_shift_divconquer(fmpz_poly_t g, const
   fmpz_poly_t f, const fmpz_t c)
```

Performs the Taylor shift composing `f` by $x+c$. Uses the divide-and-conquer polynomial composition algorithm.

```
void _fmpz_poly_taylor_shift(fmpz * poly, const fmpz_t c,
   slong n)
```

Performs the Taylor shift composing `poly` by $x + c$ in-place.

```
void fmpz_poly_taylor_shift(fmpz_poly_t g, const
   fmpz_poly_t f, const fmpz_t c)
```

Performs the Taylor shift composing `f` by $x + c$.

22.32 Power series composition

```
void _fmpz_poly_compose_series_horner(fmpz * res, const
   fmpz * poly1, slong len1, const fmpz * poly2, slong
   len2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

Assumes that `len1, len2, n > 0`, that `len1, len2 <= n`, and that `(len1-1)* (len2-1)+ 1 <= n`, and that `res` has space for `n` coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses the Horner scheme.

```
void fmpz_poly_compose_series_horner(fmpz_poly_t res, const
   fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

This implementation uses the Horner scheme.

```
void _fmpz_poly_compose_series_brent_kung(fmpz * res, const
    fmpz * poly1, slong len1, const fmpz * poly2, slong
    len2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

Assumes that `len1, len2, n > 0`, that `len1, len2 <= n`, and that `(len1-1)* (len2-1)+ 1 <= n`, and that `res` has space for `n` coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses Brent-Kung algorithm 2.1 [6].

```
void fmpz_poly_compose_series_brent_kung(fmpz_poly_t res,
    const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong
    n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

This implementation uses Brent-Kung algorithm 2.1 [6].

```
void _fmpz_poly_compose_series(fmpz * res, const fmpz *
    poly1, slong len1, const fmpz * poly2, slong len2, slong
    n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

Assumes that `len1, len2, n > 0`, that `len1, len2 <= n`, and that `(len1-1)* (len2-1)+ 1 <= n`, and that `res` has space for `n` coefficients. Does not support aliasing between any of the inputs and the output.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs.

```
void fmpz_poly_compose_series(fmpz_poly_t res, const
    fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs.

22.33 Power series reversion

```
void _fmpz_poly_revert_series_lagrange(fmpz * Qinv, const
    fmpz * Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. The arguments must both have length `n` and may not be aliased. It is required that $Q_0 = 0$ and $Q_1 = \pm 1$.

This implementation uses the Lagrange inversion formula.

```
void fmpz_poly_revert_series_lagrange(fmpz_poly_t Qinv,
    const fmpz_poly_t Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. It is required that $Q_0 = 0$ and $Q_1 = \pm 1$.

This implementation uses the Lagrange inversion formula.

```
void _fmpz_poly_revert_series_lagrange_fast(fmpz * Qinv,
    const fmpz * Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. The arguments must both have length `n` and may not be aliased. It is required that $Q_0 = 0$ and $Q_1 = \pm 1$.

This implementation uses a reduced-complexity implementation of the Lagrange inversion formula.

```
void fmpz_poly_revert_series_lagrange_fast(fmpz_poly_t
    Qinv, const fmpz_poly_t Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. It is required that $Q_0 = 0$ and $Q_1 = \pm 1$.

This implementation uses a reduced-complexity implementation of the Lagrange inversion formula.

```
void _fmpz_poly_revert_series_newton(fmpz * Qinv, const
    fmpz * Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. The arguments must both have length `n` and may not be aliased. It is required that $Q_0 = 0$ and $Q_1 = \pm 1$.

This implementation uses Newton iteration [6].

```
void fmpz_poly_revert_series_newton(fmpz_poly_t Qinv, const
    fmpz_poly_t Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. It is required that $Q_0 = 0$ and $Q_1 = \pm 1$.

This implementation uses Newton iteration [6].

```
void _fmpz_poly_revert_series(fmpz * Qinv, const fmpz * Q,
    slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. The arguments must both have length `n` and may not be aliased. It is required that $Q_0 = 0$ and $Q_1 = \pm 1$.

This implementation defaults to the fast version of Lagrange interpolation.

```
void fmpz_poly_revert_series(fmpz_poly_t Qinv, const
    fmpz_poly_t Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. It is required that $Q_0 = 0$ and $Q_1 = \pm 1$.

This implementation defaults to the fast version of Lagrange interpolation.

22.34 Square root

```
int _fmpz_poly_sqrt_classical(fmpz * res, const fmpz *
    poly, slong len)
```

If `(poly, len)` is a perfect square, sets `(res, len / 2 + 1)` to the square root of `poly` with positive leading coefficient and returns 1. Otherwise returns 0.

This function first uses various tests to detect nonsquares quickly. Then, it computes the square root iteratively from top to bottom, requiring $O(n^2)$ coefficient operations.

```
int fmpz_poly_sqrt_classical(fmpz_poly_t b, const
    fmpz_poly_t a)
```

If `a` is a perfect square, sets `b` to the square root of `a` with positive leading coefficient and returns 1. Otherwise returns 0.

```
int _fmpz_poly_sqrt(fmpz * res, const fmpz * poly, slong
    len)
```

If `(poly, len)` is a perfect square, sets `(res, len / 2 + 1)` to the square root of `poly` with positive leading coefficient and returns 1. Otherwise returns 0.

```
int fmpz_poly_sqrt(fmpz_poly_t b, const fmpz_poly_t a)
```

If `a` is a perfect square, sets `b` to the square root of `a` with positive leading coefficient and returns 1. Otherwise returns 0.

22.35 Signature

```
void _fmpz_poly_signature(slong * r1, slong * r2, const
    fmpz * poly, slong len)
```

Computes the signature (r_1, r_2) of the polynomial `(poly, len)`. Assumes that the polynomial is squarefree over $\mathbf{Q}$.

```
void fmpz_poly_signature(slong * r1, slong * r2, const
    fmpz_poly_t poly)
```

Computes the signature (r_1, r_2) of the polynomial `poly`, which is assumed to be square-free over $\mathbf{Q}$. The values of r_1 and $2r_2$ are the number of real and complex roots of the polynomial, respectively. For convenience, the zero polynomial is allowed, in which case the output is $(0, 0)$.

If the polynomial is not square-free, the behaviour is undefined and an exception may be raised.

This function uses the algorithm described in [8, Algorithm 4.1.11].

22.36 Hensel lifting

```
void fmpz_poly_hensel_build_tree(slong * link, fmpz_poly_t
    *v, fmpz_poly_t *w, const nmod_poly_factor_t fac)
```

Initialises and builds a Hensel tree consisting of two arrays v, w of polynomials an array of links, called `link`.

The caller supplies a set of r local factors (in the factor structure `fac`) of some polynomial F over $\mathbf{Z}$. They also supply two arrays of initialised polynomials v and w, each of length $2r - 2$ and an array `link`, also of length $2r - 2$.

We will have five arrays: a v of `fmpz_poly_t`'s and a V of `nmod_poly_t`'s and also a w and a W and `link`. Here's the idea: we sort each leaf and node of a factor tree by degree, in fact choosing to multiply the two smallest factors, then the next two smallest (factors or products) etc. until a tree is made. The tree will be stored in the v's. The first two elements of v will be the smallest modular factors, the last two elements of v

will multiply to form F itself. Since v will be rearranging the original factors we will need to be able to recover the original order. For this we use the array `link` which has nonnegative even numbers and negative numbers. It is an array of `slong`'s which aligns with V and v if `link` has a negative number in spot j that means V_j is an original modular factor which has been lifted, if `link[j]` is a nonnegative even number then V_j stores a product of the two entries at `V[link[j]]` and `V[link[j]+1]`. W and w play the role of the extended GCD, at V_0, V_2, V_4, etc. we have a new product, W_0, W_2, W_4, etc. are the XGCD cofactors of the V's. For example, $V_0W_0 + V_1W_1 \equiv 1 \pmod{p^\ell}$ for some ℓ. These will be lifted along with the entries in V. It is not enough to just lift each factor, we have to lift the entire tree and the tree of XGCD cofactors.

```
void fmpz_poly_hensel_lift(fmpz_poly_t G, fmpz_poly_t H,
    fmpz_poly_t A, fmpz_poly_t B, const fmpz_poly_t f, const
    fmpz_poly_t g, const fmpz_poly_t h, const fmpz_poly_t a,
    const fmpz_poly_t b, const fmpz_t p, const fmpz_t p1)
```

This is the main Hensel lifting routine, which performs a Hensel step from polynomials mod p to polynomials mod $P = pp_1$. One starts with polynomials f, g, h such that $f = gh \pmod{p}$. The polynomials a, b satisfy $ag + bh = 1 \pmod{p}$.

The lifting formulae are

$$\begin{aligned}
G &= \left((\frac{f - gh}{p}) b \bmod g \right) p + g \\
H &= \left((\frac{f - gh}{p}) a \bmod h \right) p + h \\
B &= \left((\frac{1 - aG - bH}{p}) b \bmod g \right) p + b \\
A &= \left((\frac{1 - aG - bH}{p}) a \bmod h \right) p + a.
\end{aligned}$$

Upon return we have $AG + BH = 1 \pmod{P}$ and $f = GH \pmod{P}$, where $G = g \pmod{p}$ etc.

We require that $1 < p_1 \leq p$ and that the input polynomials f, g, h have degree at least 1 and that the input polynomials a and b are non-zero.

The output arguments G, H, A, B may only be aliased with the input arguments g, h, a, b, respectively.

```
void fmpz_poly_hensel_lift_without_inverse(fmpz_poly_t
    Gout, fmpz_poly_t Hout, const fmpz_poly_t f, const
    fmpz_poly_t g, const fmpz_poly_t h, const fmpz_poly_t a,
    const fmpz_poly_t b, const fmpz_t p, const fmpz_t p1)
```

Given polynomials such that $f = gh \pmod{p}$ and $ag + bh = 1 \pmod{p}$, lifts only the factors g and h modulo $P = pp_1$.

See `fmpz_poly_hensel_lift()`.

```
void fmpz_poly_hensel_lift_only_inverse(fmpz_poly_t Aout,
    fmpz_poly_t Bout, const fmpz_poly_t G, const fmpz_poly_t
    H, const fmpz_poly_t a, const fmpz_poly_t b, const
    fmpz_t p, const fmpz_t p1)
```

Given polynomials such that $f = gh \pmod{p}$ and $ag + bh = 1 \pmod{p}$, lifts only the cofactors a and b modulo $P = pp_1$.

See `fmpz_poly_hensel_lift()`.

```
void fmpz_poly_hensel_lift_tree_recursive(slong *link,
   fmpz_poly_t *v, fmpz_poly_t *w, fmpz_poly_t f, slong j,
   slong inv, const fmpz_t p0, const fmpz_t p1)
```

Takes a current Hensel tree `(link, v, w)` and a pair $(j, j+1)$ of entries in the tree and lifts the tree from mod p_0 to mod $P = p_0 p_1$, where $1 < p_1 \leq p_0$.

Set `inv` to -1 if restarting Hensel lifting, 0 if stopping and 1 otherwise.

Here $f = gh$ is the polynomial whose factors we are trying to lift. We will have that `v[j]` is the product of `v[link[j]]` and `v[link[j] + 1]` as described above.

Does support aliasing of f with one of the polynomials in the lists v and w. But the polynomials in these two lists are not allowed to be aliases of each other.

```
void fmpz_poly_hensel_lift_tree(slong *link, fmpz_poly_t
   *v, fmpz_poly_t *w, fmpz_poly_t f, slong r, const fmpz_t
   p, slong e0, slong e1, slong inv)
```

Computes $p_0 = p^{e_0}$ and $p_1 = p^{e_1 - e_0}$ for a small prime p and $P = p^{e_1}$.

If we aim to lift to p^b then f is the polynomial whose factors we wish to lift, made monic mod p^b. As usual, `(link, v, w)` is an initialised tree.

This starts the recursion on lifting the *product tree* for lifting from p^{e_0} to p^{e_1}. The value of `inv` corresponds to that given for the function `fmpz_poly_hensel_lift_tree_recursive()`. We set r to the number of local factors of f.

In terms of the notation, above $P = p^{e_1}$, $p_0 = p^{e_0}$ and $p_1 = p^{e_1 - e_0}$.

Assumes that f is monic.

Assumes that $1 < p_1 \leq p_0$, that is, $0 < e_1 \leq e_0$.

```
slong _fmpz_poly_hensel_start_lift(fmpz_poly_factor_t
   lifted_fac, slong *link, fmpz_poly_t *v, fmpz_poly_t *w,
   const fmpz_poly_t f, const nmod_poly_factor_t local_fac,
   slong N)
```

This function takes the local factors in `local_fac` and Hensel lifts them until they are known mod p^N, where $N \geq 1$.

These lifted factors will be stored (in the same ordering) in `lifted_fac`. It is assumed that `link`, `v`, and `w` are initialized arrays `fmpz_poly_t`'s with at least $2 * r - 2$ entries and that $r \geq 2$. This is done outside of this function so that you can keep them for restarting Hensel lifting later. The product of local factors must be squarefree.

The return value is an exponent which must be passed to the function `_fmpz_poly_hensel_continue_lift()` as `prev_exp` if the Hensel lifting is to be resumed.

Currently, supports the case when $N = 1$ for convenience, although it is preferable in this case to simple iterate over the local factors and convert them to polynomials over $\mathbf{Z}$.

```
slong _fmpz_poly_hensel_continue_lift(fmpz_poly_factor_t
   lifted_fac, slong *link, fmpz_poly_t *v, fmpz_poly_t *w,
   const fmpz_poly_t f, slong prev, slong curr, slong N,
   const fmpz_t p)
```

This function restarts a stopped Hensel lift.

It lifts from `curr` to N. It also requires `prev` (to lift the cofactors) given as the return value of the function `_fmpz_poly_hensel_start_lift()` or the function `_fmpz_poly_hensel_continue_lift()`. The current lifted factors are supplied in `lifted_fac`

and upon return are updated there. As usual `link`, `v`, and `w` describe the current Hensel tree, r is the number of local factors and p is the small prime modulo whose power we are lifting to. It is required that `curr` be at least 1 and that `N > curr`.

Currently, supports the case when `prev` and `curr` are equal.

```
void fmpz_poly_hensel_lift_once(fmpz_poly_factor_t
   lifted_fac, const fmpz_poly_t f, const
   nmod_poly_factor_t local_fac, slong N)
```

This function does a Hensel lift.

It lifts local factors stored in `local_fac` of f to p^N, where $N \geq 2$. The lifted factors will be stored in `lifted_fac`. This lift cannot be restarted. This function is a convenience function intended for end users. The product of local factors must be squarefree.

22.37 Input and output

The functions in this section are not intended to be particularly fast. They are intended mainly as a debugging aid.

For the string output functions there are two variants. The first uses a simple string representation of polynomials which prints only the length of the polynomial and the integer coefficients, whilst the latter variant, appended with `_pretty`, uses a more traditional string representation of polynomials which prints a variable name as part of the representation.

The first string representation is given by a sequence of integers, in decimal notation, separated by white space. The first integer gives the length of the polynomial; the remaining integers are the coefficients. For example $5x^3 - x + 1$ is represented by the string `"4  1 -1 0 5"`, and the zero polynomial is represented by `"0"`. The coefficients may be signed and arbitrary precision.

The string representation of the functions appended by `_pretty` includes only the non-zero terms of the polynomial, starting with the one of highest degree. Each term starts with a coefficient, prepended with a sign, followed by the character `*`, followed by a variable name, which must be passed as a string parameter to the function, followed by a carot `^` followed by a non-negative exponent.

If the sign of the leading coefficient is positive, it is omitted. Also the exponents of the degree 1 and 0 terms are omitted, as is the variable and the `*` character in the case of the degree 0 coefficient. If the coefficient is plus or minus one, the coefficient is omitted, except for the sign.

Some examples of the `_pretty` representation are:

```
5*x^3+7*x-4
x^2+3
-x^4+2*x-1
x+1
5
```

```
int _fmpz_poly_print(const fmpz * poly, slong len)
```

Prints the polynomial `(poly, len)` to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_poly_print(const fmpz_poly_t poly)
```

Prints the polynomial to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fmpz_poly_print_pretty(const fmpz * poly, slong len,
    const char * x)
```

Prints the pretty representation of `(poly, len)` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_poly_print_pretty(const fmpz_poly_t poly, const
    char * x)
```

Prints the pretty representation of `poly` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fmpz_poly_fprint(FILE * file, const fmpz * poly, slong
    len)
```

Prints the polynomial `(poly, len)` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_poly_fprint(FILE * file, const fmpz_poly_t poly)
```

Prints the polynomial to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fmpz_poly_fprint_pretty(FILE * file, const fmpz *
    poly, slong len, char * x)
```

Prints the pretty representation of `(poly, len)` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_poly_fprint_pretty(FILE * file, const fmpz_poly_t
    poly, char * x)
```

Prints the pretty representation of `poly` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_poly_read(fmpz_poly_t poly)
```

Reads a polynomial from `stdin`, storing the result in `poly`.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

```
int fmpz_poly_read_pretty(fmpz_poly_t poly, char **x)
```

Reads a polynomial in pretty format from `stdin`.

For further details, see the documentation for the function `fmpz_poly_fread_pretty()`.

```
int fmpz_poly_fread(FILE * file, fmpz_poly_t poly)
```

Reads a polynomial from the stream `file`, storing the result in `poly`.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

```
int fmpz_poly_fread_pretty(FILE *file, fmpz_poly_t poly,
    char **x)
```

Reads a polynomial from the file `file` and sets `poly` to this polynomial. The string `*x` is set to the variable name that is used in the input.

The parser is implemented via a finite state machine as follows:

state	event	next state
0	'-'	1
	D	2
	V0	3
1	D	2
	V0	3
2	D	2
	'*'	4
	'+', '-'	1
3	V	3
	'^'	5
	'+', '-'	1
4	V0	3
5	D	6
6	D	6
	'+', '-'	1

Here, `D` refers to any digit, `V0` to any character which is allowed as the first character in the variable name (an alphetic character), and `V` to any character which is allowed in the remaining part of the variable name (an alphanumeric character or underscore).

Once we encounter a character which does not fit into the above pattern, we stop.

Returns a positive value, equal to the number of characters read from the file, in case of success. Returns a non-positive value in case of failure, which could either be a read error or the indicator of a malformed input.

22.38 Modular reduction and reconstruction

```
void fmpz_poly_get_nmod_poly(nmod_poly_t Amod, fmpz_poly_t
    A)
```

Sets the coefficients of `Amod` to the coefficients in `A`, reduced by the modulus of `Amod`.

```
void fmpz_poly_set_nmod_poly(fmpz_poly_t A, const
    nmod_poly_t Amod)
```

Sets the coefficients of `A` to the residues in `Amod`, normalised to the interval $-m/2 \leq r < m/2$ where m is the modulus.

```
void fmpz_poly_set_nmod_poly_unsigned(fmpz_poly_t A, const
    nmod_poly_t Amod)
```

Sets the coefficients of `A` to the residues in `Amod`, normalised to the interval $0 \le r < m$ where m is the modulus.

```
void _fmpz_poly_CRT_ui_precomp(fmpz * res, const fmpz *
   poly1, slong len1, const fmpz_t m1, mp_srcptr poly2,
   slong len2, mp_limb_t m2, mp_limb_t m2inv, fmpz_t m1m2,
   mp_limb_t c, int sign)
```

Sets the coefficients in `res` to the CRT reconstruction modulo $m_1 m_2$ of the residues `(poly1, len1)` and `(poly2, len2)` which are images modulo m_1 and m_2 respectively. The caller must supply the precomputed product of the input moduli as $m_1 m_2$, the inverse of m_1 modulo m_2 as c, and the precomputed inverse of m_2 (in the form computed by `n_preinvert_limb`) as `m2inv`.

If `sign` $= 0$, residues $0 <= r < m_1 m_2$ are computed, while if `sign` $= 1$, residues $-m_1 m_2/2 <= r < m_1 m_2/2$ are computed.

Coefficients of `res` are written up to the maximum of `len1` and `len2`.

```
void _fmpz_poly_CRT_ui(fmpz * res, const fmpz * poly1,
   slong len1, const fmpz_t m1, mp_srcptr poly2, slong
   len2, mp_limb_t m2, mp_limb_t m2inv, int sign)
```

This function is identical to `_fmpz_poly_CRT_ui_precomp`, apart from automatically computing $m_1 m_2$ and c. It also aborts if c cannot be computed.

```
void fmpz_poly_CRT_ui(fmpz_poly_t res, const fmpz_poly_t
   poly1, const fmpz_t m, const nmod_poly_t poly2, int sign)
```

Given `poly1` with coefficients modulo `m` and `poly2` with modulus n, sets `res` to the CRT reconstruction modulo mn with coefficients satisfying $-mn/2 \le c < mn/2$ (if sign $= 1$) or $0 \le c < mn$ (if sign $= 0$).

22.39 Products

```
void _fmpz_poly_product_roots_fmpz_vec(fmpz * poly, const
   fmpz * xs, slong n)
```

Sets `(poly, n + 1)` to the monic polynomial which is the product of $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$, the roots x_i being given by `xs`.

Aliasing of the input and output is not allowed.

```
void fmpz_poly_product_roots_fmpz_vec(fmpz_poly_t poly,
   const fmpz * xs, slong n)
```

Sets `poly` to the monic polynomial which is the product of $(x-x_0)(x-x_1)\cdots(x-x_{n-1})$, the roots x_i being given by `xs`.

22.40 Newton basis conversion

```
void _fmpz_poly_monomial_to_newton(fmpz * poly, const fmpz
   * roots, slong n)
```

Converts the polynomial in-place from its coefficients in the monomial basis to the Newton basis $1, (x-r_0), (x-r_0)(x-r_1), \ldots$. Uses Horner's rule, requiring $O(n^2)$ operations.

```
void _fmpz_poly_newton_to_monomial(fmpz * poly, const fmpz
   * roots, slong n)
```

Converts the polynomial in-place from its coefficients in the Newton basis $1, (x-r_0), (x-r_0)(x-r_1), \ldots$ to the monomial basis. Uses repeated polynomial division, requiring $O(n^2)$ operations.

22.41 Roots

```
void _fmpz_poly_bound_roots(fmpz_t bound, const fmpz *
    poly, slong len)

void fmpz_poly_bound_roots(fmpz_t bound, const fmpz_poly_t
    poly)
```

Computes a nonnegative integer `bound` that bounds the absolute value of all complex roots of `poly`. Uses Fujiwara's bound

$$2 \max \left(\left| \frac{a_{n-1}}{a_n} \right|, \left| \frac{a_{n-2}}{a_n} \right|^{\frac{1}{2}}, \ldots \left| \frac{a_1}{a_n} \right|^{\frac{1}{n-1}}, \left| \frac{a_0}{2a_n} \right|^{\frac{1}{n}} \right)$$

where the coefficients of the polynomial are $a_0, \ldots, a_n$.

§23. fmpz_poly_factor

Factorisation of polynomials over **Z**

The `fmpz_poly_factor` module is included automatically with `fmpz_poly.h`. One should not try to include `fmpz_poly_factor.h` directly.

23.1 Memory management

```
void fmpz_poly_factor_init(fmpz_poly_factor_t fac)
```

Initialises a new factor structure.

```
void fmpz_poly_factor_init2(fmpz_poly_factor_t fac, slong
    alloc)
```

Initialises a new factor structure, providing space for at least `alloc` factors.

```
void fmpz_poly_factor_realloc(fmpz_poly_factor_t fac, slong
    alloc)
```

Reallocates the factor structure to provide space for precisely `alloc` factors.

```
void fmpz_poly_factor_fit_length(fmpz_poly_factor_t fac,
    slong len)
```

Ensures that the factor structure has space for at least `len` factors. This functions takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

```
void fmpz_poly_factor_clear(fmpz_poly_factor_t fac)
```

Releases all memory occupied by the factor structure.

23.2 Manipulating factors

```
void fmpz_poly_factor_set(fmpz_poly_factor_t res, const
    fmpz_poly_factor_t fac)
```

Sets `res` to the same factorisation as `fac`.

```
void fmpz_poly_factor_insert(fmpz_poly_factor_t fac, const
    fmpz_poly_t p, slong e)
```

Adds the primitive polynomial p^e to the factorisation `fac`.

Assumes that $\deg(p) \geq 2$ and $e \neq 0$.

```
void fmpz_poly_factor_concat(fmpz_poly_factor_t res, const
    fmpz_poly_factor_t fac)
```

Concatenates two factorisations.

This is equivalent to calling `fmpz_poly_factor_insert()` repeatedly with the individual factors of `fac`.

Does not support aliasing between `res` and `fac`.

23.3 Input and output

```
void fmpz_poly_factor_print(const fmpz_poly_factor_t fac)
```

Prints the entries of `fac` to standard output.

23.4 Factoring algorithms

```
void fmpz_poly_factor_squarefree(fmpz_poly_factor_t fac,
    fmpz_poly_t F)
```

Takes as input a polynomial F and a freshly initialized factor structure `fac`. Updates `fac` to contain a factorization of F into (not necessarily irreducible) factors that themselves have no repeated factors. None of the returned factors will have the same exponent. That is we return g_i and unique e_i such that

$$F = c \prod_i g_i^{e_i}$$

where c is the signed content of F and $\gcd(g_i, g_i') = 1$.

```
void
    fmpz_poly_factor_zassenhaus_recombination(fmpz_poly_factor_t
    final_fac, const fmpz_poly_factor_t lifted_fac, const
    fmpz_poly_t F, const fmpz_t P, slong exp)
```

Takes as input a factor structure `lifted_fac` containing a squarefree factorization of the polynomial F mod p. The algorithm does a brute force search for irreducible factors of F over the integers, and each factor is raised to the power `exp`.

The impact of the algorithm is to augment a factorization of `F^exp` to the factor structure `final_fac`.

```
void _fmpz_poly_factor_zassenhaus(fmpz_poly_factor_t
    final_fac, slong exp, fmpz_poly_t f, slong cutoff)
```

This is the internal wrapper of Zassenhaus.

It will attempt to find a small prime such that f modulo p has a minimal number of factors. If it cannot find a prime giving less than `cutoff` factors it aborts. Then it decides a p-adic precision to lift the factors to, hensel lifts, and finally calls Zassenhaus recombination.

Assumes that $\text{len}(f) \geq 2$.

Assumes that f is primitive.

Assumes that the constant coefficient of f is non-zero. Note that this can be easily achieved by taking out factors of the form x^k before calling this routine.

```
void fmpz_poly_factor_zassenhaus(fmpz_poly_factor_t
   final_fac, fmpz_poly_t F)
```

A wrapper of the Zassenhaus factoring algorithm, which takes as input any polynomial F, and stores a factorization in `final_fac`.

The complexity will be exponential in the number of local factors we find for the components of a squarefree factorization of F.

§24. fmpq

Arbitrary-precision rational numbers

24.1 Introduction

The `fmpq_t` data type represents rational numbers as fractions of multiprecision integers.

An `fmpq_t` is an array of length 1 of type `fmpq`, with `fmpq` being implemented as a pair of `fmpz`'s representing numerator and denominator.

This format is designed to allow rational numbers with small numerators or denominators to be stored and manipulated efficiently. When components no longer fit in single machine words, the cost of `fmpq_t` arithmetic is roughly the same as that of `mpq_t` arithmetic, plus a small amount of overhead.

A fraction is said to be in canonical form if the numerator and denominator have no common factor and the denominator is positive. Except where otherwise noted, all functions in the `fmpq` module assume that inputs are in canonical form, and produce outputs in canonical form. The user can manipulate the numerator and denominator of an `fmpq_t` as arbitrary integers, but then becomes responsible for canonicalising the number (for example by calling `fmpq_canonicalise`) before passing it to any library function.

For most operations, both a function operating on `fmpq_t`'s and an underscore version operating on `fmpz_t` components are provided. The underscore functions may perform less error checking, and may impose limitations on aliasing between the input and output variables, but generally assume that the components are in canonical form just like the non-underscore functions.

24.2 Memory management

```
void fmpq_init(fmpq_t x)
```

Initialises the `fmpq_t` variable `x` for use. Its value is set to 0.

```
void fmpq_clear(fmpq_t x)
```

Clears the `fmpq_t` variable `x`. To use the variable again, it must be re-initialised with `fmpq_init`.

```
fmpq * _fmpq_vec_init(slong n)
```

Initialises a vector of `fmpq` values of length n and sets all values to 0. This is equivalent to generating a `fmpz` vector of length $2n$ with `_fmpz_vec_init` and setting all denominators to 1.

```
void _fmpq_vec_clear(fmpq * vec, slong n)
```

Frees an `fmpq` vector.

24.3 Canonicalisation

```
void fmpq_canonicalise(fmpq_t res)
```

Puts `res` in canonical form: the numerator and denominator are reduced to lowest terms, and the denominator is made positive. If the numerator is zero, the denominator is set to one.

If the denominator is zero, the outcome of calling this function is undefined, regardless of the value of the numerator.

```
void _fmpq_canonicalise(fmpz_t num, fmpz_t den)
```

Does the same thing as `fmpq_canonicalise`, but for numerator and denominator given explicitly as `fmpz_t` variables. Aliasing of `num` and `den` is not allowed.

```
int fmpq_is_canonical(const fmpq_t x)
```

Returns nonzero if `fmpq_t` x is in canonical form (as produced by `fmpq_canonicalise`), and zero otherwise.

```
int _fmpq_is_canonical(const fmpz_t num, const fmpz_t den)
```

Does the same thing as `fmpq_is_canonical`, but for numerator and denominator given explicitly as `fmpz_t` variables.

24.4 Basic assignment

```
void fmpq_set(fmpq_t dest, const fmpq_t src)
```

Sets `dest` to a copy of `src`. No canonicalisation is performed.

```
void fmpq_swap(fmpq_t op1, fmpq_t op2)
```

Swaps the two rational numbers `op1` and `op2`.

```
void fmpq_neg(fmpq_t dest, const fmpq_t src)
```

Sets `dest` to the additive inverse of `src`.

```
void fmpq_abs(fmpq_t dest, const fmpq_t src)
```

Sets `dest` to the absolute value of `src`.

```
void fmpq_zero(fmpq_t res)
```

Sets the value of `res` to 0.

```
void fmpq_one(fmpq_t res)
```

Sets the value of `res` to 1.

24.5 Comparison

```
int fmpq_is_zero(const fmpq_t res)
```

Returns nonzero if `res` has value 0, and returns zero otherwise.

```
int fmpq_is_one(const fmpq_t res)
```

Returns nonzero if `res` has value 1, and returns zero otherwise.

```
int fmpq_equal(const fmpq_t x, const fmpq_t y)
```

Returns nonzero if `x` and `y` are equal, and zero otherwise. Assumes that `x` and `y` are both in canonical form.

```
int fmpq_sgn(const fmpq_t x)
```

Returns the sign of the rational number x.

```
int fmpq_cmp(const fmpq_t x, const fmpq_t y)
```

Returns negative if $x < y$, zero if $x = y$, and positive if $x > y$.

```
void fmpq_height(fmpz_t height, const fmpq_t x)
```

Sets `height` to the height of x, defined as the larger of the absolute values of the numerator and denominator of x.

```
mp_bitcnt_t fmpq_height_bits(const fmpq_t x)
```

Returns the number of bits in the height of x.

24.6 Conversion

```
void fmpq_set_fmpz_frac(fmpq_t res, const fmpz_t p, const
    fmpz_t q)
```

Sets `res` to the canonical form of the fraction `p / q`. This is equivalent to assigning the numerator and denominator separately and calling `fmpq_canonicalise`.

```
void fmpq_set_si(fmpq_t res, slong p, ulong q)
```

Sets `res` to the canonical form of the fraction `p / q`.

```
void _fmpq_set_si(fmpz_t rnum, fmpz_t rden, slong p, ulong
    q)
```

Sets `(rnum, rden)` to the canonical form of the fraction `p / q`. `rnum` and `rden` may not be aliased.

```
void fmpq_set_mpq(fmpq_t dest, const mpq_t src)
```

Sets the value of `dest` to that of the `mpq_t` variable `src`.

```
void fmpq_get_mpq(mpq_t dest, const fmpq_t src)
```

Sets the value of `dest`

```
int fmpq_get_mpfr(mpfr_t dest, const fmpq_t src, mpfr_rnd_t
    rnd)
```

Sets the MPFR variable `dest` to the value of `src`, rounded to the nearest representable binary floating-point value in direction `rnd`. Returns the sign of the rounding, according to MPFR conventions.

```
char * _fmpq_get_str(char * str, int b, const fmpz_t num,
   const fmpz_t den)
```

```
char * fmpq_get_str(char * str, int b, const fmpq_t x)
```

Prints the string representation of x in base $b \in [2, 36]$ to a suitable buffer.

If `str` is not `NULL`, this is used as the buffer and also the return value. If `str` is `NULL`, allocates sufficient space and returns a pointer to the string.

```
void flint_mpq_init_set_readonly(mpq_t z, const fmpq_t f)
```

Sets the unitialised `mpq_t` z to the value of the readonly `fmpq_t` f.

Note that it is assumed that f does not change during the lifetime of z.

The rational z has to be cleared by a call to `flint_mpq_clear_readonly()`.

The suggested use of the two functions is as follows:

```
fmpq_t f;
...
{
    mpq_t z;

    flint_mpq_init_set_readonly(z, f);
    foo(..., z);
    flint_mpq_clear_readonly(z);
}
```

This provides a convenient function for user code, only requiring to work with the types `fmpq_t` and `mpq_t`.

```
void flint_mpq_clear_readonly(mpq_t z)
```

Clears the readonly `mpq_t` z.

```
void fmpq_init_set_readonly(fmpq_t f, const mpq_t z)
```

Sets the uninitialised `fmpq_t` f to a readonly version of the rational z.

Note that the value of z is assumed to remain constant throughout the lifetime of f.

The `fmpq_t` f has to be cleared by calling the function `fmpq_clear_readonly()`.

The suggested use of the two functions is as follows:

```
mpq_t z;
...
{
    fmpq_t f;

    fmpq_init_set_readonly(f, z);
    foo(..., f);
    fmpq_clear_readonly(f);
}
```

```
void fmpq_clear_readonly(fmpq_t f)
```

Clears the readonly `fmpq_t` f.

24.7 Input and output

```
void fmpq_fprint(FILE * file, const fmpq_t x)
```

Prints `x` as a fraction to the stream `file`. The numerator and denominator are printed verbatim as integers, with a forward slash (/) printed in between.

```
void _fmpq_fprint(FILE * file, const fmpz_t num, const
    fmpz_t den)
```

Does the same thing as `fmpq_fprint`, but for numerator and denominator given explicitly as `fmpz_t` variables.

```
void fmpq_print(const fmpq_t x)
```

Prints `x` as a fraction. The numerator and denominator are printed verbatim as integers, with a forward slash (/) printed in between.

```
void _fmpq_print(const fmpz_t num, const fmpz_t den)
```

Does the same thing as `fmpq_print`, but for numerator and denominator given explicitly as `fmpz_t` variables.

24.8 Random number generation

```
void fmpq_randtest(fmpq_t res, flint_rand_t state,
    mp_bitcnt_t bits)
```

Sets `res` to a random value, with numerator and denominator having up to `bits` bits. The fraction will be in canonical form. This function has an increased probability of generating special values which are likely to trigger corner cases.

```
void _fmpq_randtest(fmpz_t num, fmpz_t den, flint_rand_t
    state, mp_bitcnt_t bits)
```

Does the same thing as `fmpq_randtest`, but for numerator and denominator given explicitly as `fmpz_t` variables. Aliasing of `num` and `den` is not allowed.

```
void fmpq_randtest_not_zero(fmpq_t res, flint_rand_t state,
    mp_bitcnt_t bits)
```

As per `fmpq_randtest`, but the result will not be 0. If `bits` is set to 0, an exception will result.

```
void fmpq_randbits(fmpq_t res, flint_rand_t state,
    mp_bitcnt_t bits)
```

Sets `res` to a random value, with numerator and denominator both having exactly `bits` bits before canonicalisation, and then puts `res` in canonical form. Note that as a result of the canonicalisation, the resulting numerator and denominator can be slightly smaller than `bits` bits.

```
void _fmpq_randbits(fmpz_t num, fmpz_t den, flint_rand_t
    state, mp_bitcnt_t bits)
```

Does the same thing as `fmpq_randbits`, but for numerator and denominator given explicitly as `fmpz_t` variables. Aliasing of `num` and `den` is not allowed.

24.9 Arithmetic

```
void fmpq_add(fmpq_t res, const fmpq_t op1, const fmpq_t
    op2)
```

```
void fmpq_sub(fmpq_t res, const fmpq_t op1, const fmpq_t
   op2)

void fmpq_mul(fmpq_t res, const fmpq_t op1, const fmpq_t
   op2)

void fmpq_div(fmpq_t res, const fmpq_t op1, const fmpq_t
   op2)
```

Sets `res` respectively to `op1 + op2`, `op1 - op2`, `op1 * op2`, or `op1 / op2`. Assumes that the inputs are in canonical form, and produces output in canonical form. Division by zero results in an error. Aliasing between any combination of the variables is allowed.

```
void _fmpq_add(fmpz_t rnum, fmpz_t rden, const fmpz_t
   op1num, const fmpz_t op1den, const fmpz_t op2num, const
   fmpz_t op2den)

void _fmpq_sub(fmpz_t rnum, fmpz_t rden, const fmpz_t
   op1num, const fmpz_t op1den, const fmpz_t op2num, const
   fmpz_t op2den)

void _fmpq_mul(fmpz_t rnum, fmpz_t rden, const fmpz_t
   op1num, const fmpz_t op1den, const fmpz_t op2num, const
   fmpz_t op2den)

void _fmpq_div(fmpz_t rnum, fmpz_t rden, const fmpz_t
   op1num, const fmpz_t op1den, const fmpz_t op2num, const
   fmpz_t op2den)
```

Sets `(rnum, rden)` to the canonical form of the sum, difference, product or quotient respectively of the fractions represented by `(op1num, op1den)` and `(op2num, op2den)`. Aliasing between any combination of the variables is allowed, whilst no numerator is aliased with a denominator.

```
void fmpq_addmul(fmpq_t res, const fmpq_t op1, const fmpq_t
   op2)

void fmpq_submul(fmpq_t res, const fmpq_t op1, const fmpq_t
   op2)
```

Sets `res` to `res + op1 * op2` or `res - op1 * op2` respectively, placing the result in canonical form. Aliasing between any combination of the variables is allowed.

```
void _fmpq_addmul(fmpz_t rnum, fmpz_t rden, const fmpz_t
   op1num, const fmpz_t op1den, const fmpz_t op2num, const
   fmpz_t op2den)

void _fmpq_submul(fmpz_t rnum, fmpz_t rden, const fmpz_t
   op1num, const fmpz_t op1den, const fmpz_t op2num, const
   fmpz_t op2den)
```

Sets `(rnum, rden)` to the canonical form of the fraction `(rnum, rden)` + `(op1num, op1den)` * `(op2num, op2den)` or `(rnum, rden)` - `(op1num, op1den)` * `(op2num, op2den)` respectively. Aliasing between any combination of the variables is allowed, whilst no numerator is aliased with a denominator.

```
void fmpq_inv(fmpq_t dest, const fmpq_t src)
```

Sets `dest` to `1 / src`. The result is placed in canonical form, assuming that `src` is already in canonical form.

```
void _fmpq_pow_si(fmpz_t rnum, fmpz_t rden, const fmpz_t
    opnum, const fmpz_t opden, slong e);

void fmpq_pow_si(fmpq_t res, const fmpq_t op, slong e);
```

Sets `res` to `op` raised to the power e, where e is a `slong`. If e is 0 and `op` is 0, then `res` will be set to 1.

```
void fmpq_mul_fmpz(fmpq_t res, const fmpq_t op, const
    fmpz_t x)
```

Sets `res` to the product of the rational number `op` and the integer `x`.

```
void fmpq_div_fmpz(fmpq_t res, const fmpq_t op, const
    fmpz_t x)
```

Sets `res` to the quotient of the rational number `op` and the integer `x`.

```
void fmpq_mul_2exp(fmpq_t res, const fmpq_t x, mp_bitcnt_t
    exp)
```

Sets `res` to `x` multiplied by `2^exp`.

```
void fmpq_div_2exp(fmpq_t res, const fmpq_t x, mp_bitcnt_t
    exp)
```

Sets `res` to `x` divided by `2^exp`.

24.10 Modular reduction and rational reconstruction

```
int _fmpq_mod_fmpz(fmpz_t res, const fmpz_t num, const
    fmpz_t den, const fmpz_t mod)

int fmpq_mod_fmpz(fmpz_t res, const fmpq_t x, const fmpz_t
    mod)
```

Sets the integer `res` to the residue a of $x = n/d =$ `(num, den)` modulo the positive integer $m =$ `mod`, defined as the $0 \le a < m$ satisfying $n \equiv ad \pmod{m}$. If such an a exists, 1 will be returned, otherwise 0 will be returned.

```
int _fmpq_reconstruct_fmpz_2(fmpz_t n, fmpz_t d, const
    fmpz_t a, const fmpz_t m, const fmpz_t N, const fmpz_t D)

int fmpq_reconstruct_fmpz_2(fmpq_t res, const fmpz_t a,
    const fmpz_t m, const fmpz_t N, const fmpz_t D)
```

Reconstructs a rational number from its residue a modulo m.

Given a modulus $m > 1$, a residue $0 \le a < m$, and positive N, D satisfying $2ND < m$, this function attempts to find a fraction n/d with $0 \le |n| \le N$ and $0 < d \le D$ such that $\gcd(n, d) = 1$ and $n \equiv ad \pmod{m}$. If a solution exists, then it is also unique. The function returns 1 if successful, and 0 to indicate that no solution exists.

```
int _fmpq_reconstruct_fmpz(fmpz_t n, fmpz_t d, const fmpz_t
    a, const fmpz_t m)
```

```
int fmpq_reconstruct_fmpz(fmpq_t res, const fmpz_t a, const
    fmpz_t m)
```

Reconstructs a rational number from its residue a modulo m, returning 1 if successful and 0 if no solution exists. Uses the balanced bounds $N = D = \lfloor \sqrt{m/2} \rfloor$.

24.11 Rational enumeration

```
void _fmpq_next_minimal(fmpz_t rnum, fmpz_t rden, const
    fmpz_t num, const fmpz_t den)

void fmpq_next_minimal(fmpq_t res, const fmpq_t x)
```

Given x which is assumed to be nonnegative and in canonical form, sets `res` to the next rational number in the sequence obtained by enumerating all positive denominators q, for each q enumerating the numerators $1 \le p < q$ in order and generating both p/q and q/p, but skipping all $\gcd(p, q) \neq 1$. Starting with zero, this generates every nonnegative rational number once and only once, with the first few entries being:

$$0, 1, 1/2, 2, 1/3, 3, 2/3, 3/2, 1/4, 4, 3/4, 4/3, 1/5, 5, 2/5, \ldots.$$

This enumeration produces the rational numbers in order of minimal height. It has the disadvantage of being somewhat slower to compute than the Calkin-Wilf enumeration.

```
void _fmpq_next_signed_minimal(fmpz_t rnum, fmpz_t rden,
    const fmpz_t num, const fmpz_t den)

void fmpq_next_signed_minimal(fmpq_t res, const fmpq_t x)
```

Given a signed rational number x assumed to be in canonical form, sets `res` to the next element in the minimal-height sequence generated by `fmpq_next_minimal` but with negative numbers interleaved:

$$0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, \ldots.$$

Starting with zero, this generates every rational number once and only once, in order of minimal height.

```
void _fmpq_next_calkin_wilf(fmpz_t rnum, fmpz_t rden, const
    fmpz_t num, const fmpz_t den)

void fmpq_next_calkin_wilf(fmpq_t res, const fmpq_t x)
```

Given x which is assumed to be nonnegative and in canonical form, sets `res` to the next number in the breadth-first traversal of the Calkin-Wilf tree. Starting with zero, this generates every nonnegative rational number once and only once, with the first few entries being:

$$0, 1, 1/2, 2, 1/3, 3/2, 2/3, 3, 1/4, 4/3, 3/5, 5/2, 2/5, \ldots.$$

Despite the appearance of the initial entries, the Calkin-Wilf enumeration does not produce the rational numbers in order of height: some small fractions will appear late in the sequence. This order has the advantage of being faster to produce than the minimal-height order.

```
void _fmpq_next_signed_calkin_wilf(fmpz_t rnum, fmpz_t
    rden, const fmpz_t num, const fmpz_t den)
```

```
void fmpq_next_signed_calkin_wilf(fmpq_t res, const fmpq_t
    x)
```

Given a signed rational number x assumed to be in canonical form, sets `res` to the next element in the Calkin-Wilf sequence with negative numbers interleaved:

$$0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, \ldots.$$

Starting with zero, this generates every rational number once and only once, but not in order of minimal height.

24.12 Continued fractions

```
slong fmpq_get_cfrac(fmpz * c, fmpq_t rem, const fmpq_t x,
    slong n)
```

Generates up to n terms of the (simple) continued fraction expansion of x, writing the coefficients to the vector c and the remainder r to the `rem` variable. The return value is the number k of generated terms. The output satisfies:

$$x = c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \cfrac{1}{\ddots + \cfrac{1}{c_{k-1} + r}}}}$$

If r is zero, the continued fraction expansion is complete. If r is nonzero, $1/r$ can be passed back as input to generate $c_k, c_{k+1}, \ldots$. Calls to `fmpq_get_cfrac` can therefore be chained to generate the continued fraction incrementally, extracting any desired number of coefficients at a time.

In general, a rational number has exactly two continued fraction expansions. By convention, we generate the shorter one. The longer expansion can be obtained by replacing the last coefficient a_{k-1} by the pair of coefficients $a_{k-1} - 1, 1$.

As a special case, the continued fraction expansion of zero consists of a single zero (and not the empty sequence).

This function implements a simple algorithm, performing repeated divisions. The running time is quadratic.

```
void fmpq_set_cfrac(fmpq_t x, const fmpz * c, slong n)
```

Sets x to the value of the continued fraction

$$x = c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \cfrac{1}{\ddots + \cfrac{1}{c_{n-1}}}}}$$

where all c_i except c_0 should be nonnegative. It is assumed that $n > 0$.

For large n, this function implements a subquadratic algorithm. The convergents are given by a chain product of 2 by 2 matrices. This product is split in half recursively to balance the size of the coefficients.

`slong fmpq_cfrac_bound(const fmpq_t x)`

Returns an upper bound for the number of terms in the continued fraction expansion of x. The computed bound is not necessarily sharp.

We use the fact that the smallest denominator that can give a continued fraction of length n is the Fibonacci number F_{n+1}.

§25. fmpq_mat

Matrices over **Q**

25.1 Introduction

The `fmpq_mat_t` data type represents matrices over **Q**.

A rational matrix is stored as an array of `fmpq` elements in order to allow convenient and efficient manipulation of individual entries. In general, `fmpq_mat` functions assume that input entries are in canonical form, and produce output with entries in canonical form.

Since rational arithmetic is expensive, computations are typically performed by clearing denominators, performing the heavy work over the integers, and converting the final result back to a rational matrix. The `fmpq_mat` functions take care of such conversions transparently. For users who need fine-grained control, various functions for conversion between rational and integer matrices are provided.

25.2 Memory management

`void fmpq_mat_init(fmpq_mat_t mat, slong rows, slong cols)`

Initialises a matrix with the given number of rows and columns for use.

`void fmpq_mat_clear(fmpq_mat_t mat)`

Frees all memory associated with the matrix. The matrix must be reinitialised if it is to be used again.

25.3 Entry access

`MACRO fmpq_mat_entry(mat,i,j)`

Gives a reference to the entry at row `i` and column `j`. The reference can be passed as an input or output variable to any `fmpq` function for direct manipulation of the matrix element. No bounds checking is performed.

`MACRO fmpq_mat_entry_num(mat,i,j)`

Gives a reference to the numerator of the entry at row `i` and column `j`. The reference can be passed as an input or output variable to any `fmpz` function for direct manipulation of the matrix element. No bounds checking is performed.

`MACRO fmpq_mat_entry_den(mat,i,j)`

Gives a reference to the denominator of the entry at row `i` and column `j`. The reference can be passed as an input or output variable to any `fmpz` function for direct manipulation of the matrix element. No bounds checking is performed.

25.4 Basic assignment

`void fmpq_mat_set(fmpq_mat_t dest, const fmpq_mat_t src)`

Sets the entries in `dest` to the same values as in `src`, assuming the two matrices have the same dimensions.

`void fmpq_mat_zero(fmpq_mat_t mat)`

Sets `mat` to the zero matrix.

`void fmpq_mat_one(fmpq_mat_t mat)`

Let m be the minimum of the number of rows and columns in the matrix `mat`. This function sets the first $m \times m$ block to the identity matrix, and the remaining block to zero.

`void fmpq_mat_transpose(fmpq_mat_t rop, const fmpq_mat_t op)`

Sets the matrix `rop` to the tranpose of the matrix `op`, assuming that their dimensios are compatible.

25.5 Addition, scalar multiplication

```
void fmpq_mat_add(fmpq_mat_t mat, const fmpq_mat_t mat1,
    const fmpq_mat_t mat2)
```

Sets `mat` to the sum of `mat1` and `mat2`, assuming that all three matrices have the same dimensions.

```
void fmpq_mat_sub(fmpq_mat_t mat, const fmpq_mat_t mat1,
    const fmpq_mat_t mat2)
```

Sets `mat` to the difference of `mat1` and `mat2`, assuming that all three matrices have the same dimensions.

`void fmpq_mat_neg(fmpq_mat_t rop, const fmpq_mat_t op)`

Sets `rop` to the negative of `op`, assuming that the two matrices have the same dimensions.

```
void fmpq_mat_scalar_mul_fmpz(fmpq_mat_t rop, const
    fmpq_mat_t op, const fmpz_t x)
```

Sets `rop` to `op` multiplied by the integer x, assuming that the two matrices have the same dimensions.

Note that the integer x may not be aliased with any part of the entries of `rop`.

```
void fmpq_mat_scalar_div_fmpz(fmpq_mat_t rop, const
    fmpq_mat_t op, const fmpz_t x)
```

Sets `rop` to `op` divided by the integer x, assuming that the two matrices have the same dimensions and that x is non-zero.

Note that the integer x may not be aliased with any part of the entries of `rop`.

25.6 Input and output

```
void fmpq_mat_print(const fmpq_mat_t mat)
```

Prints the matrix `mat` to standard output.

25.7 Random matrix generation

```
void fmpq_mat_randbits(fmpq_mat_t mat, flint_rand_t state,
    mp_bitcnt_t bits)
```

This is equivalent to applying `fmpq_randbits` to all entries in the matrix.

```
void fmpq_mat_randtest(fmpq_mat_t mat, flint_rand_t state,
    mp_bitcnt_t bits)
```

This is equivalent to applying `fmpq_randtest` to all entries in the matrix.

25.8 Special matrices

```
void fmpq_mat_hilbert_matrix(fmpq_mat_t mat)
```

Sets `mat` to a Hilbert matrix of the given size. That is, the entry at row i and column j is set to $1/(i+j+1)$.

25.9 Basic comparison and properties

```
int fmpq_mat_equal(const fmpq_mat_t mat1, const fmpq_mat_t
    mat2)
```

Returns nonzero if `mat1` and `mat2` have the same shape and all their entries agree, and returns zero otherwise. Assumes the entries in both `mat1` and `mat2` are in canonical form.

```
int fmpq_mat_is_integral(const fmpq_mat_t mat)
```

Returns nonzero if all entries in `mat` are integer-valued, and returns zero otherwise. Assumes that the entries in `mat` are in canonical form.

```
int fmpq_mat_is_zero(const fmpq_mat_t mat)
```

Returns nonzero if all entries in `mat` are zero, and returns zero otherwise.

```
int fmpq_mat_is_empty(const fmpq_mat_t mat)
```

Returns a non-zero value if the number of rows or the number of columns in `mat` is zero, and otherwise returns zero.

```
int fmpq_mat_is_square(const fmpq_mat_t mat)
```

Returns a non-zero value if the number of rows is equal to the number of columns in `mat`, and otherwise returns zero.

25.10 Integer matrix conversion

```
int fmpq_mat_get_fmpz_mat(fmpz_mat_t dest, const fmpq_mat_t
    mat)
```

Sets `dest` to `mat` and returns nonzero if all entries in `mat` are integer-valued. If not all entries in `mat` are integer-valued, sets `dest` to an undefined matrix and returns zero. Assumes that the entries in `mat` are in canonical form.

```
void fmpq_mat_get_fmpz_mat_entrywise(fmpz_mat_t num,
    fmpz_mat_t den, const fmpq_mat_t mat)
```

Sets the integer matrices `num` and `den` respectively to the numerators and denominators of the entries in `mat`.

```
void fmpq_mat_get_fmpz_mat_matwise(fmpz_mat_t num, fmpz_t
    den, const fmpq_mat_t mat)
```

Converts all entries in `mat` to a common denominator, storing the rescaled numerators in `num` and the denominator in `den`. The denominator will be minimal if the entries in `mat` are in canonical form.

```
void fmpq_mat_get_fmpz_mat_rowwise(fmpz_mat_t num, fmpz *
    den, const fmpq_mat_t mat)
```

Clears denominators in `mat` row by row. The rescaled numerators are written to `num`, and the denominator of row `i` is written to position `i` in `den` which can be a preinitialised `fmpz` vector. Alternatively, `NULL` can be passed as the `den` variable, in which case the denominators will not be stored.

```
void fmpq_mat_get_fmpz_mat_rowwise_2(fmpz_mat_t num,
    fmpz_mat_t num2, fmpz * den, const fmpq_mat_t mat, const
    fmpq_mat_t mat2)
```

Clears denominators row by row of both `mat` and `mat2`, writing the respective numerators to `num` and `num2`. This is equivalent to concatenating `mat` and `mat2` horizontally, calling `fmpq_mat_get_fmpz_mat_rowwise`, and extracting the two submatrices in the result.

```
void fmpq_mat_get_fmpz_mat_colwise(fmpz_mat_t num, fmpz *
    den, const fmpq_mat_t mat)
```

Clears denominators in `mat` column by column. The rescaled numerators are written to `num`, and the denominator of column `i` is written to position `i` in `den` which can be a preinitialised `fmpz` vector. Alternatively, `NULL` can be passed as the `den` variable, in which case the denominators will not be stored.

```
void fmpq_mat_set_fmpz_mat(fmpq_mat_t dest, const
    fmpz_mat_t src)
```

Sets `dest` to `src`.

```
void fmpq_mat_set_fmpz_mat_div_fmpz(fmpq_mat_t mat, const
    fmpz_mat_t num, const fmpz_t den)
```

Sets `mat` to the integer matrix `num` divided by the common denominator `den`.

25.11 Modular reduction and rational reconstruction

```
void fmpq_mat_get_fmpz_mat_mod_fmpz(fmpz_mat_t dest, const
    fmpq_mat_t mat, const fmpz_t mod)
```

Sets each entry in `dest` to the corresponding entry in `mat`, reduced modulo `mod`.

```
int fmpq_mat_set_fmpz_mat_mod_fmpz(fmpq_mat_t X, const
    fmpz_mat_t Xmod, const fmpz_t mod)
```

Set `X` to the entrywise rational reconstruction integer matrix `Xmod` modulo `mod`, and returns nonzero if the reconstruction is successful. If rational reconstruction fails for any element, returns zero and sets the entries in `X` to undefined values.

25.12 Matrix multiplication

```
void fmpq_mat_mul_direct(fmpq_mat_t C, const fmpq_mat_t A,
    const fmpq_mat_t B)
```

Sets `C` to the matrix product `AB`, computed naively using rational arithmetic. This is typically very slow and should only be used in circumstances where clearing denominators would consume too much memory.

```
void fmpq_mat_mul_cleared(fmpq_mat_t C, const fmpq_mat_t A,
    const fmpq_mat_t B)
```

Sets `C` to the matrix product `AB`, computed by clearing denominators and multiplying over the integers.

```
void fmpq_mat_mul(fmpq_mat_t C, const fmpq_mat_t A, const
    fmpq_mat_t B)
```

Sets `C` to the matrix product `AB`. This simply calls `fmpq_mat_mul_cleared`.

```
void fmpq_mat_mul_fmpz_mat(fmpq_mat_t C, const fmpq_mat_t
    A, const fmpz_mat_t B)
```

Sets `C` to the matrix product `AB`, with `B` an integer matrix. This function works efficiently by clearing denominators of `A`.

```
void fmpq_mat_mul_r_fmpz_mat(fmpq_mat_t C, const fmpz_mat_t
    A, const fmpq_mat_t B)
```

Sets `C` to the matrix product `AB`, with `A` an integer matrix. This function works efficiently by clearing denominators of `B`.

25.13 Trace

```
void fmpq_mat_trace(fmpq_t trace, const fmpq_mat_t mat)
```

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

25.14 Determinant

```
void fmpq_mat_det(fmpq_t det, const fmpq_mat_t mat)
```

Sets `det` to the determinant of `mat`. In the general case, the determinant is computed by clearing denominators and computing a determinant over the integers. Matrices of size 0, 1 or 2 are handled directly.

25.15 Nonsingular solving

```
int fmpq_mat_solve_fraction_free(fmpq_mat_t X, const
    fmpq_mat_t A, const fmpq_mat_t B)
```

Solves `AX = B` for nonsingular `A` by clearing denominators and solving the rescaled system over the integers using a fraction-free algorithm. This is usually the fastest algorithm for small systems. Returns nonzero if `X` is nonsingular or if the right hand side is empty, and zero otherwise.

```
int fmpq_mat_solve_dixon(fmpq_mat_t X, const fmpq_mat_t A,
    const fmpq_mat_t B)
```

Solves `AX = B` for nonsingular `A` by clearing denominators and solving the rescaled system over the integers using Dixon's algorithm. The rational solution matrix is generated using rational reconstruction. This is usually the fastest algorithm for large systems. Returns nonzero if `X` is nonsingular or if the right hand side is empty, and zero otherwise.

25.16 Inverse

```
int fmpq_mat_inv(fmpq_mat_t B, const fmpq_mat_t A)
```

Sets `B` to the inverse matrix of `A` and returns nonzero. Returns zero if `A` is singular. `A` must be a square matrix.

25.17 Echelon form

```
int fmpq_mat_pivot(slong * perm, fmpq_mat_t mat, slong r,
    slong c)
```

Helper function for row reduction. Returns 1 if the entry of `mat` at row r and column c is nonzero. Otherwise searches for a nonzero entry in the same column among rows $r+1, r+2, \ldots$. If a nonzero entry is found at row s, swaps rows r and s and the corresponding entries in `perm` (unless `NULL`) and returns -1. If no nonzero pivot entry is found, leaves the inputs unchanged and returns 0.

```
slong fmpq_mat_rref_classical(fmpq_mat_t B, const
    fmpq_mat_t A)
```

Sets `B` to the reduced row echelon form of `A` and returns the rank. Performs Gauss-Jordan elimination directly over the rational numbers. This algorithm is usually inefficient and is mainly intended to be used for testing purposes.

```
slong fmpq_mat_rref_fraction_free(fmpq_mat_t B, const
    fmpq_mat_t A)
```

Sets `B` to the reduced row echelon form of `A` and returns the rank. Clears denominators and performs fraction-free Gauss-Jordan elimination using `fmpz_mat` functions.

```
slong fmpq_mat_rref(fmpq_mat_t B, const fmpq_mat_t A)
```

Sets `B` to the reduced row echelon form of `A` and returns the rank. This function automatically chooses between the classical and fraction-free algorithms depending on the size of the matrix.

§26. fmpq_poly

Polynomials over **Q**

26.1 Introduction

The `fmpq_poly_t` data type represents elements of $\mathbf{Q}[x]$. The `fmpq_poly` module provides routines for memory management, basic arithmetic, and conversions from or to other types.

A rational polynomial is stored as the quotient of an integer polynomial and an integer denominator. To be more precise, the coefficient vector of the numerator can be accessed with the function `fmpq_poly_numref()` and the denominator with `fmpq_poly_denref()`. Although one can construct use cases in which a representation as a list of rational coefficients would be beneficial, the choice made here is typically more efficient.

We can obtain a unique representation based on this choice by enforcing, for non-zero polynomials, that the numerator and denominator are coprime and that the denominator is positive. The unique representation of the zero polynomial is chosen as $0/1$.

Similar to the situation in the `fmpz_poly_t` case, an `fmpq_poly_t` object also has a `length` parameter, which denotes the length of the vector of coefficients of the numerator. We say a polynomial is *normalised* either if this length is zero or if the leading coefficient is non-zero.

We say a polynomial is in *canonical* form if it is given in the unique representation discussed above and normalised.

The functions provided in this module roughly fall into two categories:

On the one hand, there are functions mainly provided for the user, whose names do not begin with an underscore. These typically operate on polynomials of type `fmpq_poly_t` in canonical form and, unless specified otherwise, permit aliasing between their input arguments and between their output arguments.

On the other hand, there are versions of these functions whose names are prefixed with a single underscore. These typically operate on polynomials given in the form of a triple of object of types `fmpz *`, `fmpz_t`, and `slong`, containing the numerator, denominator and length, respectively. In general, these functions expect their input to be normalised, i.e. they do not allow zero padding, and to be in lowest terms, and they do not allow their input and output arguments to be aliased.

26.2 Memory management

```
void fmpq_poly_init(fmpq_poly_t poly)
```

Initialises the polynomial for use. The length is set to zero.

```
void fmpq_poly_init2(fmpq_poly_t poly, slong alloc)
```

Initialises the polynomial with space for at least `alloc` coefficients and set the length to zero. The `alloc` coefficients are all set to zero.

```
void fmpq_poly_realloc(fmpq_poly_t poly, slong alloc)
```

Reallocates the given polynomial to have space for `alloc` coefficients. If `alloc` is zero then the polynomial is cleared and then reinitialised. If the current length is greater than `alloc` then `poly` is first truncated to length `alloc`. Note that this might leave the rational polynomial in non-canonical form.

```
void fmpq_poly_fit_length(fmpq_poly_t poly, slong len)
```

If `len` is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least `len` coefficients. No data is lost when calling this function. The function efficiently deals with the case where `fit_length()` is called many times in small increments by at least doubling the number of allocated coefficients when `len` is larger than the number of coefficients currently allocated.

```
void _fmpq_poly_set_length(fmpq_poly_t poly, slong len)
```

Sets the length of the numerator polynomial to `len`, demoting coefficients beyond the new length. Note that this method does not guarantee that the rational polynomial is in canonical form.

```
void fmpq_poly_clear(fmpq_poly_t poly)
```

Clears the given polynomial, releasing any memory used. The polynomial must be reinitialised in order to be used again.

```
void _fmpq_poly_normalise(fmpq_poly_t poly)
```

Sets the length of `poly` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. Note that this function does not guarantee the coprimality of the numerator polynomial and the integer denominator.

```
void _fmpq_poly_canonicalise(fmpz * poly, fmpz_t den, slong
    len)
```

Puts `(poly, den)` of length `len` into canonical form.

It is assumed that the array `poly` contains a non-zero entry in position `len - 1` whenever `len > 0`. Assumes that `den` is non-zero.

```
void fmpq_poly_canonicalise(fmpq_poly_t poly)
```

Puts the polynomial `poly` into canonical form. Firstly, the length is set to the actual length of the numerator polynomial. For non-zero polynomials, it is then ensured that the numerator and denominator are coprime and that the denominator is positive. The canonical form of the zero polynomial is a zero numerator polynomial and a one denominator.

```
int _fmpq_poly_is_canonical(const fmpz * poly, const fmpz_t
    den, slong len)
```

Returns whether the polynomial is in canonical form.

```
int fmpq_poly_is_canonical(const fmpq_poly_t poly)
```

Returns whether the polynomial is in canonical form.

26.3 Polynomial parameters

```
slong fmpq_poly_degree(const fmpq_poly_t poly)
```

Returns the degree of `poly`, which is one less than its length, as a `slong`.

```
slong fmpq_poly_length(const fmpq_poly_t poly)
```

Returns the length of `poly`.

26.4 Accessing the numerator and denominator

```
fmpz * fmpq_poly_numref(fmpq_poly_t poly)
```

Returns a reference to the numerator polynomial as an array.

Note that, because of a delayed initialisation approach, this might be `NULL` for zero polynomials. This situation can be salvaged by calling either `fmpq_poly_fit_length()` or `fmpq_poly_realloc()`.

This function is implemented as a macro returning `(poly)->coeffs`.

```
fmpz_t fmpq_poly_denref(fmpq_poly_t poly)
```

Returns a reference to the denominator as a `fmpz_t`. The integer is guaranteed to be properly initialised.

This function is implemented as a macro returning `(poly)->den`.

26.5 Random testing

The functions `fmpq_poly_randtest_foo()` provide random polynomials suitable for testing. On an integer level, this means that long strings of zeros and ones in the binary representation are favoured as well as the special absolute values 0, 1, `COEFF_MAX`, and `WORD_MAX`. On a polynomial level, the integer numerator has a reasonable chance to have a non-trivial content.

```
void fmpq_poly_randtest(fmpq_poly_t f, flint_rand_t state,
   slong len, mp_bitcnt_t bits)
```

Sets f to a random polynomial with coefficients up to the given length and where each coefficient has up to the given number of bits. The coefficients are signed randomly. One must call `flint_randinit()` before calling this function.

```
void fmpq_poly_randtest_unsigned(fmpq_poly_t f,
   flint_rand_t state, slong len, mp_bitcnt_t bits)
```

Sets f to a random polynomial with coefficients up to the given length and where each coefficient has up to the given number of bits. One must call `flint_randinit()` before calling this function.

```
void fmpq_poly_randtest_not_zero(fmpq_poly_t f,
   flint_rand_t state, slong len, mp_bitcnt_t bits)
```

As for `fmpq_poly_randtest()` except that `len` and `bits` may not be zero and the polynomial generated is guaranteed not to be the zero polynomial. One must call `flint_randinit()` before calling this function.

26.6 Assignment, swap, negation

```
void fmpq_poly_set(fmpq_poly_t poly1, const fmpq_poly_t
    poly2)
```

Sets `poly1` to equal `poly2`.

```
void fmpq_poly_set_si(fmpq_poly_t poly, slong x)
```

Sets `poly` to the integer x.

```
void fmpq_poly_set_ui(fmpq_poly_t poly, ulong x)
```

Sets `poly` to the integer x.

```
void fmpq_poly_set_fmpz(fmpq_poly_t poly, const fmpz_t x)
```

Sets `poly` to the integer x.

```
void fmpq_poly_set_fmpq(fmpq_poly_t poly, const fmpq_t x)
```

Sets `poly` to the rational x, which is assumed to be given in lowest terms.

```
void fmpq_poly_set_mpz(fmpq_poly_t poly, const mpz_t x)
```

Sets `poly` to the integer x.

```
void fmpq_poly_set_mpq(fmpq_poly_t poly, const mpq_t x)
```

Sets `poly` to the rational x, which is assumed to be given in lowest terms.

```
void fmpq_poly_set_fmpz_poly(fmpq_poly_t rop, const
    fmpz_poly_t op)
```

Sets the rational polynomial `rop` to the same value as the integer polynomial `op`.

```
void _fmpq_poly_set_array_mpq(fmpz * poly, fmpz_t den,
    const mpq_t * a, slong n)
```

Sets `(poly, den)` to the polynomial given by the first $n \geq 1$ coefficients in the array a, from lowest degree to highest degree.

The result is only guaranteed to be in lowest terms if all input coefficients are given in lowest terms.

```
void fmpq_poly_set_array_mpq(fmpq_poly_t poly, const mpq_t
    * a, slong n)
```

Sets `poly` to the polynomial with coefficients as given in the array a of length $n \geq 0$, from lowest degree to highest degree.

The result is only guaranteed to be in canonical form if all input coefficients are given in lowest terms.

```
int _fmpq_poly_set_str(fmpz * poly, fmpz_t den, const char
    * str)
```

Sets `(poly, den)` to the polynomial specified by the null-terminated string `str`.

The result is only guaranteed to be in lowest terms if all coefficients in the input string are in lowest terms.

Returns 0 if no error occurred. Otherwise, returns a non-zero value, in which case the resulting value of `(poly, den)` is undefined. If `str` is not null-terminated, calling this method might result in a segmentation fault.

```
int fmpq_poly_set_str(fmpq_poly_t poly, const char * str)
```

Sets `poly` to the polynomial specified by the null-terminated string `str`.

The result is only guaranteed to be in canonical for if all coefficients in the input string are in lowest terms.

Returns 0 if no error occurred. Otherwise, returns a non-zero value, in which case the resulting value of `poly` is undefined. If `str` is not null-terminated, calling this method might result in a segmentation fault.

```
char * fmpq_poly_get_str(const fmpq_poly_t poly)
```

Returns the string representation of `poly`.

```
char * fmpq_poly_get_str_pretty(const fmpq_poly_t poly,
   const char * var)
```

Returns the pretty representation of `poly`, using the null-terminated string `var` not equal to `"\0"` as the variable name.

```
void fmpq_poly_zero(fmpq_poly_t poly)
```

Sets `poly` to zero.

```
void fmpq_poly_one(fmpq_poly_t poly)
```

Sets `poly` to the constant polynomial 1.

```
void fmpq_poly_neg(fmpq_poly_t poly1, const fmpq_poly_t
   poly2)
```

Sets `poly1` to the additive inverse of `poly2`.

```
void fmpq_poly_inv(fmpq_poly_t poly1, const fmpq_poly_t
   poly2)
```

Sets `poly1` to the multiplicative inverse of `poly2` if possible. Otherwise, if `poly2` is not a unit, leaves `poly1` unmodified and calls `abort()`.

```
void fmpq_poly_swap(fmpq_poly_t poly1, fmpq_poly_t poly2)
```

Efficiently swaps the polynomials `poly1` and `poly2`.

```
void fmpq_poly_truncate(fmpq_poly_t poly, slong n)
```

If the current length of `poly` is greater than n, it is truncated to the given length. Discarded coefficients are demoted, but they are not necessarily set to zero.

```
void fmpq_poly_get_slice(fmpq_poly_t rop, const fmpq_poly_t
   op, slong i, slong j)
```

Returns the slice with coefficients from x^i (including) to x^j (excluding).

```
void fmpq_poly_reverse(fmpq_poly_t res, const fmpq_poly_t
   poly, slong n)
```

This function considers the polynomial `poly` to be of length n, notionally truncating and zero padding if required, and reverses the result. Since the function normalises its result `res` may be of length less than n.

26.7 Getting and setting coefficients

```
void fmpq_poly_get_coeff_fmpq(fmpq_t x, const fmpq_poly_t
   poly, slong n)
```

Retrieves the nth coefficient of `poly`, in lowest terms.

```
void fmpq_poly_get_coeff_mpq(mpq_t x, const fmpq_poly_t
   poly, slong n)
```

Retrieves the nth coefficient of `poly`, in lowest terms.

```
void fmpq_poly_set_coeff_si(fmpq_poly_t poly, slong n,
   slong x)
```

Sets the nth coefficient in `poly` to the integer x.

```
void fmpq_poly_set_coeff_ui(fmpq_poly_t poly, slong n,
   ulong x)
```

Sets the nth coefficient in `poly` to the integer x.

```
void fmpq_poly_set_coeff_fmpz(fmpq_poly_t poly, slong n,
   const fmpz_t x)
```

Sets the nth coefficient in `poly` to the integer x.

```
void fmpq_poly_set_coeff_fmpq(fmpq_poly_t poly, slong n,
   const fmpq_t x)
```

Sets the nth coefficient in `poly` to the rational x.

```
void fmpq_poly_set_coeff_mpz(fmpq_poly_t rop, slong n,
   const mpz_t x)
```

Sets the nth coefficient in `poly` to the integer x.

```
void fmpq_poly_set_coeff_mpq(fmpq_poly_t rop, slong n,
   const mpq_t x)
```

Sets the nth coefficient in `poly` to the rational x, which is expected to be provided in lowest terms.

26.8 Comparison

```
int fmpq_poly_equal(const fmpq_poly_t poly1, const
   fmpq_poly_t poly2)
```

Returns 1 if `poly1` is equal to `poly2`, otherwise returns 0.

```
int _fmpq_poly_cmp(const fmpz * lpoly, const fmpz_t lden,
   const fmpz * rpoly, const fmpz_t rden, slong len)
```

Compares two non-zero polynomials, assuming they have the same length `len > 0`.

The polynomials are expected to be provided in canonical form.

```
int fmpq_poly_cmp(const fmpq_poly_t left, const fmpq_poly_t
   right)
```

Compares the two polynomials `left` and `right`.

Compares the two polynomials `left` and `right`, returning -1, 0, or 1 as `left` is less than, equal to, or greater than `right`. The comparison is first done by the degree, and then, in case of a tie, by the individual coefficients from highest to lowest.

```
int fmpq_poly_is_one(const fmpq_poly_t poly)
```

Returns 1 if `poly` is the constant polynomial 1, otherwise returns 0.

```
int fmpq_poly_is_zero(const fmpq_poly_t poly)
```

Returns 1 if `poly` is the zero polynomial, otherwise returns 0.

26.9 Addition and subtraction

```
void _fmpq_poly_add(fmpz * rpoly, fmpz_t rden, const fmpz *
   poly1, const fmpz_t den1, slong len1, const fmpz *
   poly2, const fmpz_t den2, slong len2)
```

Forms the sum `(rpoly, rden)` of `(poly1, den1, len1)` and `(poly2, den2, len2)`, placing the result into canonical form.

Assumes that `rpoly` is an array of length the maximum of `len1` and `len2`. The input operands are assumed to be in canonical form and are also allowed to be of length 0.

`(rpoly, rden)` and `(poly1, den1)` may be aliased, but `(rpoly, rden)` and `(poly2, den2)` may *not* be aliased.

```
void _fmpq_poly_add_can(fmpz * rpoly, fmpz_t rden, const
   fmpz * poly1, const fmpz_t den1, slong len1, const fmpz
   * poly2, const fmpz_t den2, slong len2, int can)
```

As per `_fmpq_poly_add` except that one can specify whether to canonicalise the output or not. This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.

```
void fmpq_poly_add(fmpq_poly_t res, fmpq_poly poly1,
   fmpq_poly poly2)
```

Sets `res` to the sum of `poly1` and `poly2`, using Henrici's algorithm.

```
void fmpq_poly_add_can(fmpq_poly_t res, fmpq_poly poly1,
   fmpq_poly poly2, int can)
```

As per `fmpq_poly_add` except that one can specify whether to canonicalise the output or not. This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.

```
void _fmpq_poly_sub(fmpz * rpoly, fmpz_t rden, const fmpz *
   poly1, const fmpz_t den1, slong len1, const fmpz *
   poly2, const fmpz_t den2, slong len2)
```

Forms the difference `(rpoly, rden)` of `(poly1, den1, len1)` and `(poly2, den2, len2)`, placing the result into canonical form.

Assumes that `rpoly` is an array of length the maximum of `len1` and `len2`. The input operands are assumed to be in canonical form and are also allowed to be of length 0.

`(rpoly, rden)` and `(poly1, den1, len1)` may be aliased, but `(rpoly, rden)` and `(poly2, den2, len2)` may *not* be aliased.

```
void _fmpq_poly_sub_can(fmpz * rpoly, fmpz_t rden, const
   fmpz * poly1, const fmpz_t den1, slong len1, const fmpz
   * poly2, const fmpz_t den2, slong len2, int can)
```

As per `_fmpq_poly_sub` except that one can specify whether to canonicalise the output or not. This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.

```
void fmpq_poly_sub(fmpq_poly_t res, fmpq_poly poly1,
    fmpq_poly poly2)
```

Sets `res` to the difference of `poly1` and `poly2`, using Henrici's algorithm.

```
void fmpq_poly_sub_can(fmpq_poly_t res, fmpq_poly poly1,
    fmpq_poly poly2, int can)
```

As per `_fmpq_poly_sub` except that one can specify whether to canonicalise the output or not. This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.

26.10 Scalar multiplication and division

```
void _fmpq_poly_scalar_mul_si(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, slong len, slong c)
```

Sets `(rpoly, rden, len)` to the product of c of `(poly, den, len)`.

If the input is normalised, then so is the output, provided it is non-zero. If the input is in lowest terms, then so is the output. However, even if neither of these conditions are met, the result will be (mathematically) correct.

Supports exact aliasing between `(rpoly, den)` and `(poly, den)`.

```
void _fmpq_poly_scalar_mul_ui(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, slong len, ulong c)
```

Sets `(rpoly, rden, len)` to the product of c of `(poly, den, len)`.

If the input is normalised, then so is the output, provided it is non-zero. If the input is in lowest terms, then so is the output. However, even if neither of these conditions are met, the result will be (mathematically) correct.

Supports exact aliasing between `(rpoly, den)` and `(poly, den)`.

```
void _fmpq_poly_scalar_mul_fmpz(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, slong len, const
    fmpz_t c)
```

Sets `(rpoly, rden, len)` to the product of c of `(poly, den, len)`.

If the input is normalised, then so is the output, provided it is non-zero. If the input is in lowest terms, then so is the output. However, even if neither of these conditions are met, the result will be (mathematically) correct.

Supports exact aliasing between `(rpoly, den)` and `(poly, den)`.

```
void _fmpq_poly_scalar_mul_fmpq(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, slong len, const
    fmpz_t r, const fmpz_t s)
```

Sets `(rpoly, rden)` to the product of r/s and `(poly, den, len)`, in lowest terms.

Assumes that `(poly, den, len)` and r/s are provided in lowest terms. Assumes that `rpoly` is an array of length `len`. Supports aliasing of `(rpoly, den)` and `(poly, den)`. The `fmpz_t`'s r and s may not be part of `(rpoly, rden)`.

```
void fmpq_poly_scalar_mul_si(fmpq_poly_t rop, const
    fmpq_poly_t op, slong c)
```

Sets `rop` to c times `op`.

```
void fmpq_poly_scalar_mul_ui(fmpq_poly_t rop, const
   fmpq_poly_t op, ulong c)
```

Sets `rop` to c times `op`.

```
void fmpq_poly_scalar_mul_fmpz(fmpq_poly_t rop, const
   fmpq_poly_t op, const fmpz_t c)
```

Sets `rop` to c times `op`. Assumes that the `fmpz_t c` is not part of `rop`.

```
void fmpq_poly_scalar_mul_fmpq(fmpq_poly_t rop, const
   fmpq_poly_t op, const mpq_t c)
```

Sets `rop` to c times `op`.

```
void fmpq_poly_scalar_mul_mpz(fmpq_poly_t rop, const
   fmpq_poly_t op, const mpz_t c)
```

Sets `rop` to c times `op`.

```
void fmpq_poly_scalar_mul_mpq(fmpq_poly_t rop, const
   fmpq_poly_t op, const fmpq_t c)
```

Sets `rop` to c times `op`.

```
void _fmpq_poly_scalar_div_fmpz(fmpz * rpoly, fmpz_t rden,
   const fmpz * poly, const fmpz_t den, slong len, const
   fmpz_t c)
```

Sets `(rpoly, rden, len)` to `(poly, den, len)` divided by c, in lowest terms.

Assumes that `len` is positive. Assumes that c is non-zero. Supports aliasing between `(rpoly, rden)` and `(poly, den)`. Assumes that c is not part of `(rpoly, rden)`.

```
void _fmpq_poly_scalar_div_si(fmpz * rpoly, fmpz_t rden,
   const fmpz * poly, const fmpz_t den, slong len, slong c)
```

Sets `(rpoly, rden, len)` to `(poly, den, len)` divided by c, in lowest terms.

Assumes that `len` is positive. Assumes that c is non-zero. Supports aliasing between `(rpoly, rden)` and `(poly, den)`.

```
void _fmpq_poly_scalar_div_ui(fmpz * rpoly, fmpz_t rden,
   const fmpz * poly, const fmpz_t den, slong len, ulong c)
```

Sets `(rpoly, rden, len)` to `(poly, den, len)` divided by c, in lowest terms.

Assumes that `len` is positive. Assumes that c is non-zero. Supports aliasing between `(rpoly, rden)` and `(poly, den)`.

```
void _fmpq_poly_scalar_div_fmpq(fmpz * rpoly, fmpz_t rden,
   const fmpz * poly, const fmpz_t den, slong len, const
   fmpz_t r, const fmpz_t s)
```

Sets `(rpoly, rden, len)` to `(poly, den, len)` divided by r/s, in lowest terms.

Assumes that `len` is positive. Assumes that r/s is non-zero and in lowest terms. Supports aliasing between `(rpoly, rden)` and `(poly, den)`. The `fmpz_t`'s r and s may not be part of `(rpoly, poly)`.

```
void fmpq_poly_scalar_div_si(fmpq_poly_t rop, const
   fmpq_poly_t op, slong c)
```

```
void fmpq_poly_scalar_div_ui(fmpq_poly_t rop, const
   fmpq_poly_t op, ulong c);

void fmpq_poly_scalar_div_fmpz(fmpq_poly_t rop, const
   fmpq_poly_t op, const fmpz_t c);

void fmpq_poly_scalar_div_fmpq(fmpq_poly_t rop, const
   fmpq_poly_t op, const fmpq_t c);

void fmpq_poly_scalar_div_mpz(fmpq_poly_t rop, const
   fmpq_poly_t op, const mpz_t c);

void fmpq_poly_scalar_div_mpq(fmpq_poly_t rop, const
   fmpq_poly_t op, const mpq_t c);
```

26.11 Multiplication

```
void _fmpq_poly_mul(fmpz * rpoly, fmpz_t rden, const fmpz *
   poly1, const fmpz_t den1, slong len1, const fmpz *
   poly2, const fmpz_t den2, slong len2)
```

Sets `(rpoly, rden, len1 + len2 - 1)` to the product of `(poly1, den1, len1)` and `(poly2, den2, len2)`. If the input is provided in canonical form, then so is the output.

Assumes `len1 >= len2 > 0`. Allows zero-padding in the input. Does not allow aliasing between the inputs and outputs.

```
void fmpq_poly_mul(fmpq_poly_t res, const fmpq_poly_t
   poly1, const fmpq_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _fmpq_poly_mullow(fmpz * rpoly, fmpz_t rden, const
   fmpz * poly1, const fmpz_t den1, slong len1, const fmpz
   * poly2, const fmpz_t den2, slong len2, slong n)
```

Sets `(rpoly, rden, n)` to the low n coefficients of `(poly1, den1)` and `(poly2, den2)`. The output is not guaranteed to be in canonical form.

Assumes `len1 >= len2 > 0` and `0 < n <= len1 + len2 - 1`. Allows for zero-padding in the inputs. Does not allow aliasing between the inputs and outputs.

```
void fmpq_poly_mullow(fmpq_poly_t res, const fmpq_poly_t
   poly1, const fmpq_poly_t poly2, slong n)
```

Sets `res` to the product of `poly1` and `poly2`, truncated to length n.

```
void fmpq_poly_addmul(fmpq_poly_t rop, const fmpq_poly_t
   op1, const fmpq_poly_t op2)
```

Adds the product of `op1` and `op2` to `rop`.

```
void fmpq_poly_submul(fmpq_poly_t rop, const fmpq_poly_t
   op1, const fmpq_poly_t op2)
```

Subtracts the product of `op1` and `op2` from `rop`.

26.12 Powering

```
void _fmpq_poly_pow(fmpz * rpoly, fmpz_t rden, const fmpz *
   poly, const fmpz_t den, slong len, ulong e)
```

Sets `(rpoly, rden)` to `(poly, den)^e`, assuming `e, len > 0`. Assumes that `rpoly` is an array of length at least `e * (len - 1)+ 1`. Supports aliasing of `(rpoly, den)` and `(poly, den)`.

```
void fmpq_poly_pow(fmpq_poly_t res, const fmpq_poly_t poly,
   ulong e)
```

Sets `res` to `pow^e`, where the only special case 0^0 is defined as 1.

26.13 Shifting

```
void fmpz_poly_shift_left(fmpz_poly_t res, const
   fmpz_poly_t poly, slong n)
```

Set `res` to `poly` shifted left by n coefficients. Zero coefficients are inserted.

```
void fmpz_poly_shift_right(fmpz_poly_t res, const
   fmpz_poly_t poly, slong n)
```

Set `res` to `poly` shifted right by n coefficients. If n is equal to or greater than the current length of `poly`, `res` is set to the zero polynomial.

26.14 Euclidean division

```
void _fmpq_poly_divrem(fmpz * Q, fmpz_t q, fmpz * R, fmpz_t
   r, const fmpz * A, const fmpz_t a, slong lenA, const
   fmpz * B, const fmpz_t b, slong lenB, const
   fmpz_preinvn_t inv)
```

Finds the quotient `(Q, q)` and remainder `(R, r)` of the Euclidean division of `(A, a)` by `(B, b)`.

Assumes that `lenA >= lenB > 0`. Assumes that R has space for `lenA` coefficients, although only the bottom `lenB - 1` will carry meaningful data on exit. Supports no aliasing between the two outputs, or between the inputs and the outputs.

An optional precomputed inverse of the leading coefficient of B from `fmpz_preinvn_init` can be supplied. Otherwise `inv` should be `NULL`.

```
void fmpq_poly_divrem(fmpq_poly_t Q, fmpq_poly_t R, const
   fmpq_poly_t poly1, const fmpq_poly_t poly2)
```

Finds the quotient Q and remainder R of the Euclidean division of `poly1` by `poly2`.

```
void _fmpq_poly_div(fmpz * Q, fmpz_t q, const fmpz * A,
   const fmpz_t a, slong lenA, const fmpz * B, const fmpz_t
   b, slong lenB, const fmpz_preinvn_t inv)
```

Finds the quotient `(Q, q)` of the Euclidean division of `(A, a)` by `(B, b)`.

Assumes that `lenA >= lenB > 0`. Supports no aliasing between the inputs and the outputs.

An optional precomputed inverse of the leading coefficient of B from `fmpz_preinvn_init` can be supplied. Otherwise `inv` should be `NULL`.

```
void fmpq_poly_div(fmpq_poly_t Q, const fmpq_poly_t poly1,
   const fmpq_poly_t poly2)
```

Finds the quotient Q and remainder R of the Euclidean division of `poly1` by `poly2`.

```
void _fmpq_poly_rem(fmpz * R, fmpz_t r, const fmpz * A,
   const fmpz_t a, slong lenA, const fmpz * B, const fmpz_t
   b, slong lenB, const fmpz_preinvn_t inv)
```

Finds the remainder `(R, r)` of the Euclidean division of `(A, a)` by `(B, b)`.

Assumes that `lenA >= lenB > 0`. Supports no aliasing between the inputs and the outputs.

An optional precomputed inverse of the leading coefficient of B from `fmpz_preinvn_init` can be supplied. Otherwise `inv` should be `NULL`.

```
void fmpq_poly_rem(fmpq_poly_t R, const fmpq_poly_t poly1,
   const fmpq_poly_t poly2)
```

Finds the remainder R of the Euclidean division of `poly1` by `poly2`.

26.15 Euclidean division

```
fmpq_poly_struct * _fmpq_poly_powers_precompute(const fmpz
   * B, const fmpz_t denB, slong len)
```

Computes `2*len - 1` powers of x modulo the polynomial B of the given length. This is used as a kind of precomputed inverse in the remainder routine below.

```
void fmpq_poly_powers_precompute(fmpq_poly_powers_precomp_t
   pinv, fmpq_poly_t poly)
```

Computes `2*len - 1` powers of x modulo the polynomial B of the given length. This is used as a kind of precomputed inverse in the remainder routine below.

```
void _fmpq_poly_powers_clear(fmpq_poly_struct * powers,
   slong len)
```

Clean up resources used by precomputed powers which have been computed by `_fmpq_poly_powers_precompute`.

```
void fmpq_poly_powers_clear(fmpq_poly_powers_precomp_t pinv)
```

Clean up resources used by precomputed powers which have been computed by `fmpq_poly_powers_precompute`.

```
void _fmpq_poly_rem_powers_precomp(fmpz * A, fmpz_t denA,
   slong m, const fmpz * B, const fmpz_t denB, slong n,
   const fmpq_poly_struct * const powers)
```

Set A to the remainder of A divide B given precomputed powers mod B provided by `_fmpq_poly_powers_precompute`. No aliasing is allowed.

This function is only faster if $m \leq 2 * n - 1$.

The output of this function is *not* canonicalised.

```
void fmpq_poly_rem_powers_precomp(fmpq_poly_t R, const
   fmpq_poly_t A, const fmpq_poly_t B, const
   fmpq_poly_powers_precomp_t B_inv)
```

Set R to the remainder of A divide B given precomputed powers mod B provided by `fmpq_poly_powers_precompute`.

This function is only faster if `A->length <= 2*B->length - 1`.

The output of this function is *not* canonicalised.

26.16 Power series division

```
void _fmpq_poly_inv_series_newton(fmpz * rpoly, fmpz_t
   rden, const fmpz * poly, const fmpz_t den, slong n)
```

Computes the first n terms of the inverse power series of `poly` using Newton iteration.

The result is produced in canonical form.

Assumes that $n \geq 1$, that `poly` has length at least n and non-zero constant term. Does not support aliasing.

```
void fmpq_poly_inv_series_newton(fmpq_poly_t res, const
   fmpq_poly_t poly, slong n)
```

Computes the first n terms of the inverse power series of `poly` using Newton iteration, assuming that `poly` has non-zero constant term and $n \geq 1$.

```
void _fmpq_poly_inv_series(fmpz * rpoly, fmpz_t rden, const
   fmpz * poly, const fmpz_t den, slong n)
```

Computes the first n terms of the inverse power series of `poly`.

Assumes that $n \geq 1$, that `poly` has length at least n and non-zero constant term. Does not support aliasing.

```
void fmpq_poly_inv_series(fmpq_poly_t res, const
   fmpq_poly_t poly, slong n)
```

Computes the first n terms of the inverse power series of `poly`, assuming that `poly` has non-zero constant term and $n \geq 1$.

```
void _fmpq_poly_div_series(fmpz * Q, fmpz_t denQ, const
   fmpz * A, const fmpz_t denA, const fmpz * B, const
   fmpz_t denB, slong n)
```

Divides `(A, denA, n)` by `(B, denB, n)` as power series over $\mathbf{Q}$, assuming B has non-zero constant term and $n \geq 1$.

Supports no aliasing other than that of `(Q, denQ, n)` and `(B, denB, n)`.

This function does not ensure that the numerator and denominator are coprime on exit.

```
void fmpq_poly_div_series(fmpq_poly_t Q, const fmpq_poly_t
   A, const fmpq_poly_t B, slong n)
```

Performs power series division in $\mathbf{Q}[[x]]/(x^n)$. The function considers the polynomials A and B as power series of length n starting with the constant terms. The function assumes that B has non-zero constant term and $n \geq 1$.

26.17 Greatest common divisor

```
void _fmpq_poly_gcd(fmpz *G, fmpz_t denG, const fmpz *A,
   slong lenA, const fmpz *B, slong lenB)
```

Computes the monic greatest common divisor G of A and B.

Assumes that G has space for $\operatorname{len}(B)$ coefficients, where $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$.

Aliasing between the output and input arguments is not supported.

Does not support zero-padding.

```
void fmpq_poly_gcd(fmpq_poly_t G, const fmpq_poly_t A,
    const fmpq_poly_t B)
```

Computes the monic greatest common divisor G of A and B.

In the the special case when $A = B = 0$, sets $G = 0$.

```
void _fmpq_poly_xgcd(fmpz *G, fmpz_t denG, fmpz *S, fmpz_t
    denS, fmpz *T, fmpz_t denT, const fmpz *A, const fmpz_t
    denA, slong lenA, const fmpz *B, const fmpz_t denB,
    slong lenB)
```

Computes polynomials G, S, and T such that $G = \gcd(A, B) = SA + TB$, where G is the monic greatest common divisor of A and B.

Assumes that G, S, and T have space for $\operatorname{len}(B)$, $\operatorname{len}(B)$, and $\operatorname{len}(A)$ coefficients, respectively, where it is also assumed that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$.

Does not support zero padding of the input arguments.

```
void fmpq_poly_xgcd(fmpq_poly_t G, fmpz_poly_t S,
    fmpz_poly_t T, const fmpq_poly_t A, const fmpq_poly_t B)
```

Computes polynomials G, S, and T such that $G = \gcd(A, B) = SA + TB$, where G is the monic greatest common divisor of A and B.

Corner cases are handled as follows. If $A = B = 0$, returns $G = S = T = 0$. If $A \neq 0$, $B = 0$, returns the suitable scalar multiple of $G = A$, $S = 1$, and $T = 0$. The case when $A = 0$, $B \neq 0$ is handled similarly.

```
void _fmpq_poly_lcm(fmpz *L, fmpz_t denL, const fmpz *A,
    slong lenA, const fmpz *B, slong lenB)
```

Computes the monic least common multiple L of A and B.

Assumes that L has space for $\operatorname{len}(A)+\operatorname{len}(B)-1$ coefficients, where $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$.

Aliasing between the output and input arguments is not supported.

Does not support zero-padding.

```
void fmpq_poly_lcm(fmpq_poly_t L, const fmpq_poly_t A,
    const fmpq_poly_t B)
```

Computes the monic least common multiple L of A and B.

In the special case when $A = B = 0$, sets $L = 0$.

```
void _fmpq_poly_resultant(fmpz_t rnum, fmpz_t rden, const
    fmpz *poly1, const fmpz_t den1, slong len1, const fmpz
    *poly2, const fmpz_t den2, slong len2)
```

Sets `(rnum, rden)` to the resultant of the two input polynomials.

Assumes that `len1 >= len2 > 0`. Does not support zero-padding of the input polynomials. Does not support aliasing of the input and output arguments.

```
void fmpq_poly_resultant(fmpq_t r, const fmpq_poly_t f,
    const fmpq_poly_t g)
```

Returns the resultant of f and g.

Enumerating the roots of f and g over $\bar{\mathbf{Q}}$ as $r_1, \ldots, r_m$ and $s_1, \ldots, s_n$, respectively, and letting x and y denote the leading coefficients, the resultant is defined as

$$x^{\deg(f)} y^{\deg(g)} \prod_{1 \leq i,j \leq n} (r_i - s_j).$$

We handle special cases as follows: if one of the polynomials is zero, the resultant is zero. Note that otherwise if one of the polynomials is constant, the last term in the above expression is the empty product.

26.18 Derivative and integral

```
void _fmpq_poly_derivative(fmpz * rpoly, fmpz_t rden, const
    fmpz * poly, const fmpz_t den, slong len)
```

Sets `(rpoly, rden, len - 1)` to the derivative of `(poly, den, len)`. Does nothing if `len <= 1`. Supports aliasing between the two polynomials.

```
void fmpq_poly_derivative(fmpq_poly_t res, const
    fmpq_poly_t poly)
```

Sets `res` to the derivative of `poly`.

```
void _fmpq_poly_integral(fmpz * rpoly, fmpz_t rden, const
    fmpz * poly, const fmpz_t den, slong len)
```

Sets `(rpoly, rden, len)` to the integral of `(poly, den, len - 1)`. Assumes `len >= 0`. Supports aliasing between the two polynomials.

```
void fmpq_poly_integral(fmpq_poly_t res, const fmpq_poly_t
    poly)
```

Sets `res` to the integral of `poly`. The constant term is set to zero. In particular, the integral of the zero polynomial is the zero polynomial.

26.19 Square roots

```
void _fmpq_poly_sqrt_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the square root of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 1. Does not support aliasing between the input and output polynomials.

```
void fmpq_poly_sqrt_series(fmpq_poly_t res, const
    fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the square root of `f` to order `n > 1`. Requires `f` to have constant term 1.

```
void _fmpq_poly_invsqrt_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the inverse square root of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 1. Does not support aliasing between the input and output polynomials.

```
void fmpq_poly_invsqrt_series(fmpq_poly_t res, const
    fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the inverse square root of `f` to order `n > 0`. Requires `f` to have constant term 1.

26.20 Transcendental functions

```
void _fmpq_poly_log_series(fmpz * g, fmpz_t gden, const
   fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the logarithm of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 1. Supports aliasing between the input and output polynomials.

```
void fmpq_poly_log_series(fmpq_poly_t res, const
   fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the logarithm of `f` to order `n > 0`. Requires `f` to have constant term 1.

```
void _fmpq_poly_exp_series(fmpz * g, fmpz_t gden, const
   fmpz * h, const fmpz_t hden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the exponential function of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Does not support aliasing between the input and output polynomials.

```
void fmpq_poly_exp_series(fmpq_poly_t res, const
   fmpq_poly_t h, slong n)
```

Sets `res` to the series expansion of the exponential function of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_atan_series(fmpz * g, fmpz_t gden, const
   fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the inverse tangent of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Supports aliasing between the input and output polynomials.

```
void fmpq_poly_atan_series(fmpq_poly_t res, const
   fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the inverse tangent of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_atanh_series(fmpz * g, fmpz_t gden, const
   fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the inverse hyperbolic tangent of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Supports aliasing between the input and output polynomials.

```
void fmpq_poly_atanh_series(fmpq_poly_t res, const
   fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the inverse hyperbolic tangent of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_asin_series(fmpz * g, fmpz_t gden, const
   fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the inverse sine of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Supports aliasing between the input and output polynomials.

```
void fmpq_poly_asin_series(fmpq_poly_t res, const
   fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the inverse sine of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_asinh_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the inverse hyperbolic sine of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Supports aliasing between the input and output polynomials.

```
void fmpq_poly_asinh_series(fmpq_poly_t res, const
    fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the inverse hyperbolic sine of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_tan_series(fmpz * g, fmpz_t gden, const
    fmpz * h, const fmpz_t hden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the tangent function of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Does not support aliasing between the input and output polynomials.

```
void fmpq_poly_tan_series(fmpq_poly_t res, const
    fmpq_poly_t h, slong n)
```

Sets `res` to the series expansion of the tangent function of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_sin_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the sine of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Supports aliasing between the input and output polynomials.

```
void fmpq_poly_sin_series(fmpq_poly_t res, const
    fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the sine of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_cos_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the cosine of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Supports aliasing between the input and output polynomials.

```
void fmpq_poly_cos_series(fmpq_poly_t res, const
    fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the cosine of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_sinh_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the hyperbolic sine of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Does not support aliasing between the input and output polynomials.

```
void fmpq_poly_sinh_series(fmpq_poly_t res, const
   fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the hyperbolic sine of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_cosh_series(fmpz * g, fmpz_t gden, const
   fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the hyperbolic cosine of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Does not support aliasing between the input and output polynomials.

```
void fmpq_poly_cosh_series(fmpq_poly_t res, const
   fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the hyperbolic cosine of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_tanh_series(fmpz * g, fmpz_t gden, const
   fmpz * f, const fmpz_t fden, slong n)
```

Sets `(g, gden, n)` to the series expansion of the hyperbolic tangent of `(f, fden, n)`. Assumes `n > 0` and that `(f, fden, n)` has constant term 0. Does not support aliasing between the input and output polynomials.

```
void fmpq_poly_tanh_series(fmpq_poly_t res, const
   fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the hyperbolic tangent of `f` to order `n > 0`. Requires `f` to have constant term 0.

26.21 Evaluation

```
void _fmpq_poly_evaluate_fmpz(fmpz_t rnum, fmpz_t rden,
   const fmpz * poly, const fmpz_t den, slong len, const
   fmpz_t a)
```

Evaluates the polynomial `(poly, den, len)` at the integer a and sets `(rnum, rden)` to the result in lowest terms.

```
void fmpq_poly_evaluate_fmpz(fmpq_t res, const fmpq_poly_t
   poly, const fmpz_t a)
```

Evaluates the polynomial `poly` at the integer a and sets `res` to the result.

```
void _fmpq_poly_evaluate_fmpq(fmpz_t rnum, fmpz_t rden,
   const fmpz * poly, const fmpz_t den, slong len, const
   fmpz_t anum, const fmpz_t aden)
```

Evaluates the polynomial `(poly, den, len)` at the rational `(anum, aden)` and sets `(rnum, rden)` to the result in lowest terms. Aliasing between `(rnum, rden)` and `(anum, aden)` is not supported.

```
void fmpq_poly_evaluate_fmpq(fmpq_t res, const fmpq_poly_t
   poly, const fmpq_t a)
```

Evaluates the polynomial `poly` at the rational a and sets `res` to the result.

```
void fmpq_poly_evaluate_mpz(mpq_t res, const fmpq_poly_t
   poly, const mpz_t a)
```

Evaluates the polynomial `poly` at the integer a of type `mpz` and sets `res` to the result.

```
void fmpq_poly_evaluate_mpq(mpq_t res, const fmpq_poly_t
    poly, const mpq_t a)
```

Evaluates the polynomial `poly` at the rational a of type `mpq` and sets `res` to the result.

26.22 Interpolation

```
void _fmpq_poly_interpolate_fmpz_vec(fmpz * poly, fmpz_t
    den, const fmpz * xs, const fmpz * ys, slong n)
```

Sets `poly` / `den` to the unique interpolating polynomial of degree at most $n-1$ satisfying $f(x_i) = y_i$ for every pair x_i, y_i in `xs` and `ys`.

The vector `poly` must have room for `n+1` coefficients, even if the interpolating polynomial is shorter. Aliasing of `poly` or `den` with any other argument is not allowed.

It is assumed that the x values are distinct.

This function uses a simple $O(n^2)$ implementation of Lagrange interpolation, clearing denominators to avoid working with fractions. It is currently not designed to be efficient for large n.

```
fmpq_poly_interpolate_fmpz_vec(fmpq_poly_t poly, const fmpz
    * xs, const fmpz * ys, slong n)
```

Sets `poly` to the unique interpolating polynomial of degree at most $n-1$ satisfying $f(x_i) = y_i$ for every pair x_i, y_i in `xs` and `ys`. It is assumed that the x values are distinct.

26.23 Composition

```
void _fmpq_poly_compose(fmpz * res, fmpz_t den, const fmpz
    * poly1, const fmpz_t den1, slong len1, const fmpz *
    poly2, const fmpz_t den2, slong len2)
```

Sets `(res, den)` to the composition of `(poly1, den1, len1)` and `(poly2, den2, len2)`, assuming `len1, len2 > 0`.

Assumes that `res` has space for `(len1 - 1)* (len2 - 1)+ 1` coefficients. Does not support aliasing.

```
void fmpq_poly_compose(fmpq_poly_t res, const fmpq_poly_t
    poly1, const fmpq_poly_t poly2)
```

Sets `res` to the composition of `poly1` and `poly2`.

```
void _fmpq_poly_rescale(fmpz * res, fmpz_t denr, const fmpz
    * poly, const fmpz_t den, slong len, const fmpz_t anum,
    const fmpz_t aden)
```

Sets `(res, denr, len)` to `(poly, den, len)` with the indeterminate rescaled by `(anum, aden)`.

Assumes that `len > 0` and that `(anum, aden)` is non-zero and in lowest terms. Supports aliasing between `(res, denr, len)` and `(poly, den, len)`.

```
void fmpz_poly_rescale(fmpq_poly_t res, const fmpq_poly_t
    poly, const fmpq_t a)
```

Sets `res` to `poly` with the indeterminate rescaled by a.

26.24 Power series composition

```
void _fmpq_poly_compose_series_horner(fmpz * res, fmpz_t
    den, const fmpz * poly1, const fmpz_t den1, slong len1,
    const fmpz * poly2, const fmpz_t den2, slong len2, slong
    n)
```

Sets `(res, den, n)` to the composition of `(poly1, den1, len1)` and `(poly2, den2, len2)` modulo x^n, where the constant term of `poly2` is required to be zero.

Assumes that `len1, len2, n > 0`, that `len1, len2 <= n`, that `(len1-1)* (len2-1)+ 1 <= n`, and that `res` has space for `n` coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses the Horner scheme. The default `fmpz_poly` composition algorithm is automatically used when the composition can be performed over the integers.

```
void fmpq_poly_compose_series_horner(fmpq_poly_t res, const
    fmpq_poly_t poly1, const fmpq_poly_t poly2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

This implementation uses the Horner scheme. The default `fmpz_poly` composition algorithm is automatically used when the composition can be performed over the integers.

```
void _fmpq_poly_compose_series_brent_kung(fmpz * res,
    fmpz_t den, const fmpz * poly1, const fmpz_t den1, slong
    len1, const fmpz * poly2, const fmpz_t den2, slong len2,
    slong n)
```

Sets `(res, den, n)` to the composition of `(poly1, den1, len1)` and `(poly2, den2, len2)` modulo x^n, where the constant term of `poly2` is required to be zero.

Assumes that `len1, len2, n > 0`, that `len1, len2 <= n`, that `(len1-1)* (len2-1)+ 1 <= n`, and that `res` has space for `n` coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses Brent-Kung algorithm 2.1 [6]. The default `fmpz_poly` composition algorithm is automatically used when the composition can be performed over the integers.

```
void fmpq_poly_compose_series_brent_kung(fmpq_poly_t res,
    const fmpq_poly_t poly1, const fmpq_poly_t poly2, slong
    n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

This implementation uses Brent-Kung algorithm 2.1 [6]. The default `fmpz_poly` composition algorithm is automatically used when the composition can be performed over the integers.

```
void _fmpq_poly_compose_series(fmpz * res, fmpz_t den,
    const fmpz * poly1, const fmpz_t den1, slong len1, const
    fmpz * poly2, const fmpz_t den2, slong len2, slong n)
```

Sets `(res, den, n)` to the composition of `(poly1, den1, len1)` and `(poly2, den2, len2)` modulo x^n, where the constant term of `poly2` is required to be zero.

Assumes that `len1, len2, n > 0`, that `len1, len2 <= n`, that `(len1-1)* (len2-1)+ 1 <= n`, and that `res` has space for `n` coefficients. Does not support aliasing between any of the inputs and the output.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs. The default `fmpz_poly` composition algorithm is automatically used when the composition can be performed over the integers.

```
void fmpq_poly_compose_series(fmpq_poly_t res, const
   fmpq_poly_t poly1, const fmpq_poly_t poly2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs. The default `fmpz_poly` composition algorithm is automatically used when the composition can be performed over the integers.

26.25 Power series reversion

```
void _fmpq_poly_revert_series_lagrange(fmpz * res, fmpz_t
   den, const fmpz * poly1, const fmpz_t den1, slong n)
```

Sets `(res, den)` to the power series reversion of `(poly1, den1)` modulo x^n, where the input has length n (possibly being zero-padded).

The constant term of `poly2` is required to be zero and the linear term is required to be nonzero. Assumes that $n > 0$. Does not support aliasing between any of the inputs and the output.

This implementation uses the Lagrange inversion formula. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

```
void fmpq_poly_revert_series_lagrange(fmpq_poly_t res,
   const fmpq_poly_t poly, slong n)
```

Sets `res` to the power series reversion of `poly1` modulo x^n. The constant term of `poly2` is required to be zero and the linear term is required to be nonzero.

This implementation uses the Lagrange inversion formula. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

```
void _fmpq_poly_revert_series_lagrange_fast(fmpz * res,
   fmpz_t den, const fmpz * poly1, const fmpz_t den1, slong
   n)
```

Sets `(res, den)` to the power series reversion of `(poly1, den1)` modulo x^n, where the input has length n (possibly being zero-padded).

The constant term of `poly2` is required to be zero and the linear term is required to be nonzero. Assumes that $n > 0$. Does not support aliasing between any of the inputs and the output.

This implementation uses a reduced-complexity implementation of the Lagrange inversion formula. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

```
void fmpq_poly_revert_series_lagrange_fast(fmpq_poly_t res,
   const fmpq_poly_t poly, slong n)
```

Sets `res` to the power series reversion of `poly1` modulo x^n. The constant term of `poly2` is required to be zero and the linear term is required to be nonzero.

This implementation uses a reduced-complexity implementation of the Lagrange inversion formula. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

```
void _fmpq_poly_revert_series_newton(fmpz * res, fmpz_t
   den, const fmpz * poly1, const fmpz_t den1, slong n)
```

Sets `(res, den)` to the power series reversion of `(poly1, den1)` modulo x^n, where the input has length n (possibly being zero-padded).

The constant term of `poly2` is required to be zero and the linear term is required to be nonzero. Assumes that $n > 0$. Does not support aliasing between any of the inputs and the output.

This implementation uses Newton iteration. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

```
void fmpq_poly_revert_series_newton(fmpq_poly_t res, const
   fmpq_poly_t poly, slong n)
```

Sets `res` to the power series reversion of `poly1` modulo x^n. The constant term of `poly2` is required to be zero and the linear term is required to be nonzero.

This implementation uses Newton iteration. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

```
void _fmpq_poly_revert_series(fmpz * res, fmpz_t den, const
   fmpz * poly1, const fmpz_t den1, slong n)
```

Sets `(res, den)` to the power series reversion of `(poly1, den1)` modulo x^n, where the input has length n (possibly being zero-padded).

The constant term of `poly2` is required to be zero and the linear term is required to be nonzero. Assumes that $n > 0$. Does not support aliasing between any of the inputs and the output.

This implementation defaults to using Newton iteration. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

```
void fmpq_poly_revert_series(fmpq_poly_t res, const
   fmpq_poly_t poly, slong n)
```

Sets `res` to the power series reversion of `poly1` modulo x^n. The constant term of `poly2` is required to be zero and the linear term is required to be nonzero.

This implementation defaults to using Newton iteration. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

26.26 Gaussian content

```
void _fmpq_poly_content(fmpq_t res, const fmpz * poly,
   const fmpz_t den, slong len)
```

Sets `res` to the content of `(poly, den, len)`. If `len == 0`, sets `res` to zero.

```
void fmpq_poly_content(fmpq_t res, const fmpq_poly_t poly)
```

Sets `res` to the content of `poly`. The content of the zero polynomial is defined to be zero.

```
void _fmpq_poly_primitive_part(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, slong len)
```

Sets `(rpoly, rden, len)` to the primitive part, with non-negative leading coefficient, of `(poly, den, len)`. Assumes that `len > 0`. Supports aliasing between the two polynomials.

```
void fmpq_poly_primitive_part(fmpq_poly_t res, const
    fmpq_poly_t poly)
```

Sets `res` to the primitive part, with non-negative leading coefficient, of `poly`.

```
int _fmpq_poly_is_monic(const fmpz * poly, const fmpz_t
    den, slong len)
```

Returns whether the polynomial `(poly, den, len)` is monic. The zero polynomial is not monic by definition.

```
int fmpq_poly_is_monic(const fmpq_poly_t poly)
```

Returns whether the polynomial `poly` is monic. The zero polynomial is not monic by definition.

```
void _fmpq_poly_make_monic(fmpz * rpoly, fmpz_t rden, const
    fmpz * poly, const fmpz_t den, slong len)
```

Sets `(rpoly, rden, len)` to the monic scalar multiple of `(poly, den, len)`. Assumes that `len > 0`. Supports aliasing between the two polynomials.

```
void fmpq_poly_make_monic(fmpq_poly_t res, const
    fmpq_poly_t poly)
```

Sets `res` to the monic scalar multiple of `poly` whenever `poly` is non-zero. If `poly` is the zero polynomial, sets `res` to zero.

26.27 Square-free

```
int fmpq_poly_is_squarefree(const fmpq_poly_t poly)
```

Returns whether the polynomial `poly` is square-free. A non-zero polynomial is defined to be square-free if it has no non-unit square factors. We also define the zero polynomial to be square-free.

26.28 Input and output

```
int _fmpq_poly_print(const fmpz * poly, const fmpz_t den,
    slong len)
```

Prints the polynomial `(poly, den, len)` to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpq_poly_print(const fmpq_poly_t poly)
```

Prints the polynomial to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fmpq_poly_print_pretty(const fmpz *poly, const fmpz_t
    den, slong len, const char * x)

int fmpq_poly_print_pretty(const fmpq_poly_t poly, const
    char * var)
```

Prints the pretty representation of `poly` to `stdout`, using the null-terminated string `var` not equal to `"\0"` as the variable name.

In the current implementation always returns 1.

```
int _fmpq_poly_fprint(FILE * file, const fmpz * poly, const
    fmpz_t den, slong len)
```

Prints the polynomial `(poly, den, len)` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpq_poly_fprint(FILE * file, const fmpq_poly_t poly)
```

Prints the polynomial to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fmpq_poly_fprint_pretty(FILE * file, const fmpz *poly,
    const fmpz_t den, slong len, const char * x)

int fmpq_poly_print_pretty(const fmpq_poly_t poly, const
    char * var)
```

Prints the pretty representation of `poly` to `stdout`, using the null-terminated string `var` not equal to `"\0"` as the variable name.

In the current implementation, always returns 1.

```
int fmpq_poly_read(fmpq_poly_t poly)
```

Reads a polynomial from `stdin`, storing the result in `poly`.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

```
int fmpq_poly_fread(FILE * file, fmpq_poly_t poly)
```

Reads a polynomial from the stream `file`, storing the result in `poly`.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

§27. fmpz_poly_q

Rational functions over **Q**

27.1 Introduction

The module `fmpz_poly_q` provides functions for performing arithmetic on rational functions in $\mathbf{Q}(t)$, represented as quotients of integer polynomials of type `fmpz_poly_t`. These functions start with the prefix `fmpz_poly_q_`.

Rational functions are stored in objects of type `fmpz_poly_q_t`, which is an array of `fmpz_poly_q_struct`'s of length one. This permits passing parameters of type `fmpz_poly_q_t` by reference.

The representation of a rational function as the quotient of two integer polynomials can be made canonical by demanding the numerator and denominator to be coprime (as integer polynomials) and the denominator to have positive leading coefficient. As the only special case, we represent the zero function as 0/1. All arithmetic functions assume that the operands are in this canonical form, and canonicalize their result. If the numerator or denominator is modified individually, for example using the macros `fmpz_poly_q_numref()` and `fmpz_poly_q_denref()`, it is the user's responsibility to canonicalise the rational function using the function `fmpz_poly_q_canonicalise()` if necessary.

All methods support aliasing of their inputs and outputs *unless* explicitly stated otherwise, subject to the following caveat. If different rational functions (as objects in memory, not necessarily in the mathematical sense) share some of the underlying integer polynomial objects, the behaviour is undefined.

The basic arithmetic operations, addition, subtraction and multiplication, are all implemented using adapted versions of Henrici's algorithms, see [17]. Differentiation is implemented in a way slightly improving on the algorithm described in [18].

27.2 Simple example

The following example computes the product of two rational functions and prints the result:

```
#include "fmpz_poly_q.h"
...
char *str, *strf, *strg;
fmpz_poly_q_t f, g;
```

```
fmpz_poly_q_init(f);
fmpz_poly_q_init(g);
fmpz_poly_q_set_str(f, "2  1 3/1  2");
fmpz_poly_q_set_str(g, "1  3/2  2 7");
strf = fmpz_poly_q_get_str_pretty(f, "t");
strg = fmpz_poly_q_get_str_pretty(g, "t");
fmpz_poly_q_mul(f, f, g);
str  = fmpz_poly_q_get_str_pretty(f, "t");
flint_printf("%s * %s = %s\n", strf, strg, str);
free(str);
free(strf);
free(strg);
fmpz_poly_q_clear(f);
fmpz_poly_q_clear(g);
```

The output is:

```
(3*t+1)/2 * 3/(7*t+2) = (9*t+3)/(14*t+4)
```

27.3 Memory management

We represent a rational function over $\mathbf{Q}$ as the quotient of two coprime integer polynomials of type `fmpz_poly_t`, enforcing that the leading coefficient of the denominator is positive. The zero function is represented as $0/1$.

```
void fmpz_poly_q_init(fmpz_poly_q_t rop)
```

Initialises `rop`.

```
void fmpz_poly_q_clear(fmpz_poly_q_t rop)
```

Clears the object `rop`.

```
fmpz_poly_struct * fmpz_poly_q_numref(const fmpz_poly_q_t
   op)
```

Returns a reference to the numerator of `op`.

```
fmpz_poly_struct * fmpz_poly_q_denref(const fmpz_poly_q_t
   op)
```

Returns a reference to the denominator of `op`.

```
void fmpz_poly_q_canonicalise(fmpz_poly_q_t rop)
```

Brings `rop` into canonical form, only assuming that the denominator is non-zero.

```
int fmpz_poly_q_is_canonical(const fmpz_poly_q_t op)
```

Checks whether the rational function `op` is in canonical form.

27.4 Randomisation

```
void fmpz_poly_q_randtest(fmpz_poly_q_t poly, flint_rand_t
   state, slong len1, mp_bitcnt_t bits1, slong len2,
   mp_bitcnt_t bits2)
```

Sets `poly` to a random rational function.

```
void fmpz_poly_q_randtest_not_zero(fmpz_poly_q_t poly,
   flint_rand_t state, slong len1, mp_bitcnt_t bits1, slong
   len2, mp_bitcnt_t bits2)
```

Sets `poly` to a random non-zero rational function.

27.5 Assignment

```
void fmpz_poly_q_set(fmpz_poly_q_t rop, const fmpz_poly_q_t
   op)
```

Sets the element `rop` to the same value as the element `op`.

```
void fmpz_poly_q_set_si(fmpz_poly_q_t rop, slong op)
```

Sets the element `rop` to the value given by the `slong` `op`.

```
void fmpz_poly_q_swap(fmpz_poly_q_t op1, fmpz_poly_q_t op2)
```

Swaps the elements `op1` and `op2`.

This is done efficiently by swapping pointers.

```
void fmpz_poly_q_zero(fmpz_poly_q_t rop)
```

Sets `rop` to zero.

```
void fmpz_poly_q_one(fmpz_poly_q_t rop)
```

Sets `rop` to one.

```
void fmpz_poly_q_neg(fmpz_poly_q_t rop, const fmpz_poly_q_t
   op)
```

Sets the element `rop` to the additive inverse of `op`.

```
void fmpz_poly_q_inv(fmpz_poly_q_t rop, const fmpz_poly_q_t
   op)
```

Sets the element `rop` to the multiplicative inverse of `op`.

Assumes that the element `op` is non-zero.

27.6 Comparison

```
int fmpz_poly_q_is_zero(const fmpz_poly_q_t op)
```

Returns whether the element `op` is zero.

```
int fmpz_poly_q_is_one(const fmpz_poly_q_t op)
```

Returns whether the element `rop` is equal to the constant polynomial 1.

```
int fmpz_poly_q_equal(const fmpz_poly_q_t op1, const
   fmpz_poly_q_t op2)
```

Returns whether the two elements `op1` and `op2` are equal.

27.7 Addition and subtraction

```
void fmpz_poly_q_add(fmpz_poly_q_t rop, const fmpz_poly_q_t
   op1, const fmpz_poly_q_t op2)
```

Sets `rop` to the sum of `op1` and `op2`.

```
void fmpz_poly_q_sub(fmpz_poly_q_t rop, const fmpz_poly_q_t
   op1, const fmpz_poly_q_t op2)
```

Sets `rop` to the difference of `op1` and `op2`.

```
void fmpz_poly_q_addmul(fmpz_poly_q_t rop, const
   fmpz_poly_q_t op1, const fmpz_poly_q_t op2)
```

Adds the product of `op1` and `op2` to `rop`.

```
void fmpz_poly_q_submul(fmpz_poly_q_t rop, const
   fmpz_poly_q_t op1, const fmpz_poly_q_t op2)
```

Subtracts the product of `op1` and `op2` from `rop`.

27.8 Scalar multiplication and division

```
void fmpz_poly_q_scalar_mul_si(fmpz_poly_q_t rop, const
   fmpz_poly_q_t op, slong x)
```

Sets `rop` to the product of the rational function `op` and the `slong` integer x.

```
void fmpz_poly_q_scalar_mul_mpz(fmpz_poly_q_t rop, const
   fmpz_poly_q_t op, const mpz_t x)
```

Sets `rop` to the product of the rational function `op` and the `mpz_t` integer x.

```
void fmpz_poly_q_scalar_mul_mpq(fmpz_poly_q_t rop, const
   fmpz_poly_q_t op, const mpq_t x)
```

Sets `rop` to the product of the rational function `op` and the `mpq_t` rational x.

```
void fmpz_poly_q_scalar_div_si(fmpz_poly_q_t rop, const
   fmpz_poly_q_t op, slong x)
```

Sets `rop` to the quotient of the rational function `op` and the `slong` integer x.

```
void fmpz_poly_q_scalar_div_mpz(fmpz_poly_q_t rop, const
   fmpz_poly_q_t op, const mpz_t x)
```

Sets `rop` to the quotient of the rational function `op` and the `mpz_t` integer x.

```
void fmpz_poly_q_scalar_div_mpq(fmpz_poly_q_t rop, const
   fmpz_poly_q_t op, const mpq_t x)
```

Sets `rop` to the quotient of the rational function `op` and the `mpq_t` rational x.

27.9 Multiplication and division

```
void fmpz_poly_q_mul(fmpz_poly_q_t rop, const fmpz_poly_q_t
   op1, const fmpz_poly_q_t op2)
```

Sets `rop` to the product of `op1` and `op2`.

```
void fmpz_poly_q_div(fmpz_poly_q_t rop, const fmpz_poly_q_t
   op1, const fmpz_poly_q_t op2)
```

Sets `rop` to the quotient of `op1` and `op2`.

27.10 Powering

```
void fmpz_poly_q_pow(fmpz_poly_q_t rop, const fmpz_poly_q_t
    op, ulong exp)
```

Sets `rop` to the `exp`-th power of `op`.

The corner case of `exp == 0` is handled by setting `rop` to the constant function 1. Note that this includes the case $0^0 = 1$.

27.11 Derivative

```
void fmpz_poly_q_derivative(fmpz_poly_q_t rop, const
    fmpz_poly_q_t op)
```

Sets `rop` to the derivative of `op`.

27.12 Evaluation

```
int fmpz_poly_q_evaluate(mpq_t rop, const fmpz_poly_q_t f,
    const mpq_t a)
```

Sets `rop` to f evaluated at the rational a.

If the denominator evaluates to zero at a, returns non-zero and does not modify any of the variables. Otherwise, returns 0 and sets `rop` to the rational $f(a)$.

27.13 Input and output

The following three methods enable users to construct elements of type `fmpz_poly_q_t` from strings or to obtain string representations of such elements.

The format used is based on the FLINT format for integer polynomials of type `fmpz_poly_t`, which we recall first:

A non-zero polynomial $a_0 + a_1 X + \cdots + a_n X^n$ of length $n+1$ is represented by the string `"n+1 a_0 a_1 ... a_n"`, where there are two space characters following the length and single space characters separating the individual coefficients. There is no leading or trailing white-space. The zero polynomial is simply represented by `"0"`.

We adapt this notation for rational functions as follows. We denote the zero function by `"0"`. Given a non-zero function with numerator and denominator string representations `num` and `den`, respectively, we use the string `num/den` to represent the rational function, unless the denominator is equal to one, in which case we simply use `num`.

There is also a `_pretty` variant available, which bases the string parts for the numerator and denominator on the output of the function `fmpz_poly_get_str_pretty` and introduces parentheses where necessary.

Note that currently these functions are not optimised for performance and are intended to be used only for debugging purposes or one-off input and output, rather than as a low-level parser.

```
int fmpz_poly_q_set_str(fmpz_poly_q_t rop, const char *s)
```

Sets `rop` to the rational function given by the string `s`.

```
char * fmpz_poly_q_get_str(const fmpz_poly_q_t op)
```

Returns the string representation of the rational function `op`.

```
char * fmpz_poly_q_get_str_pretty(const fmpz_poly_q_t op,
   const char *x)
```

Returns the pretty string representation of the rational function `op`.

```
int fmpz_poly_q_print(const fmpz_poly_q_t op)
```

Prints the representation of the rational function `op` to `stdout`.

```
int fmpz_poly_q_print_pretty(const fmpz_poly_q_t op, const
   char *x)
```

Prints the pretty representation of the rational function `op` to `stdout`.

§28. fmpz_poly_mat

Matrices over $\mathbf{Z}[x]$

The `fmpz_poly_mat_t` data type represents matrices whose entries are integer polynomials.

The `fmpz_poly_mat_t` type is defined as an array of `fmpz_poly_mat_struct`'s of length one. This permits passing parameters of type `fmpz_poly_mat_t` by reference.

An integer polynomial matrix internally consists of a single array of `fmpz_poly_struct`'s, representing a dense matrix in row-major order. This array is only directly indexed during memory allocation and deallocation. A separate array holds pointers to the start of each row, and is used for all indexing. This allows the rows of a matrix to be permuted quickly by swapping pointers.

Matrices having zero rows or columns are allowed.

The shape of a matrix is fixed upon initialisation. The user is assumed to provide input and output variables whose dimensions are compatible with the given operation.

28.1 Simple example

The following example constructs the matrix $\begin{pmatrix} 2x+1 & x \\ 1-x & -1 \end{pmatrix}$ and computes its determinant.

```
#include "fmpz_poly.h"
#include "fmpz_poly_mat.h"
...
fmpz_poly_mat_t A;
fmpz_poly_t P;

fmpz_poly_mat_init(A, 2, 2);
fmpz_poly_init(P);

fmpz_poly_set_str(fmpz_poly_mat_entry(A, 0, 0), "2  1 2");
fmpz_poly_set_str(fmpz_poly_mat_entry(A, 0, 1), "2  0 1");
fmpz_poly_set_str(fmpz_poly_mat_entry(A, 1, 0), "2  1 -1");
fmpz_poly_set_str(fmpz_poly_mat_entry(A, 1, 1), "1  -1");

fmpz_poly_mat_det(P, A);
fmpz_poly_print_pretty(P, "x");
```

```
fmpz_poly_clear(P);
fmpz_poly_mat_clear(A);
```

The output is:

```
x^2-3*x-1
```

28.2 Memory management

```
void fmpz_poly_mat_init(fmpz_poly_mat_t mat, slong rows,
   slong cols)
```

Initialises a matrix with the given number of rows and columns for use.

```
void fmpz_poly_mat_init_set(fmpz_poly_mat_t mat, const
   fmpz_poly_mat_t src)
```

Initialises a matrix `mat` of the same dimensions as `src`, and sets it to a copy of `src`.

```
void fmpz_poly_mat_clear(fmpz_poly_mat_t mat)
```

Frees all memory associated with the matrix. The matrix must be reinitialised if it is to be used again.

28.3 Basic properties

```
slong fmpz_poly_mat_nrows(const fmpz_poly_mat_t mat)
```

Returns the number of rows in `mat`.

```
slong fmpz_poly_mat_ncols(const fmpz_poly_mat_t mat)
```

Returns the number of columns in `mat`.

28.4 Basic assignment and manipulation

```
MACRO fmpz_poly_mat_entry(mat,i,j)
```

Gives a reference to the entry at row `i` and column `j`. The reference can be passed as an input or output variable to any `fmpz_poly` function for direct manipulation of the matrix element. No bounds checking is performed.

```
void fmpz_poly_mat_set(fmpz_poly_mat_t mat1, const
   fmpz_poly_mat_t mat2)
```

Sets `mat1` to a copy of `mat2`.

```
void fmpz_poly_mat_swap(fmpz_poly_mat_t mat1,
   fmpz_poly_mat_t mat2)
```

Swaps `mat1` and `mat2` efficiently.

28.5 Input and output

```
void fmpz_poly_mat_print(const fmpz_poly_mat_t mat, const
   char * x)
```

Prints the matrix `mat` to standard output, using the variable `x`.

28.6 Random matrix generation

```
void fmpz_poly_mat_randtest(fmpz_poly_mat_t mat,
   flint_rand_t state, slong len, mp_bitcnt_t bits)
```

This is equivalent to applying `fmpz_poly_randtest` to all entries in the matrix.

```
void fmpz_poly_mat_randtest_unsigned(fmpz_poly_mat_t mat,
   flint_rand_t state, slong len, mp_bitcnt_t bits)
```

This is equivalent to applying `fmpz_poly_randtest_unsigned` to all entries in the matrix.

```
void fmpz_poly_mat_randtest_sparse(fmpz_poly_mat_t A,
   flint_rand_t state, slong len, mp_bitcnt_t bits, float
   density)
```

Creates a random matrix with the amount of nonzero entries given approximately by the `density` variable, which should be a fraction between 0 (most sparse) and 1 (most dense).

The nonzero entries will have random lengths between 1 and `len`.

28.7 Special matrices

```
void fmpz_poly_mat_zero(fmpz_poly_mat_t mat)
```

Sets `mat` to the zero matrix.

```
void fmpz_poly_mat_one(fmpz_poly_mat_t mat)
```

Sets `mat` to the unit or identity matrix of given shape, having the element 1 on the main diagonal and zeros elsewhere. If `mat` is nonsquare, it is set to the truncation of a unit matrix.

28.8 Basic comparison and properties

```
int fmpz_poly_mat_equal(const fmpz_poly_mat_t mat1, const
   fmpz_poly_mat_t mat2)
```

Returns nonzero if `mat1` and `mat2` have the same shape and all their entries agree, and returns zero otherwise.

```
int fmpz_poly_mat_is_zero(const fmpz_poly_mat_t mat)
```

Returns nonzero if all entries in `mat` are zero, and returns zero otherwise.

```
int fmpz_poly_mat_is_one(const fmpz_poly_mat_t mat)
```

Returns nonzero if all entry of `mat` on the main diagonal are the constant polynomial 1 and all remaining entries are zero, and returns zero otherwise. The matrix need not be square.

```
int fmpz_poly_mat_is_empty(const fmpz_poly_mat_t mat)
```

Returns a non-zero value if the number of rows or the number of columns in `mat` is zero, and otherwise returns zero.

```
int fmpz_poly_mat_is_square(const fmpz_poly_mat_t mat)
```

Returns a non-zero value if the number of rows is equal to the number of columns in `mat`, and otherwise returns zero.

28.9 Norms

```
slong fmpz_poly_mat_max_bits(const fmpz_poly_mat_t A)
```

Returns the maximum number of bits among the coefficients of the entries in `A`, or the negative of that value if any coefficient is negative.

```
slong fmpz_poly_mat_max_length(const fmpz_poly_mat_t A)
```

Returns the maximum polynomial length among all the entries in `A`.

28.10 Transpose

```
void fmpz_poly_mat_transpose(fmpz_poly_mat_t B, const
   fmpz_poly_mat_t A)
```

Sets B to A^t.

28.11 Evaluation

```
void fmpz_poly_mat_evaluate_fmpz(fmpz_mat_t B, const
   fmpz_poly_mat_t A, const fmpz_t x)
```

Sets the `fmpz_mat_t` `B` to `A` evaluated entrywise at the point `x`.

28.12 Arithmetic

```
void fmpz_poly_mat_scalar_mul_fmpz_poly(fmpz_poly_mat_t B,
   const fmpz_poly_mat_t A, const fmpz_poly_t c)
```

Sets `B` to `A` multiplied entrywise by the polynomial `c`.

```
void fmpz_poly_mat_scalar_mul_fmpz(fmpz_poly_mat_t B, const
   fmpz_poly_mat_t A, const fmpz_t c)
```

Sets `B` to `A` multiplied entrywise by the integer `c`.

```
void fmpz_poly_mat_add(fmpz_poly_mat_t C, const
   fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
```

Sets `C` to the sum of `A` and `B`. All matrices must have the same shape. Aliasing is allowed.

```
void fmpz_poly_mat_sub(fmpz_poly_mat_t C, const
   fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
```

Sets `C` to the sum of `A` and `B`. All matrices must have the same shape. Aliasing is allowed.

```
void fmpz_poly_mat_neg(fmpz_poly_mat_t B, const
   fmpz_poly_mat_t A)
```

Sets `B` to the negation of `A`. The matrices must have the same shape. Aliasing is allowed.

```
void fmpz_poly_mat_mul(fmpz_poly_mat_t C, const
   fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
```

Sets `C` to the matrix product of `A` and `B`. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical and KS multiplication.

```
void fmpz_poly_mat_mul_classical(fmpz_poly_mat_t C, const
   fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
```

Sets `C` to the matrix product of `A` and `B`, computed using the classical algorithm. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

```
void fmpz_poly_mat_mul_KS(fmpz_poly_mat_t C, const
   fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
```

Sets `C` to the matrix product of `A` and `B`, computed using Kronecker segmentation. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

```
void fmpz_poly_mat_mullow(fmpz_poly_mat_t C, const
   fmpz_poly_mat_t A, const fmpz_poly_mat_t B, slong len)
```

Sets `C` to the matrix product of `A` and `B`, truncating each entry in the result to length `len`. Uses classical matrix multiplication. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

```
void fmpz_poly_mat_sqr(fmpz_poly_mat_t B, const
   fmpz_poly_mat_t A)
```

Sets `B` to the square of `A`, which must be a square matrix. Aliasing is allowed. This function automatically chooses between classical and KS squaring.

```
void fmpz_poly_mat_sqr_classical(fmpz_poly_mat_t B, const
   fmpz_poly_mat_t A)
```

Sets `B` to the square of `A`, which must be a square matrix. Aliasing is allowed. This function uses direct formulas for very small matrices, and otherwise classical matrix multiplication.

```
void fmpz_poly_mat_sqr_KS(fmpz_poly_mat_t B, const
   fmpz_poly_mat_t A)
```

Sets `B` to the square of `A`, which must be a square matrix. Aliasing is allowed. This function uses Kronecker segmentation.

```
void fmpz_poly_mat_sqrlow(fmpz_poly_mat_t B, const
   fmpz_poly_mat_t A, slong len)
```

Sets `B` to the square of `A`, which must be a square matrix, truncating all entries to length `len`. Aliasing is allowed. This function uses direct formulas for very small matrices, and otherwise classical matrix multiplication.

```
void fmpz_poly_mat_pow(fmpz_poly_mat_t B, const
   fmpz_poly_mat_t A, ulong exp)
```

Sets `B` to `A` raised to the power `exp`, where `A` is a square matrix. Uses exponentiation by squaring. Aliasing is allowed.

```
void fmpz_poly_mat_pow_trunc(fmpz_poly_mat_t B, const
   fmpz_poly_mat_t A, ulong exp, slong len)
```

Sets `B` to `A` raised to the power `exp`, truncating all entries to length `len`, where `A` is a square matrix. Uses exponentiation by squaring. Aliasing is allowed.

```
void fmpz_poly_mat_prod(fmpz_poly_mat_t res,
    fmpz_poly_mat_t * const factors, slong n)
```

Sets `res` to the product of the `n` matrices given in the vector `factors`, all of which must be square and of the same size. Uses binary splitting.

28.13 Row reduction

```
slong fmpz_poly_mat_find_pivot_any(const fmpz_poly_mat_t
    mat, slong start_row, slong end_row, slong c)
```

Attempts to find a pivot entry for row reduction. Returns a row index r between `start_row` (inclusive) and `stop_row` (exclusive) such that column c in `mat` has a nonzero entry on row r, or returns -1 if no such entry exists.

This implementation simply chooses the first nonzero entry from it encounters. This is likely to be a nearly optimal choice if all entries in the matrix have roughly the same size, but can lead to unnecessary coefficient growth if the entries vary in size.

```
slong fmpz_poly_mat_find_pivot_partial(const
    fmpz_poly_mat_t mat, slong start_row, slong end_row,
    slong c)
```

Attempts to find a pivot entry for row reduction. Returns a row index r between `start_row` (inclusive) and `stop_row` (exclusive) such that column c in `mat` has a nonzero entry on row r, or returns -1 if no such entry exists.

This implementation searches all the rows in the column and chooses the nonzero entry of smallest degree. If there are several entries with the same minimal degree, it chooses the entry with the smallest coefficient bit bound. This heuristic typically reduces coefficient growth when the matrix entries vary in size.

```
slong fmpz_poly_mat_fflu(fmpz_poly_mat_t B, fmpz_poly_t
    den, slong * perm, const fmpz_poly_mat_t A, int
    rank_check)
```

Uses fraction-free Gaussian elimination to set (`B`, `den`) to a fraction-free LU decomposition of `A` and returns the rank of `A`. Aliasing of `A` and `B` is allowed.

Pivot elements are chosen with `fmpz_poly_mat_find_pivot_partial`. If `perm` is non-`NULL`, the permutation of rows in the matrix will also be applied to `perm`.

If `rank_check` is set, the function aborts and returns 0 if the matrix is detected not to have full rank without completing the elimination.

The denominator `den` is set to $\pm \det(A)$, where the sign is decided by the parity of the permutation. Note that the determinant is not generally the minimal denominator.

```
slong fmpz_poly_mat_rref(fmpz_poly_mat_t B, fmpz_poly_t
    den, const fmpz_poly_mat_t A)
```

Sets (`B`, `den`) to the reduced row echelon form of `A` and returns the rank of `A`. Aliasing of `A` and `B` is allowed.

The denominator `den` is set to $\pm \det(A)$. Note that the determinant is not generally the minimal denominator.

28.14 Trace

```
void fmpz_poly_mat_trace(fmpz_poly_t trace, const
    fmpz_poly_mat_t mat)
```

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

28.15 Determinant and rank

```
void fmpz_poly_mat_det(fmpz_poly_t det, const
    fmpz_poly_mat_t A)
```

Sets `det` to the determinant of the square matrix `A`. Uses a direct formula, fraction-free LU decomposition, or interpolation, depending on the size of the matrix.

```
void fmpz_poly_mat_det_fflu(fmpz_poly_t det, const
    fmpz_poly_mat_t A)
```

Sets `det` to the determinant of the square matrix `A`. The determinant is computed by performing a fraction-free LU decomposition on a copy of `A`.

```
void fmpz_poly_mat_det_interpolate(fmpz_poly_t det, const
    fmpz_poly_mat_t A)
```

Sets `det` to the determinant of the square matrix `A`. The determinant is computed by determing a bound n for its length, evaluating the matrix at n distinct points, computing the determinant of each integer matrix, and forming the interpolating polynomial.

```
slong fmpz_poly_mat_rank(const fmpz_poly_mat_t A)
```

Returns the rank of `A`. Performs fraction-free LU decomposition on a copy of `A`.

28.16 Inverse

```
int fmpz_poly_mat_inv(fmpz_poly_mat_t Ainv, fmpz_poly_t
    den, const fmpz_poly_mat_t A)
```

Sets (`Ainv`, `den`) to the inverse matrix of `A`. Returns 1 if `A` is nonsingular and 0 if `A` is singular. Aliasing of `Ainv` and `A` is allowed.

More precisely, `det` will be set to the determinant of `A` and `Ainv` will be set to the adjugate matrix of `A`. Note that the determinant is not necessarily the minimal denominator.

Uses fraction-free LU decomposition, followed by solving for the identity matrix.

28.17 Nullspace

```
slong fmpz_poly_mat_nullspace(fmpz_poly_mat_t res, const
    fmpz_poly_mat_t mat)
```

Computes the right rational nullspace of the matrix `mat` and returns the nullity.

More precisely, assume that `mat` has rank r and nullity n. Then this function sets the first n columns of `res` to linearly independent vectors spanning the nullspace of `mat`. As a result, we always have rank(`res`) $= n$, and `mat` $\times$ `res` is the zero matrix.

The computed basis vectors will not generally be in a reduced form. In general, the polynomials in each column vector in the result will have a nontrivial common GCD.

28.18 Solving

```
int fmpz_poly_mat_solve(fmpz_poly_mat_t X, fmpz_poly_t den,
    const fmpz_poly_mat_t A, const fmpz_poly_mat_t B)
```

Solves the equation $AX = B$ for nonsingular A. More precisely, computes (`X`, `den`) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

```
int fmpz_poly_mat_solve_fflu(fmpz_poly_mat_t X, fmpz_poly_t
   den, const fmpz_poly_mat_t A, const fmpz_poly_mat_t B);
```

Solves the equation $AX = B$ for nonsingular A. More precisely, computes (`X`, `den`) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

```
void fmpz_poly_mat_solve_fflu_precomp(fmpz_poly_mat_t X,
   const slong * perm, const fmpz_poly_mat_t FFLU, const
   fmpz_poly_mat_t B);
```

Performs fraction-free forward and back substitution given a precomputed fraction-free LU decomposition and corresponding permutation.

§29. nmod_vec

Vectors over $\mathbf{Z}/n\mathbf{Z}$ for word-sized moduli

29.1 Memory management

`mp_ptr _nmod_vec_init(slong len)`

Returns a vector of the given length. The entries are not necessarily zero.

`void _nmod_vec_clear(mp_ptr vec)`

Frees the memory used by the given vector.

29.2 Modular reduction and arithmetic

`void nmod_init(nmod_t * mod, mp_limb_t n)`

Initialises the given `nmod_t` structure for reduction modulo n with a precomputed inverse.

`NMOD_RED2(r, a_hi, a_lo, mod)`

Macro to set r to a reduced modulo `mod.n`, where a consists of two limbs `(a_hi, a_lo)`. The `mod` parameter must be a valid `nmod_t` structure. It is assumed that `a_hi` is already reduced modulo `mod.n`.

`NMOD_RED(r, a, mod)`

Macro to set r to a reduced modulo `mod.n`. The `mod` parameter must be a valid `nmod_t` structure.

`NMOD2_RED2(r, a_hi, a_lo, mod)`

Macro to set r to a reduced modulo `mod.n`, where a consists of two limbs `(a_hi, a_lo)`. The `mod` parameter must be a valid `nmod_t` structure. No assumptions are made about `a_hi`.

`NMOD_RED3(r, a_hi, a_me, a_lo, mod)`

Macro to set r to a reduced modulo `mod.n`, where a consists of three limbs `(a_hi, a_me, a_lo)`. The `mod` parameter must be a valid `nmod_t` structure. It is assumed that `a_hi` is already reduced modulo `mod.n`.

`NMOD_ADDMUL(r, a, b, mod)`

Macro to set r to $r + ab$ reduced modulo `mod.n`. The `mod` parameter must be a valid `nmod_t` structure. It is assumed that r, a, b are already reduced modulo `mod.n`.

`mp_limb_t _nmod_add(mp_limb_t a, mp_limb_t b, nmod_t mod)`

Returns $a + b$ modulo `mod.n`. It is assumed that `mod` is no more than `FLINT_BITS - 1` bits. It is assumed that a and b are already reduced modulo `mod.n`.

`mp_limb_t nmod_add(mp_limb_t a, mp_limb_t b, nmod_t mod)`

Returns $a+b$ modulo `mod.n`. No assumptions are made about `mod.n`. It is assumed that a and b are already reduced modulo `mod.n`.

`mp_limb_t _nmod_sub(mp_limb_t a, mp_limb_t b, nmod_t mod)`

Returns $a - b$ modulo `mod.n`. It is assumed that `mod` is no more than `FLINT_BITS - 1` bits. It is assumed that a and b are already reduced modulo `mod.n`.

`mp_limb_t nmod_sub(mp_limb_t a, mp_limb_t b, nmod_t mod)`

Returns $a-b$ modulo `mod.n`. No assumptions are made about `mod.n`. It is assumed that a and b are already reduced modulo `mod.n`.

`mp_limb_t nmod_neg(mp_limb_t a, nmod_t mod)`

Returns $-a$ modulo `mod.n`. It is assumed that a is already reduced modulo `mod.n`, but no assumptions are made about the latter.

`mp_limb_t nmod_mul(mp_limb_t a, mp_limb_t b, nmod_t mod)`

Returns ab modulo `mod.n`. No assumptions are made about `mod.n`. It is assumed that a and b are already reduced modulo `mod.n`.

`mp_limb_t nmod_inv(mp_limb_t a, nmod_t mod)`

Returns a^{-1} modulo `mod.n`. The inverse is assumed to exist.

`mp_limb_t nmod_div(mp_limb_t a, mp_limb_t b, nmod_t mod)`

Returns a^{-1} modulo `mod.n`. The inverse of b is assumed to exist. It is assumed that a is already reduced modulo `mod.n`.

`mp_limb_t nmod_pow_ui(mp_limb_t a, ulong e, nmod_t mod)`

Returns a^e modulo `mod.n`. No assumptions are made about `mod.n`. It is assumed that a is already reduced modulo `mod.n`.

29.3 Random functions

```
void _nmod_vec_randtest(mp_ptr vec, flint_rand_t state,
   slong len, nmod_t mod)
```

Sets `vec` to a random vector of the given length with entries reduced modulo `mod.n`.

29.4 Basic manipulation and comparison

`void _nmod_vec_set(mp_ptr res, mp_srcptr vec, slong len)`

Copies `len` entries from the vector `vec` to `res`.

```
void _nmod_vec_zero(mp_ptr vec, slong len)
```

Zeros the given vector of the given length.

```
void _nmod_vec_swap(mp_ptr a, mp_ptr b, slong length)
```

Swaps the vectors `a` and `b` of length n by actually swapping the entries.

```
void _nmod_vec_reduce(mp_ptr res, mp_srcptr vec, slong len,
   nmod_t mod)
```

Reduces the entries of `(vec, len)` modulo `mod.n` and set `res` to the result.

```
mp_bitcnt_t _nmod_vec_max_bits(mp_srcptr vec, slong len)
```

Returns the maximum number of bits of any entry in the vector.

```
int _nmod_vec_equal(mp_srcptr vec, mp_srcptr vec2, slong
   len)
```

Returns 1 if `(vec, len)` is equal to `(vec2, len)`, otherwise returns 0.

29.5 Arithmetic operations

```
void _nmod_vec_add(mp_ptr res, mp_srcptr vec1, mp_srcptr
   vec2, slong len, nmod_t mod)
```

Sets `(res, len)` to the sum of `(vec1, len)` and `(vec2, len)`.

```
void _nmod_vec_sub(mp_ptr res, mp_srcptr vec1, mp_srcptr
   vec2, slong len, nmod_t mod)
```

Sets `(res, len)` to the difference of `(vec1, len)` and `(vec2, len)`.

```
void _nmod_vec_neg(mp_ptr res, mp_srcptr vec, slong len,
   nmod_t mod)
```

Sets `(res, len)` to the negation of `(vec, len)`.

```
void _nmod_vec_scalar_mul_nmod(mp_ptr res, mp_srcptr vec,
   slong len, mp_limb_t c, nmod_t mod)
```

Sets `(res, len)` to `(vec, len)` multiplied by c.

```
void _nmod_vec_scalar_addmul_nmod(mp_ptr res, mp_srcptr
   vec, slong len, mp_limb_t c, nmod_t mod)
```

Adds `(vec, len)` times c to the vector `(res, len)`.

29.6 Dot products

```
int _nmod_vec_dot_bound_limbs(slong len, nmod_t mod)
```

Returns the number of limbs (0, 1, 2 or 3) needed to represent the unreduced dot product of two vectors of length `len` having entries modulo `mod.n`, assuming that `len` is nonnegative and that `mod.n` is nonzero. The computed bound is tight. In other words, this function returns the precise limb size of `len` times `(mod.n - 1)^ 2`.

```
macro NMOD_VEC_DOT(res, i, len, expr1, expr2, mod, nlimbs)
```

Effectively performs the computation

```
res = 0;
for (i = 0; i < len; i++)
    res += (expr1) * (expr2);
```

but with the arithmetic performed modulo `mod`. The `nlimbs` parameter should be 0, 1, 2 or 3, specifying the number of limbs needed to represent the unreduced result.

```
mp_limb_t _nmod_vec_dot(mp_srcptr vec1, mp_srcptr vec2,
   slong len, nmod_t mod, int nlimbs)
```

Returns the dot product of (`vec1`, `len`) and (`vec2`, `len`). The `nlimbs` parameter should be 0, 1, 2 or 3, specifying the number of limbs needed to represent the unreduced result.

```
mp_limb_t _nmod_vec_dot_ptr(mp_srcptr vec1, const mp_ptr *
   vec2, slong offset, slong len, nmod_t mod, int nlimbs)
```

Returns the dot product of (`vec1`, `len`) and the values at `vec2[i][offset]`. The `nlimbs` parameter should be 0, 1, 2 or 3, specifying the number of limbs needed to represent the unreduced result.

§30. nmod_poly

Polynomials over $\mathbf{Z}/n\mathbf{Z}$ for word-sized moduli

30.1 Introduction

The `nmod_poly_t` data type represents elements of $\mathbf{Z}/n\mathbf{Z}[x]$ for a fixed modulus n. The `nmod_poly` module provides routines for memory management, basic arithmetic and some higher level functions such as GCD, etc.

Each coefficient of an `nmod_poly_t` is of type `mp_limb_t` and represents an integer reduced modulo the fixed modulus n.

Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

30.2 Simple example

The following example computes the square of the polynomial $5x^3 + 6$ in $\mathbf{Z}/7\mathbf{Z}[x]$.

```
#include "nmod_poly.h"
...
nmod_poly_t x, y;
nmod_poly_init(x, 7);
nmod_poly_init(y, 7);
nmod_poly_set_coeff_ui(x, 3, 5);
nmod_poly_set_coeff_ui(x, 0, 6);
nmod_poly_mul(y, x, x);
nmod_poly_print(x); flint_printf("\n");
nmod_poly_print(y); flint_printf("\n");
nmod_poly_clear(x);
nmod_poly_clear(y);
```

The output is:

```
4 7  6 0 0 5
7 7  1 0 0 4 0 0 4
```

30.3 Definition of the nmod_poly_t type

The `nmod_poly_t` type is a typedef for an array of length 1 of `nmod_poly_struct`'s. This permits passing parameters of type `nmod_poly_t` by reference.

In reality one never deals directly with the `struct` and simply deals with objects of type `nmod_poly_t`. For simplicity we will think of an `nmod_poly_t` as a `struct`, though in practice to access fields of this `struct`, one needs to dereference first, e.g. to access the `length` field of an `nmod_poly_t` called `poly1` one writes `poly1->length`.

An `nmod_poly_t` is said to be *normalised* if either `length` is zero, or if the leading coefficient of the polynomial is non-zero. All `nmod_poly` functions expect their inputs to be normalised and for all coefficients to be reduced modulo n, and unless otherwise specified they produce output that is normalised with coefficients reduced modulo n.

It is recommended that users do not access the fields of an `nmod_poly_t` or its coefficient data directly, but make use of the functions designed for this purpose, detailed below.

Functions in `nmod_poly` do all the memory management for the user. One does not need to specify the maximum length in advance before using a polynomial object. FLINT reallocates space automatically as the computation proceeds, if more space is required.

We now describe the functions available in `nmod_poly`.

30.4 Helper functions

```
int signed_mpn_sub_n(mp_ptr res, mp_srcptr op1, mp_srcptr
    op2, slong n)
```

If `op1 >= op2` return 0 and set `res` to `op1 - op2` else return 1 and set `res` to `op2 - op1`.

30.5 Memory management

```
void nmod_poly_init(nmod_poly_t poly, mp_limb_t n)
```

Initialises `poly`. It will have coefficients modulo n.

```
void nmod_poly_init_preinv(nmod_poly_t poly, mp_limb_t n,
    mp_limb_t ninv)
```

Initialises `poly`. It will have coefficients modulo n. The caller supplies a precomputed inverse limb generated by `n_preinvert_limb()`.

```
void nmod_poly_init2(nmod_poly_t poly, mp_limb_t n, slong
    alloc)
```

Initialises `poly`. It will have coefficients modulo n. Up to `alloc` coefficients may be stored in `poly`.

```
void nmod_poly_init2_preinv(nmod_poly_t poly, mp_limb_t n,
    mp_limb_t ninv, slong alloc)
```

Initialises `poly`. It will have coefficients modulo n. The caller supplies a precomputed inverse limb generated by `n_preinvert_limb()`. Up to `alloc` coefficients may be stored in `poly`.

```
void nmod_poly_realloc(nmod_poly_t poly, slong alloc)
```

Reallocates `poly` to the given length. If the current length is less than `alloc`, the polynomial is truncated and normalised. If `alloc` is zero, the polynomial is cleared.

```
void nmod_poly_clear(nmod_poly_t poly)
```

Clears the polynomial and releases any memory it used. The polynomial cannot be used again until it is initialised.

```
void nmod_poly_fit_length(nmod_poly_t poly, slong alloc)
```

Ensures `poly` has space for at least `alloc` coefficients. This function only ever grows the allocated space, so no data loss can occur.

```
void _nmod_poly_normalise(nmod_poly_t poly)
```

Internal function for normalising a polynomial so that the top coefficient, if there is one at all, is not zero.

30.6 Polynomial properties

```
slong nmod_poly_length(const nmod_poly_t poly)
```

Returns the length of the polynomial `poly`. The zero polynomial has length zero.

```
slong nmod_poly_degree(const nmod_poly_t poly)
```

Returns the degree of the polynomial `poly`. The zero polynomial is deemed to have degree -1.

```
mp_limb_t nmod_poly_modulus(const nmod_poly_t poly)
```

Returns the modulus of the polynomial `poly`. This will be a positive integer.

```
mp_bitcnt_t nmod_poly_max_bits(const nmod_poly_t poly)
```

Returns the maximum number of bits of any coefficient of `poly`.

30.7 Assignment and basic manipulation

```
void nmod_poly_set(nmod_poly_t a, const nmod_poly_t b)
```

Sets `a` to a copy of `b`.

```
void nmod_poly_swap(nmod_poly_t poly1, nmod_poly_t poly2)
```

Efficiently swaps `poly1` and `poly2` by swapping pointers internally.

```
void nmod_poly_zero(nmod_poly_t res)
```

Sets `res` to the zero polynomial.

```
void nmod_poly_truncate(nmod_poly_t poly, slong len)
```

Truncates `poly` to the given length and normalises it. If `len` is greater than the current length of `poly`, then nothing happens.

```
void _nmod_poly_reverse(mp_ptr output, mp_srcptr input,
    slong len, slong m)
```

Sets `output` to the reverse of `input`, which is of length `len`, but thinking of it as a polynomial of length `m`, notionally zero-padded if necessary. The length `m` must be non-negative, but there are no other restrictions. The polynomial `output` must have space for `m` coefficients.

```
void nmod_poly_reverse(nmod_poly_t output, const
    nmod_poly_t input, slong m)
```

Sets output to the reverse of input, thinking of it as a polynomial of length m, notionally zero-padded if necessary). The length m must be non-negative, but there are no other restrictions. The output polynomial will be set to length m and then normalised.

30.8 Randomization

```
void nmod_poly_randtest(nmod_poly_t poly, flint_rand_t
   state, slong len)
```

Generates a random polynomial with length up to len.

```
void nmod_poly_randtest_irreducible(nmod_poly_t poly,
   flint_rand_t state, slong len)
```

Generates a random irreducible polynomial with length up to len.

```
void nmod_poly_randtest_monic(nmod_poly_t poly,
   flint_rand_t state, slong len)
```

Generates a random monic polynomial with length len.

```
void nmod_poly_randtest_monic_irreducible(nmod_poly_t poly,
   flint_rand_t state, slong len)
```

Generates a random monic irreducible polynomial with length len.

```
void nmod_poly_randtest_trinomial(nmod_poly_t poly,
   flint_rand_t state, slong len)
```

Generates a random monic trinomial of length len.

```
int nmod_poly_randtest_trinomial_irreducible(nmod_poly_t
   poly, flint_rand_t state, slong len, slong max_attempts)
```

Attempts to set poly to a monic irreducible trinomial of length len. It will generate up to max_attempts trinomials in attempt to find an irreducible one. If max_attempts is 0, then it will keep generating trinomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.

```
void nmod_poly_randtest_pentomial(nmod_poly_t poly,
   flint_rand_t state, slong len)
```

Generates a random monic pentomial of length len.

```
int nmod_poly_randtest_pentomial_irreducible(nmod_poly_t
   poly, flint_rand_t state, slong len, slong max_attempts)
```

Attempts to set poly to a monic irreducible pentomial of length len. It will generate up to max_attempts pentomials in attempt to find an irreducible one. If max_attempts is 0, then it will keep generating pentomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.

```
void nmod_poly_randtest_sparse_irreducible(nmod_poly_t
   poly, flint_rand_t state, slong len)
```

Attempts to set poly to a sparse, monic irreducible polynomial with length len. It attempts to find an irreducible trinomial. If that does not succeed, it attempts to find a irreducible pentomial. If that fails, then poly is just set to a random monic irreducible polynomial.

30.9 Getting and setting coefficients

```
ulong nmod_poly_get_coeff_ui(const nmod_poly_t poly, slong
    j)
```

Returns the coefficient of `poly` at index `j`, where coefficients are numbered with zero being the constant coefficient, and returns it as an `ulong`. If `j` refers to a coefficient beyond the end of `poly`, zero is returned.

```
void nmod_poly_set_coeff_ui(nmod_poly_t poly, slong j,
    ulong c)
```

Sets the coefficient of `poly` at index `j`, where coefficients are numbered with zero being the constant coefficient, to the value `c` reduced modulo the modulus of `poly`. If `j` refers to a coefficient beyond the current end of `poly`, the polynomial is first resized, with intervening coefficients being set to zero.

30.10 Input and output

```
char * nmod_poly_get_str(const nmod_poly_t poly)
```

Writes `poly` to a string representation. The format is as described for `nmod_poly_print()`. The string must be freed by the user when finished. For this it is sufficient to call `flint_free()`.

```
int nmod_poly_set_str(nmod_poly_t poly, const char * s)
```

Reads `poly` from a string `s`. The format is as described for `nmod_poly_print()`. If a polynomial in the correct format is read, a positive value is returned, otherwise a non-positive value is returned.

```
int nmod_poly_print(const nmod_poly_t a)
```

Prints the polynomial to `stdout`. The length is printed, followed by a space, then the modulus. If the length is zero this is all that is printed, otherwise two spaces followed by a space separated list of coefficients is printed, beginning with the constant coefficient.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int nmod_poly_fread(FILE * f, nmod_poly_t poly)
```

Reads `poly` from the file stream `f`. If this is a file that has just been written, the file should be closed then opened again. The format is as described for `nmod_poly_print()`. If a polynomial in the correct format is read, a positive value is returned, otherwise a non-positive value is returned.

```
int nmod_poly_fprint(FILE * f, const nmod_poly_t poly)
```

Writes a polynomial to the file stream `f`. If this is a file then the file should be closed and reopened before being read. The format is as described for `nmod_poly_print()`. If a polynomial in the correct format is read, a positive value is returned, otherwise a non-positive value is returned. If an error occurs whilst writing to the file, an error message is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int nmod_poly_read(nmod_poly_t poly)
```

Read `poly` from `stdin`. The format is as described for `nmod_poly_print()`. If a polynomial in the correct format is read, a positive value is returned, otherwise a non-positive value is returned.

30.11 Comparison

```
int nmod_poly_equal(const nmod_poly_t a, const nmod_poly_t
    b)
```

Returns 1 if the polynomials are equal, otherwise 0.

```
int nmod_poly_is_zero(const nmod_poly_t poly)
```

Returns 1 if the polynomial `poly` is the zero polynomial, otherwise returns 0.

```
int nmod_poly_is_one(const nmod_poly_t poly)
```

Returns 1 if the polynomial `poly` is the constant polynomial 1, otherwise returns 0.

30.12 Shifting

```
void _nmod_poly_shift_left(mp_ptr res, mp_srcptr poly,
    slong len, slong k)
```

Sets `(res, len + k)` to `(poly, len)` shifted left by `k` coefficients. Assumes that `res` has space for `len + k` coefficients.

```
void nmod_poly_shift_left(nmod_poly_t res, const
    nmod_poly_t poly, slong k)
```

Sets `res` to `poly` shifted left by `k` coefficients, i.e. multiplied by x^k.

```
void _nmod_poly_shift_right(mp_ptr res, mp_srcptr poly,
    slong len, slong k)
```

Sets `(res, len - k)` to `(poly, len)` shifted left by `k` coefficients. It is assumed that `k <= len` and that `res` has space for at least `len - k` coefficients.

```
void nmod_poly_shift_right(nmod_poly_t res, const
    nmod_poly_t poly, slong k)
```

Sets `res` to `poly` shifted right by `k` coefficients, i.e. divide by x^k and throws away the remainder. If `k` is greater than or equal to the length of `poly`, the result is the zero polynomial.

30.13 Addition and subtraction

```
void _nmod_poly_add(mp_ptr res, mp_srcptr poly1, slong
    len1, mp_srcptr poly2, slong len2, nmod_t mod)
```

Sets `res` to the sum of `(poly1, len1)` and `(poly2, len2)`. There are no restrictions on the lengths.

```
void nmod_poly_add(nmod_poly_t res, const nmod_poly_t
    poly1, const nmod_poly_t poly2)
```

Sets `res` to the sum of `poly1` and `poly2`.

```
void _nmod_poly_sub(mp_ptr res, mp_srcptr poly1, slong
    len1, mp_srcptr poly2, slong len2, nmod_t mod)
```

Sets `res` to the difference of `(poly1, len1)` and `(poly2, len2)`. There are no restrictions on the lengths.

```
void nmod_poly_sub(nmod_poly_t res, const nmod_poly_t
    poly1, const nmod_poly_t poly2)
```

Sets `res` to the difference of `poly1` and `poly2`.

```
void nmod_poly_neg(nmod_poly_t res, const nmod_poly_t poly)
```

Sets `res` to the negation of `poly`.

30.14 Scalar multiplication and division

```
void nmod_poly_scalar_mul_nmod(nmod_poly_t res, const
    nmod_poly_t poly, ulong c)
```

Sets `res` to `(poly, len)` multiplied by c, where c is reduced modulo the modulus of `poly`.

```
void _nmod_poly_make_monic(mp_ptr output, mp_srcptr input,
    slong len, nmod_t mod)
```

Sets `output` to be the scalar multiple of `input` of length `len > 0` that has leading coefficient one, if such a polynomial exists. If the leading coefficient of `input` is not invertible, `output` is set to the multiple of `input` whose leading coefficient is the greatest common divisor of the leading coefficient and the modulus of `input`.

```
void nmod_poly_make_monic(nmod_poly_t output, const
    nmod_poly_t input)
```

Sets `output` to be the scalar multiple of `input` with leading coefficient one, if such a polynomial exists. If `input` is zero an exception is raised. If the leading coefficient of `input` is not invertible, `output` is set to the multiple of `input` whose leading coefficient is the greatest common divisor of the leading coefficient and the modulus of `input`.

30.15 Bit packing and unpacking

```
void _nmod_poly_bit_pack(mp_ptr res, mp_srcptr poly, slong
    len, mp_bitcnt_t bits)
```

Packs `len` coefficients of `poly` into fields of the given number of bits in the large integer `res`, i.e. evaluates `poly` at `2^bits` and store the result in `res`. Assumes `len > 0` and `bits > 0`. Also assumes that no coefficient of `poly` is bigger than `bits/2` bits. We also assume `bits < 3 * FLINT_BITS`.

```
void _nmod_poly_bit_unpack(mp_ptr res, slong len, mp_srcptr
    mpn, ulong bits, nmod_t mod)
```

Unpacks `len` coefficients stored in the big integer `mpn` in bit fields of the given number of bits, reduces them modulo the given modulus, then stores them in the polynomial `res`. We assume `len > 0` and `3 * FLINT_BITS > bits > 0`. There are no restrictions on the size of the actual coefficients as stored within the bitfields.

```
void nmod_poly_bit_pack(fmpz_t f, const nmod_poly_t poly,
    mp_bitcnt_t bit_size)
```

Packs `poly` into bitfields of size `bit_size`, writing the result to `f`.

```
void nmod_poly_bit_unpack(nmod_poly_t poly, const fmpz_t f,
    mp_bitcnt_t bit_size)
```

Unpacks the polynomial from fields of size `bit_size` as represented by the integer `f`.

```
void _nmod_poly_KS2_pack1(mp_ptr res, mp_srcptr op, slong
    n, slong s, ulong b, ulong k, slong r)
```

Same as _nmod_poly_KS2_pack, but requires b <= FLINT_BITS.

```
void _nmod_poly_KS2_pack(mp_ptr res, mp_srcptr op, slong n,
   slong s, ulong b, ulong k, slong r)
```

Bit packing routine used by KS2 and KS4 multiplication.

```
void _nmod_poly_KS2_unpack1(mp_ptr res, mp_srcptr op, slong
   n, ulong b, ulong k)
```

Same as _nmod_poly_KS2_unpack, but requires b <= FLINT_BITS (i.e. writes one word per coefficient).

```
void _nmod_poly_KS2_unpack2(mp_ptr res, mp_srcptr op, slong
   n, ulong b, ulong k)
```

Same as _nmod_poly_KS2_unpack, but requires FLINT_BITS < b <= 2 * FLINT_BITS (i.e. writes two words per coefficient).

```
void _nmod_poly_KS2_unpack3(mp_ptr res, mp_srcptr op, slong
   n, ulong b, ulong k)
```

Same as _nmod_poly_KS2_unpack, but requires 2 * FLINT_BITS < b < 3 * FLINT_BITS (i.e. writes three words per coefficient).

```
void _nmod_poly_KS2_unpack(mp_ptr res, mp_srcptr op, slong
   n, ulong b, ulong k)
```

Bit unpacking code used by KS2 and KS4 multiplication.

30.16 KS2/KS4 Reduction

```
void _nmod_poly_KS2_reduce(mp_ptr res, slong s, mp_srcptr
   op, slong n, ulong w, nmod_t mod)
```

Reduction code used by KS2 and KS4 multiplication.

```
void _nmod_poly_KS2_recover_reduce1(mp_ptr res, slong s,
   mp_srcptr op1, mp_srcptr op2, slong n, ulong b, nmod_t
   mod)
```

Same as _nmod_poly_KS2_recover_reduce, but requires 0 < 2 * b <= FLINT_BITS.

```
void _nmod_poly_KS2_recover_reduce2(mp_ptr res, slong s,
   mp_srcptr op1, mp_srcptr op2, slong n, ulong b, nmod_t
   mod)
```

Same as _nmod_poly_KS2_recover_reduce, but requires FLINT_BITS < 2 * b < 2*FLINT_BITS.

```
void _nmod_poly_KS2_recover_reduce2b(mp_ptr res, slong s,
   mp_srcptr op1, mp_srcptr op2, slong n, ulong b, nmod_t
   mod)
```

Same as _nmod_poly_KS2_recover_reduce, but requires b == FLINT_BITS.

```
void _nmod_poly_KS2_recover_reduce3(mp_ptr res, slong s,
   mp_srcptr op1, mp_srcptr op2, slong n, ulong b, nmod_t
   mod)
```

Same as _nmod_poly_KS2_recover_reduce, but requires 2 * FLINT_BITS < 2 * b <= 3 * FLINT_BITS.

```
void _nmod_poly_KS2_recover_reduce(mp_ptr res, slong s,
   mp_srcptr op1, mp_srcptr op2, slong n, ulong b, nmod_t
   mod)
```

Reduction code used by KS4 multiplication.

30.17 Multiplication

```
void _nmod_poly_mul_classical(mp_ptr res, mp_srcptr poly1,
   slong len1, mp_srcptr poly2, slong len2, nmod_t mod)
```

Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2)`. Assumes `len1 >= len2 > 0`. Aliasing of inputs and output is not permitted.

```
void nmod_poly_mul_classical(nmod_poly_t res, const
   nmod_poly_t poly1, const nmod_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _nmod_poly_mullow_classical(mp_ptr res, mp_srcptr
   poly1, slong len1, mp_srcptr poly2, slong len2, slong
   trunc, nmod_t mod)
```

Sets `res` to the lower `trunc` coefficients of the product of `(poly1, len1)` and `(poly2, len2)`. Assumes that `len1 >= len2 > 0` and `trunc > 0`. Aliasing of inputs and output is not permitted.

```
void nmod_poly_mullow_classical(nmod_poly_t res, const
   nmod_poly_t poly1, const nmod_poly_t poly2, slong trunc)
```

Sets `res` to the lower `trunc` coefficients of the product of `poly1` and `poly2`.

```
void _nmod_poly_mulhigh_classical(mp_ptr res, mp_srcptr
   poly1, slong len1, mp_srcptr poly2, slong len2, slong
   start, nmod_t mod)
```

Computes the product of `(poly1, len1)` and `(poly2, len2)` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced. Assumes that `len1 >= len2 > 0`. Aliasing of inputs and output is not permitted.

```
void nmod_poly_mulhigh_classical(nmod_poly_t res, const
   nmod_poly_t poly1, const nmod_poly_t poly2, slong start)
```

Computes the product of `poly1` and `poly2` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced.

```
void _nmod_poly_mul_KS(mp_ptr out, mp_srcptr in1, slong
   len1, mp_srcptr in2, slong len2, mp_bitcnt_t bits,
   nmod_t mod)
```

Sets `res` to the product of `in1` and `in2` assuming the output coefficients are at most the given number of bits wide. If `bits` is set to 0 an appropriate value is computed automatically. Assumes that `len1 >= len2 > 0`.

```
void nmod_poly_mul_KS(nmod_poly_t res, const nmod_poly_t
   poly1, const nmod_poly_t poly2, mp_bitcnt_t bits)
```

Sets `res` to the product of `poly1` and `poly2` assuming the output coefficients are at most the given number of bits wide. If `bits` is set to 0 an appropriate value is computed automatically.

```
void _nmod_poly_mul_KS2(mp_ptr res, mp_srcptr op1, slong
   n1, mp_srcptr op2, slong n2, nmod_t mod)
```

Sets `res` to the product of `op1` and `op2`. Assumes that `len1 >= len2 > 0`.

```
void nmod_poly_mul_KS2(nmod_poly_t res, const nmod_poly_t
   poly1, const nmod_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _nmod_poly_mul_KS4(mp_ptr res, mp_srcptr op1, slong
   n1, mp_srcptr op2, slong n2, nmod_t mod)
```

Sets `res` to the product of `op1` and `op2`. Assumes that `len1 >= len2 > 0`.

```
void nmod_poly_mul_KS4(nmod_poly_t res, const nmod_poly_t
   poly1, const nmod_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _nmod_poly_mullow_KS(mp_ptr out, mp_srcptr in1, slong
   len1, mp_srcptr in2, slong len2, mp_bitcnt_t bits, slong
   n, nmod_t mod)
```

Sets `out` to the low n coefficients of `in1` of length `len1` times `in2` of length `len2`. The output must have space for `n` coefficients. We assume that `len1 >= len2 > 0` and that `0 < n <= len1 + len2 - 1`.

```
void nmod_poly_mullow_KS(nmod_poly_t res, const nmod_poly_t
   poly1, const nmod_poly_t poly2, mp_bitcnt_t bits, slong
   n)
```

Set `res` to the low n coefficients of `in1` of length `len1` times `in2` of length `len2`.

```
void _nmod_poly_mul(mp_ptr res, mp_srcptr poly1, slong
   len1, mp_srcptr poly2, slong len2, nmod_t mod)
```

Sets `res` to the product of `poly1` of length `len1` and `poly2` of length `len2`. Assumes `len1 >= len2 > 0`. No aliasing is permitted between the inputs and the output.

```
void nmod_poly_mul(nmod_poly_t res, const nmod_poly_t poly,
   const nmod_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _nmod_poly_mullow(mp_ptr res, mp_srcptr poly1, slong
   len1, mp_srcptr poly2, slong len2, slong n, nmod_t mod)
```

Sets `res` to the first `n` coefficients of the product of `poly1` of length `len1` and `poly2` of length `len2`. It is assumed that `0 < n <= len1 + len2 - 1` and that `len1 >= len2 > 0`. No aliasing of inputs and output is permitted.

```
void nmod_poly_mullow(nmod_poly_t res, const nmod_poly_t
   poly1, const nmod_poly_t poly2, slong trunc)
```

Sets `res` to the first `trunc` coefficients of the product of `poly1` and `poly2`.

```
void _nmod_poly_mulhigh(mp_ptr res, mp_srcptr poly1, slong
   len1, mp_srcptr poly2, slong len2, slong n, nmod_t mod)
```

Sets all but the low n coefficients of `res` to the corresponding coefficients of the product of `poly1` of length `len1` and `poly2` of length `len2`, the other coefficients being arbitrary. It is assumed that `len1 >= len2 > 0` and that `0 < n <= len1 + len2 - 1`. Aliasing of inputs and output is not permitted.

```
void nmod_poly_mulhigh(nmod_poly_t res, const nmod_poly_t
   poly1, const nmod_poly_t poly2, slong n)
```

Sets all but the low n coefficients of `res` to the corresponding coefficients of the product of `poly1` and `poly2`, the remaining coefficients being arbitrary.

```
void _nmod_poly_mulmod(mp_ptr res, mp_srcptr poly1, slong
   len1, mp_srcptr poly2, slong len2, mp_srcptr f, slong
   lenf, nmod_t mod)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_nmod_poly_mul` instead.

Aliasing of `f` and `res` is not permitted.

```
void nmod_poly_mulmod(nmod_poly_t res, const nmod_poly_t
   poly1, const nmod_poly_t poly2, const nmod_poly_t f)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

```
void _nmod_poly_mulmod_preinv(mp_ptr res, mp_srcptr poly1,
   slong len1, mp_srcptr poly2, slong len2, mp_srcptr f,
   slong lenf, mp_srcptr finv, slong lenfinv, nmod_t mod)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `finv` is the inverse of the reverse of `f` mod `x^lenf`. It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. It is required that `len1 < lenf` and `len2 < lenf`. Otherwise, simply use `_nmod_poly_mul` instead.

Aliasing of `f` or `finv` and `res` is not permitted.

```
void nmod_poly_mulmod_preinv(nmod_poly_t res, const
   nmod_poly_t poly1, const nmod_poly_t poly2, const
   nmod_poly_t f, const nmod_poly_t finv)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`. `finv` is the inverse of the reverse of `f`. It is required that `poly1` and `poly2` are reduced modulo `f`.

30.18 Powering

```
void _nmod_poly_pow_binexp(mp_ptr res, mp_srcptr poly,
   slong len, ulong e, nmod_t mod)
```

Raises `poly` of length `len` to the power `e` and sets `res` to the result. We require that `res` has enough space for `(len - 1)*e + 1` coefficients. Assumes that `len > 0`, `e > 1`. Aliasing is not permitted. Uses the binary exponentiation method.

```
void nmod_poly_pow_binexp(nmod_poly_t res, const
   nmod_poly_t poly, ulong e)
```

Raises `poly` to the power `e` and sets `res` to the result. Uses the binary exponentiation method.

```
void _nmod_poly_pow(mp_ptr res, mp_srcptr poly, slong len,
   ulong e, nmod_t mod)
```

Raises `poly` of length `len` to the power `e` and sets `res` to the result. We require that `res` has enough space for `(len - 1)*e + 1` coefficients. Assumes that `len > 0`, `e > 1`. Aliasing is not permitted.

```
void nmod_poly_pow(nmod_poly_t res, const nmod_poly_t poly,
   ulong e)
```

Raises `poly` to the power `e` and sets `res` to the result.

```
void _nmod_poly_pow_trunc_binexp(mp_ptr res, mp_srcptr
   poly, ulong e, slong trunc, nmod_t mod)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc > 0` and that `e > 1`. Aliasing is not permitted. Uses the binary exponentiation method.

```
void nmod_poly_pow_trunc_binexp(nmod_poly_t res, const
   nmod_poly_t poly, ulong e, slong trunc)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation. Uses the binary exponentiation method.

```
void _nmod_poly_pow_trunc(mp_ptr res, mp_srcptr poly, ulong
   e, slong trunc, nmod_t mod)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc > 0` and that `e > 1`. Aliasing is not permitted.

```
void nmod_poly_pow_trunc(nmod_poly_t res, const nmod_poly_t
   poly, ulong e, slong trunc)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation.

```
void _nmod_poly_powmod_ui_binexp(mp_ptr res, mp_srcptr
   poly, ulong e, mp_srcptr f, slong lenf, nmod_t mod)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void nmod_poly_powmod_ui_binexp(nmod_poly_t res, const
   nmod_poly_t poly, ulong e, const nmod_poly_t f)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void _nmod_poly_powmod_ui_binexp_preinv (mp_ptr res,
   mp_srcptr poly, ulong e, mp_srcptr f, slong lenf,
   mp_srcptr finv, slong lenfinv, nmod_t mod)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void nmod_poly_powmod_ui_binexp_preinv(nmod_poly_t res,
    const nmod_poly_t poly, ulong e, const nmod_poly_t f,
    const nmod_poly_t finv)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _nmod_poly_powmod_x_ui_preinv (mp_ptr res, ulong e,
    mp_srcptr f, slong lenf, mp_srcptr finv, slong lenfinv,
    nmod_t mod)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 2`. The output `res` must have room for `lenf - 1` coefficients.

```
void nmod_poly_powmod_x_ui_preinv(nmod_poly_t res, ulong e,
    const nmod_poly_t f, const nmod_poly_t finv)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _nmod_poly_powmod_mpz_binexp(mp_ptr res, mp_srcptr
    poly, mpz_srcptr e, mp_srcptr f, slong lenf, nmod_t mod)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void nmod_poly_powmod_mpz_binexp(nmod_poly_t res, const
    nmod_poly_t poly, mpz_srcptr e, const nmod_poly_t f)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void _nmod_poly_powmod_mpz_binexp_preinv (mp_ptr res,
    mp_srcptr poly, mpz_srcptr e, mp_srcptr f, slong lenf,
    mp_srcptr finv, slong lenfinv, nmod_t mod)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void nmod_poly_powmod_mpz_binexp_preinv(nmod_poly_t res,
    const nmod_poly_t poly, mpz_srcptr e, const nmod_poly_t
    f, const nmod_poly_t finv)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

30.19 Division

```
void _nmod_poly_divrem_basecase(mp_ptr Q, mp_ptr R, mp_ptr
    W, mp_srcptr A, slong A_len, mp_srcptr B, slong B_len,
    nmod_t mod)
```

Finds Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$. If $\operatorname{len}(B) = 0$ an exception is raised. We require that `W` is temporary space of `NMOD_DIVREM_BC_ITCH(A_len, B_len, mod)` coefficients.

```
void nmod_poly_divrem_basecase(nmod_poly_t Q, nmod_poly_t
   R, const nmod_poly_t A, const nmod_poly_t B)
```

Finds Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$. If $\operatorname{len}(B) = 0$ an exception is raised.

```
void _nmod_poly_div_basecase(mp_ptr Q, mp_ptr W, mp_srcptr
   A, slong A_len, mp_srcptr B, slong B_len, nmod_t mod);
```

Notionally finds polynomials Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$, but returns only `Q`. If $\operatorname{len}(B) = 0$ an exception is raised. We require that `W` is temporary space of `NMOD_DIV_BC_ITCH(A_len, B_len, mod)` coefficients.

```
void nmod_poly_div_basecase(nmod_poly_t Q, const
   nmod_poly_t A, const nmod_poly_t B);
```

Notionally finds polynomials Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$, but returns only `Q`. If $\operatorname{len}(B) = 0$ an exception is raised.

```
void _nmod_poly_divrem_divconquer_recursive(mp_ptr Q,
   mp_ptr BQ, mp_ptr W, mp_ptr V, mp_srcptr A, mp_srcptr B,
   slong lenB, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R)$ less than `lenB`, where `A` is of length `2 * lenB - 1` and `B` is of length `lenB`. Sets `BQ` to the low `lenB - 1` coefficients of `B * Q`. We require that `Q` have space for `lenB` coefficients, that `W` be temporary space of size `lenB - 1` and `V` be temporary space for a number of coefficients computed by `NMOD_DIVREM_DC_ITCH(lenB, mod)`.

```
void _nmod_poly_divrem_divconquer(mp_ptr Q, mp_ptr R,
   mp_srcptr A, slong lenA, mp_srcptr B, slong lenB, nmod_t
   mod)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`. We require that `Q` have space for `lenA - lenB + 1` coefficients.

```
void nmod_poly_divrem_divconquer(nmod_poly_t Q, nmod_poly_t
   R, const nmod_poly_t A, const nmod_poly_t B)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$.

```
void _nmod_poly_divrem_q0(mp_ptr Q, mp_ptr R, mp_srcptr A,
   mp_srcptr B, slong lenA, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$, where $\operatorname{len}(A) = \operatorname{len}(B) > 0$.

Requires that Q and R have space for 1 and $\operatorname{len}(B) - 1$ coefficients, respectively.

Does not support aliasing or zero-padding.

```
void _nmod_poly_divrem_q1(mp_ptr Q, mp_ptr R, mp_srcptr A,
   slong lenA, mp_srcptr B, slong lenB, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$, where $\operatorname{len}(A) = \operatorname{len}(B) + 1 \geq \operatorname{len}(B) > 0$.

Requires that Q and R have space for $\operatorname{len}(A) - \operatorname{len}(B) + 1$ and $\operatorname{len}(B) - 1$ coefficients, respectively.

Does not support aliasing or zero-padding.

```
void _nmod_poly_divrem(mp_ptr Q, mp_ptr R, mp_srcptr A,
   slong lenA, mp_srcptr B, slong lenB, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`. We require that `Q` have space for `lenA - lenB + 1` coefficients.

```
void nmod_poly_divrem(nmod_poly_t Q, nmod_poly_t R, const
   nmod_poly_t A, const nmod_poly_t B)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$.

```
void _nmod_poly_div_divconquer_recursive(mp_ptr Q, mp_ptr
   W, mp_ptr V, mp_srcptr A, mp_srcptr B, slong lenB,
   nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R)$ less than `lenB`, where `A` is of length `2 * lenB - 1` and `B` is of length `lenB`. We require that `Q` have space for `lenB` coefficients and that `W` be temporary space of size `lenB - 1` and `V` be temporary space for a number of coefficients computed by `NMOD_DIV_DC_ITCH(lenB, mod)`.

```
void _nmod_poly_div_divconquer(mp_ptr Q, mp_srcptr A, slong
   lenA, mp_srcptr B, slong lenB, nmod_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\operatorname{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`, but returns only `Q`. We require that `Q` have space for `lenA - lenB + 1` coefficients.

```
void nmod_poly_div_divconquer(nmod_poly_t Q, const
   nmod_poly_t A, const nmod_poly_t B)
```

Notionally computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$, but returns only Q.

```
void _nmod_poly_div(mp_ptr Q, mp_srcptr A, slong lenA,
   mp_srcptr B, slong lenB, nmod_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\operatorname{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`, but returns only `Q`. We require that `Q` have space for `lenA - lenB + 1` coefficients.

```
void nmod_poly_div(nmod_poly_t Q, const nmod_poly_t A,
   const nmod_poly_t B)
```

Computes the quotient Q on polynomial division of A and B.

```
void _nmod_poly_rem_basecase(mp_ptr R, mp_ptr W, mp_srcptr
   A, slong lenA, mp_srcptr B, slong lenB, nmod_t mod)
```

```
void nmod_poly_rem_basecase(nmod_poly_t R, const
   nmod_poly_t A, const nmod_poly_t B)
```

```
void _nmod_poly_rem_q1(mp_ptr R, mp_srcptr A, slong lenA,
   mp_srcptr B, slong lenB, nmod_t mod)
```

Notationally, computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, where $\text{len}(A) = \text{len}(B) + 1 \geq \text{len}(B) > 0$, but returns only the remainder.

Requires that R has space for $\text{len}(B) - 1$ coefficients, respectively.

Does not support aliasing or zero-padding.

```
void _nmod_poly_rem(mp_ptr R, mp_srcptr A, slong lenA,
   mp_srcptr B, slong lenB, nmod_t mod)
```

Computes the remainder R on polynomial division of A by B.

```
void nmod_poly_rem(nmod_poly_t R, const nmod_poly_t A,
   const nmod_poly_t B)
```

Computes the remainder R on polynomial division of A by B.

```
void _nmod_poly_inv_series_basecase(mp_ptr Qinv, mp_srcptr
   Q, slong n, nmod_t mod)
```

Given `Q` of length `n` whose leading coefficient is invertible modulo the given modulus, finds a polynomial `Qinv` of length `n` such that the top `n` coefficients of the product `Q * Qinv` is x^{n-1}. Requires that `n > 0`. This function can be viewed as inverting a power series.

```
void nmod_poly_inv_series_basecase(nmod_poly_t Qinv, const
   nmod_poly_t Q, slong n)
```

Given `Q` of length at least `n` find `Qinv` of length `n` such that the top `n` coefficients of the product `Q * Qinv` is x^{n-1}. An exception is raised if `n = 0` or if the length of `Q` is less than `n`. The leading coefficient of `Q` must be invertible modulo the modulus of `Q`. This function can be viewed as inverting a power series.

```
void _nmod_poly_inv_series_newton(mp_ptr Qinv, mp_srcptr Q,
   slong n, nmod_t mod)
```

Given `Q` of length `n` whose constant coefficient is invertible modulo the given modulus, find a polynomial `Qinv` of length `n` such that `Q * Qinv` is `1` modulo x^n. Requires `n > 0`. This function can be viewed as inverting a power series via Newton iteration.

```
void nmod_poly_inv_series_newton(nmod_poly_t Qinv, const
   nmod_poly_t Q, slong n)
```

Given `Q` find `Qinv` such that `Q * Qinv` is `1` modulo x^n. The constant coefficient of `Q` must be invertible modulo the modulus of `Q`. An exception is raised if this is not the case or if `n = 0`. This function can be viewed as inverting a power series via Newton iteration.

```
void _nmod_poly_inv_series(mp_ptr Qinv, mp_srcptr Q, slong
   n, nmod_t mod)
```

Given `Q` of length `n` whose constant coefficient is invertible modulo the given modulus, find a polynomial `Qinv` of length `n` such that `Q * Qinv` is `1` modulo x^n. Requires `n > 0`. This function can be viewed as inverting a power series.

```
void nmod_poly_inv_series(nmod_poly_t Qinv, const
   nmod_poly_t Q, slong n)
```

Given `Q` find `Qinv` such that `Q * Qinv` is `1` modulo x^n. The constant coefficient of `Q` must be invertible modulo the modulus of `Q`. An exception is raised if this is not the case or if `n = 0`. This function can be viewed as inverting a power series.

```
void _nmod_poly_div_series(mp_ptr Q, mp_srcptr A, mp_srcptr
   B, slong n, nmod_t mod)
```

Given polynomials `A` and `B` of length `n`, finds the polynomial `Q` of length `n` such that `Q * B = A` modulo x^n. We assume `n > 0` and that the constant coefficient of `B` is invertible modulo the given modulus. The polynomial `Q` must have space for `n` coefficients.

```
void nmod_poly_div_series(nmod_poly_t Q, const nmod_poly_t
   A, const nmod_poly_t B, slong n)
```

Given polynomials `A` and `B` considered modulo `n`, finds the polynomial `Q` of length at most `n` such that `Q * B = A` modulo x^n. We assume `n > 0` and that the constant coefficient of `B` is invertible modulo the modulus. An exception is raised if `n == 0` or the constant coefficient of `B` is zero.

```
void _nmod_poly_div_newton(mp_ptr Q, mp_srcptr A, slong
   Alen, mp_srcptr B, slong Blen, nmod_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\operatorname{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`, but return only Q.

We require that Q have space for `lenA - lenB + 1` coefficients and assume that the leading coefficient of B is a unit.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void nmod_poly_div_newton(nmod_poly_t Q, const nmod_poly_t
   A, const nmod_poly_t B)
```

Notionally computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$, but returns only Q.

We assume that the leading coefficient of B is a unit.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void _nmod_poly_div_newton_n_preinv (mp_ptr Q, mp_srcptr A,
   slong lenA, mp_srcptr B, slong lenB, mp_srcptr Binv,
   slong lenBinv, nmod_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\operatorname{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`, but return only Q.

We require that Q have space for `lenA - lenB + 1` coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\operatorname{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void nmod_poly_div_newton_n_preinv (nmod_poly_t Q, const
   nmod_poly_t A, const nmod_poly_t B, const nmod_poly_t
   Binv)
```

Notionally computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$, but returns only Q.

We assume that the leading coefficient of B is a unit and that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void _nmod_poly_divrem_newton(mp_ptr Q, mp_ptr R, mp_srcptr
   A, slong Alen, mp_srcptr B, slong Blen, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with len(R) less than `lenB`, where A is of length `lenA` and B is of length `lenB`. We require that Q have space for `lenA - lenB + 1` coefficients. The algorithm used is to call `div_newton()` and then multiply out and compute the remainder.

```
void nmod_poly_divrem_newton(nmod_poly_t Q, nmod_poly_t R,
   const nmod_poly_t A, const nmod_poly_t B)
```

Computes Q and R such that $A = BQ + R$ with len(R) < len(B). The algorithm used is to call `div_newton()` and then multiply out and compute the remainder.

```
void _nmod_poly_divrem_newton_n_preinv (mp_ptr Q, mp_ptr R,
   mp_srcptr A, slong lenA, mp_srcptr B, slong lenB,
   mp_srcptr Binv, slong lenBinv, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with len(R) less than `lenB`, where A is of length `lenA` and B is of length `lenB`. We require that Q have space for `lenA - lenB + 1` coefficients. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$. The algorithm used is to call `div_newton_n_preinv()` and then multiply out and compute the remainder.

```
void nmod_poly_divrem_newton_n_preinv(nmod_poly_t Q,
   nmod_poly_t R, const nmod_poly_t A, const nmod_poly_t B,
   const nmod_poly_t Binv)
```

Computes Q and R such that $A = BQ + R$ with len(R) < len(B). We assume $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to call `div_newton_n()` and then multiply out and compute the remainder.

```
mp_limb_t _nmod_poly_div_root(mp_ptr Q, mp_srcptr A, slong
   len, mp_limb_t c, nmod_t mod)
```

Sets `(Q, len-1)` to the quotient of `(A, len)` on division by $(x - c)$, and returns the remainder, equal to the value of A evaluated at c. A and Q are allowed to be the same, but may not overlap partially in any other way.

```
mp_limb_t nmod_poly_div_root(nmod_poly_t Q, const
   nmod_poly_t A, mp_limb_t c)
```

Sets Q to the quotient of A on division by $(x - c)$, and returns the remainder, equal to the value of A evaluated at c.

30.20 Derivative and integral

```
void _nmod_poly_derivative(mp_ptr x_prime, mp_srcptr x,
   slong len, nmod_t mod)
```

Sets the first `len - 1` coefficients of `x_prime` to the derivative of `x` which is assumed to be of length `len`. It is assumed that `len > 0`.

```
void nmod_poly_derivative(nmod_poly_t x_prime, const
    nmod_poly_t x)
```

Sets `x_prime` to the derivative of `x`.

```
void _nmod_poly_integral(mp_ptr x_int, mp_srcptr x, slong
    len, nmod_t mod)
```

Set the first `len` coefficients of `x_int` to the integral of `x` which is assumed to be of length `len - 1`. The constant term of `x_int` is set to zero. It is assumed that `len > 0`. The result is only well-defined if the modulus is a prime number strictly larger than the degree of `x`.

```
void nmod_poly_integral(nmod_poly_t x_int, const
    nmod_poly_t x)
```

Set `x_int` to the indefinite integral of `x` with constant term zero. The result is only well-defined if the modulus is a prime number strictly larger than the degree of `x`.

30.21 Evaluation

```
mp_limb_t _nmod_poly_evaluate_nmod(mp_srcptr poly, slong
    len, mp_limb_t c, nmod_t mod)
```

Evaluates `poly` at the value `c` and reduces modulo the given modulus of `poly`. The value `c` should be reduced modulo the modulus. The algorithm used is Horner's method.

```
mp_limb_t nmod_poly_evaluate_nmod(nmod_poly_t poly,
    mp_limb_t c)
```

Evaluates `poly` at the value `c` and reduces modulo the modulus of `poly`. The value `c` should be reduced modulo the modulus. The algorithm used is Horner's method.

30.22 Multipoint evaluation

```
void _nmod_poly_evaluate_nmod_vec_iter(mp_ptr ys, mp_srcptr
    poly, slong len, mp_srcptr xs, slong n, nmod_t mod)
```

Evaluates (`coeffs`, `len`) at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

Uses Horner's method iteratively.

```
void nmod_poly_evaluate_nmod_vec_iter(mp_ptr ys, const
    nmod_poly_t poly, mp_srcptr xs, slong n)
```

Evaluates `poly` at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

Uses Horner's method iteratively.

```
void _nmod_poly_evaluate_nmod_vec_fast_precomp(mp_ptr vs,
    mp_srcptr poly, slong plen, const mp_ptr * tree, slong
    len, nmod_t mod)
```

Evaluates (`poly`, `plen`) at the `len` values given by the precomputed subproduct tree `tree`.

```
void _nmod_poly_evaluate_nmod_vec_fast(mp_ptr ys, mp_srcptr
   poly, slong len, mp_srcptr xs, slong n, nmod_t mod)
```

Evaluates (`coeffs`, `len`) at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

Uses fast multipoint evaluation, building a temporary subproduct tree.

```
void nmod_poly_evaluate_nmod_vec_fast(mp_ptr ys, const
   nmod_poly_t poly, mp_srcptr xs, slong n)
```

Evaluates `poly` at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

Uses fast multipoint evaluation, building a temporary subproduct tree.

```
void _nmod_poly_evaluate_nmod_vec(mp_ptr ys, mp_srcptr
   poly, slong len, mp_srcptr xs, slong n, nmod_t mod)
```

Evaluates (`poly`, `len`) at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

```
void nmod_poly_evaluate_nmod_vec(mp_ptr ys, const
   nmod_poly_t poly, mp_srcptr xs, slong n)
```

Evaluates `poly` at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

30.23 Interpolation

```
void _nmod_poly_interpolate_nmod_vec(mp_ptr poly, mp_srcptr
   xs, mp_srcptr ys, slong n, nmod_t mod)
```

Sets `poly` to the unique polynomial of length at most `n` that interpolates the `n` given evaluation points `xs` and values `ys`. If the interpolating polynomial is shorter than length `n`, the leading coefficients are set to zero.

The values in `xs` and `ys` should be reduced modulo the modulus, and all `xs` must be distinct. Aliasing between `poly` and `xs` or `ys` is not allowed.

```
void nmod_poly_interpolate_nmod_vec(nmod_poly_t poly,
   mp_srcptr xs, mp_srcptr ys, slong n)
```

Sets `poly` to the unique polynomial of length `n` that interpolates the `n` given evaluation points `xs` and values `ys`. The values in `xs` and `ys` should be reduced modulo the modulus, and all `xs` must be distinct.

```
void _nmod_poly_interpolation_weights(mp_ptr w, const
   mp_ptr * tree, slong len, nmod_t mod)
```

Sets `w` to the barycentric interpolation weights for fast Lagrange interpolation with respect to a given subproduct tree.

```
void _nmod_poly_interpolate_nmod_vec_fast_precomp(mp_ptr
   poly, mp_srcptr ys, const mp_ptr * tree, mp_srcptr
   weights, slong len, nmod_t mod)
```

Performs interpolation using the fast Lagrange interpolation algorithm, generating a temporary subproduct tree.

The function values are given as `ys`. The function takes a precomputed subproduct tree `tree` and barycentric interpolation weights `weights` corresponding to the roots.

```
void _nmod_poly_interpolate_nmod_vec_fast(mp_ptr poly,
   mp_srcptr xs, mp_srcptr ys, slong n, nmod_t mod)
```

Performs interpolation using the fast Lagrange interpolation algorithm, generating a temporary subproduct tree.

```
void nmod_poly_interpolate_nmod_vec_fast(nmod_poly_t poly,
   mp_srcptr xs, mp_srcptr ys, slong n)
```

Performs interpolation using the fast Lagrange interpolation algorithm, generating a temporary subproduct tree.

```
void _nmod_poly_interpolate_nmod_vec_newton(mp_ptr poly,
   mp_srcptr xs, mp_srcptr ys, slong n, nmod_t mod)
```

Forms the interpolating polynomial in the Newton basis using the method of divided differences and then converts it to monomial form.

```
void nmod_poly_interpolate_nmod_vec_newton(nmod_poly_t
   poly, mp_srcptr xs, mp_srcptr ys, slong n)
```

Forms the interpolating polynomial in the Newton basis using the method of divided differences and then converts it to monomial form.

```
void _nmod_poly_interpolate_nmod_vec_barycentric(mp_ptr
   poly, mp_srcptr xs, mp_srcptr ys, slong n, nmod_t mod)
```

Forms the interpolating polynomial using a naive implementation of the barycentric form of Lagrange interpolation.

```
void nmod_poly_interpolate_nmod_vec_barycentric(nmod_poly_t
   poly, mp_srcptr xs, mp_srcptr ys, slong n)
```

Forms the interpolating polynomial using a naive implementation of the barycentric form of Lagrange interpolation.

30.24 Composition

```
void _nmod_poly_compose_horner(mp_ptr res, mp_srcptr poly1,
   slong len1, mp_srcptr poly2, slong len2, nmod_t mod)
```

Composes `poly1` of length `len1` with `poly2` of length `len2` and sets `res` to the result, i.e. evaluates `poly1` at `poly2`. The algorithm used is Horner's algorithm. We require that `res` have space for `(len1 - 1)*(len2 - 1)+ 1` coefficients. It is assumed that `len1 > 0` and `len2 > 0`.

```
void nmod_poly_compose_horner(nmod_poly_t res, const
   nmod_poly_t poly1, const nmod_poly_t poly2)
```

Composes `poly1` with `poly2` and sets `res` to the result, i.e. evaluates `poly1` at `poly2`. The algorithm used is Horner's algorithm.

```
void _nmod_poly_compose_divconquer(mp_ptr res, mp_srcptr
   poly1, slong len1, mp_srcptr poly2, slong len2, nmod_t
   mod)
```

Composes `poly1` of length `len1` with `poly2` of length `len2` and sets `res` to the result, i.e. evaluates `poly1` at `poly2`. The algorithm used is the divide and conquer algorithm. We require that `res` have space for `(len1 - 1)*(len2 - 1)+ 1` coefficients. It is assumed that `len1 > 0` and `len2 > 0`.

```
void nmod_poly_compose_divconquer(nmod_poly_t res, const
    nmod_poly_t poly1, const nmod_poly_t poly2)
```

Composes `poly1` with `poly2` and sets `res` to the result, i.e. evaluates `poly1` at `poly2`. The algorithm used is the divide and conquer algorithm.

```
void _nmod_poly_compose(mp_ptr res, mp_srcptr poly1, slong
    len1, mp_srcptr poly2, slong len2, nmod_t mod)
```

Composes `poly1` of length `len1` with `poly2` of length `len2` and sets `res` to the result, i.e. evaluates `poly1` at `poly2`. We require that `res` have space for `(len1 - 1)*(len2 - 1)+ 1` coefficients. It is assumed that `len1 > 0` and `len2 > 0`.

```
void nmod_poly_compose(nmod_poly_t res, const nmod_poly_t
    poly1, const nmod_poly_t poly2)
```

Composes `poly1` with `poly2` and sets `res` to the result, that is, evaluates `poly1` at `poly2`.

30.25 Taylor shift

```
void _nmod_poly_taylor_shift_horner(mp_ptr poly, mp_limb_t
    c, slong len, nmod_t mod)
```

Performs the Taylor shift composing `poly` by $x + c$ in-place. Uses an efficient version Horner's rule.

```
void nmod_poly_taylor_shift_horner(nmod_poly_t g, const
    nmod_poly_t f, mp_limb_t c)
```

Performs the Taylor shift composing `f` by $x + c$.

```
void _nmod_poly_taylor_shift_convolution(mp_ptr poly,
    mp_limb_t c, slong len, nmod_t mod)
```

Performs the Taylor shift composing `poly` by $x+c$ in-place. Writes the composition as a single convolution with cost $O(M(n))$. We require that the modulus is a prime at least as large as the length.

```
void nmod_poly_taylor_shift_convolution(nmod_poly_t g,
    const nmod_poly_t f, mp_limb_t c)
```

Performs the Taylor shift composing `f` by $x + c$. Writes the composition as a single convolution with cost $O(M(n))$. We require that the modulus is a prime at least as large as the length.

```
void _nmod_poly_taylor_shift(mp_ptr poly, mp_limb_t c,
    slong len, nmod_t mod)
```

Performs the Taylor shift composing `poly` by $x+c$ in-place. We require that the modulus is a prime.

```
void nmod_poly_taylor_shift(nmod_poly_t g, const
    nmod_poly_t f, mp_limb_t c)
```

Performs the Taylor shift composing `f` by $x+c$. We require that the modulus is a prime.

30.26 Modular composition

```
void _nmod_poly_compose_mod_horner(mp_ptr res, mp_srcptr f,
    slong lenf, mp_srcptr g, mp_srcptr h, slong lenh, nmod_t
    mod)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void nmod_poly_compose_mod_horner(nmod_poly_t res, const
    nmod_poly_t f, const nmod_poly_t g, const nmod_poly_t h)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero. The algorithm used is Horner's rule.

```
void _nmod_poly_compose_mod_brent_kung(mp_ptr res,
    mp_srcptr f, slong lenf, mp_srcptr g, mp_srcptr h, slong
    lenh, nmod_t mod)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void nmod_poly_compose_mod_brent_kung(nmod_poly_t res,
    const nmod_poly_t f, const nmod_poly_t g, const
    nmod_poly_t h)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. The algorithm used is the Brent-Kung matrix algorithm.

```
void _nmod_poly_compose_mod_brent_kung_preinv(mp_ptr res,
    mp_srcptr f, slong lenf, mp_srcptr g, mp_srcptr h, slong
    lenh, mp_srcptr hinv, slong lenhinv, nmod_t mod)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void nmod_poly_compose_mod_brent_kung_preinv(nmod_poly_t
    res, const nmod_poly_t f, const nmod_poly_t g, const
    nmod_poly_t h, const nmod_poly_t hinv)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The algorithm used is the Brent-Kung matrix algorithm.

```
void _nmod_poly_reduce_matrix_mod_poly (nmod_mat_t A, const
    nmod_mat_t B, const nmod_poly_t f)
```

Sets the ith row of `A` to the reduction of the ith row of B modulo f for $i = 1, \ldots, \sqrt{\deg(f)}$. We require B to be at least a $\sqrt{\deg(f)} \times \deg(f)$ matrix and f to be nonzero.

```
void _nmod_poly_precompute_matrix (nmod_mat_t A, mp_srcptr
    f, mp_srcptr g, slong leng, mp_srcptr ginv, slong
    lenginv, nmod_t mod)
```

Sets the ith row of `A` to f^i modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of `g` and g to be nonzero.

```
void nmod_poly_precompute_matrix (nmod_mat_t A, const
    nmod_poly_t f, const nmod_poly_t g, const nmod_poly_t
    ginv)
```

Sets the ith row of `A` to f^i modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of `g`.

```
void
    _nmod_poly_compose_mod_brent_kung_precomp_preinv(mp_ptr
    res, mp_srcptr f, slong lenf, const nmod_mat_t A,
    mp_srcptr h, slong h, mp_srcptr hinv, slong lenhinv,
    nmod_t mod)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero. We require that the ith row of A contains g^i for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void
    nmod_poly_compose_mod_brent_kung_precomp_preinv(nmod_poly_t
    res, const nmod_poly_t f, const nmod_mat_t A, const
    nmod_poly_t h, const nmod_poly_t hinv)
```

Sets `res` to the composition $f(g)$ modulo h. We require that the ith row of A contains g^i for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo h for fixed g and h.

```
void _nmod_poly_compose_mod(mp_ptr res, mp_srcptr f, slong
    lenf, mp_srcptr g, mp_srcptr h, slong lenh, nmod_t mod)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```
void nmod_poly_compose_mod(nmod_poly_t res, const
    nmod_poly_t f, const nmod_poly_t g, const nmod_poly_t h)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero.

30.27 Greatest common divisor

```
slong _nmod_poly_gcd_euclidean(mp_ptr G, mp_srcptr A, slong
    lenA, mp_srcptr B, slong lenB, nmod_t mod)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= lenB > 0`. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for `lenB` coefficients.

```
void nmod_poly_gcd_euclidean(nmod_poly_t G, const
    nmod_poly_t A, const nmod_poly_t B)
```

Computes the GCD of A and B. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

```
slong _nmod_poly_hgcd(mp_ptr *M, slong *lenM, mp_ptr A,
    slong *lenA, mp_ptr B, slong *lenB, mp_srcptr a, slong
    lena, mp_srcptr b, slong lenb, nmod_t mod)
```

Computes the HGCD of a and b, that is, a matrix M, a sign σ and two polynomials A and B such that

$$(A, B)^t = \sigma M^{-1}(a, b)^t.$$

Assumes that $\operatorname{len}(a) > \operatorname{len}(b) > 0$.

Assumes that A and B have space of size at least $\operatorname{len}(a)$ and $\operatorname{len}(b)$, respectively. On exit, `*lenA` and `*lenB` will contain the correct lengths of A and B.

Assumes that `M[0]`, `M[1]`, `M[2]`, and `M[3]` each point to a vector of size at least $\operatorname{len}(a)$.

```
slong _nmod_poly_gcd_hgcd(mp_ptr G, mp_srcptr A, slong
    lenA, mp_srcptr B, slong lenB, nmod_t mod)
```

Computes the monic GCD of A and B, assuming that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$.

Assumes that G has space for $\operatorname{len}(B)$ coefficients and returns the length of G on output.

```
void nmod_poly_gcd_hgcd(nmod_poly_t G, const nmod_poly_t A,
    const nmod_poly_t B)
```

Computes the monic GCD of A and B using the HGCD algorithm.

As a special case, the GCD of two zero polynomials is defined to be the zero polynomial.

The time complexity of the algorithm is $\mathcal{O}(n \log^2 n)$. For further details, see [27].

```
slong _nmod_poly_gcd(mp_ptr G, mp_srcptr A, slong lenA,
    mp_srcptr B, slong lenB, nmod_t mod)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= lenB > 0`. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for `lenB` coefficients.

```
void nmod_poly_gcd(nmod_poly_t G, const nmod_poly_t A,
    const nmod_poly_t B)
```

Computes the GCD of A and B. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

```
slong _nmod_poly_xgcd_euclidean(mp_ptr G, mp_ptr S, mp_ptr
    T, mp_srcptr A, slong A_len, mp_srcptr B, slong B_len,
    nmod_t mod)
```

Computes the GCD of A and B together with cofactors S and T such that $SA+TB = G$. Returns the length of G.

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) \geq 1$ and $(\operatorname{len}(A), \operatorname{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that G have space for $\operatorname{len}(B)$ coefficients. Writes $\operatorname{len}(B) - 1$ and $\operatorname{len}(A) - 1$ coefficients to S and T, respectively. Note that, in fact, $\operatorname{len}(S) \leq \max(\operatorname{len}(B) - \operatorname{len}(G), 1)$ and $\operatorname{len}(T) \leq \max(\operatorname{len}(A) - \operatorname{len}(G), 1)$.

No aliasing of input and output operands is permitted.

```
void nmod_poly_xgcd_euclidean(nmod_poly_t G, nmod_poly_t S,
   nmod_poly_t T, const nmod_poly_t A, const nmod_poly_t B)
```

Computes the GCD of A and B. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

Polynomials `S` and `T` are computed such that `S*A + T*B = G`. The length of `S` will be at most `lenB` and the length of `T` will be at most `lenA`.

```
slong _nmod_poly_xgcd_hgcd(mp_ptr G, mp_ptr S, mp_ptr T,
   mp_srcptr A, slong A_len, mp_srcptr B, slong B_len,
   nmod_t mod)
```

Computes the GCD of A and B, where $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$, together with cofactors S and T such that $SA + TB = G$. Returns the length of G.

No attempt is made to make the GCD monic.

Requires that G have space for $\operatorname{len}(B)$ coefficients. Writes $\operatorname{len}(B) - 1$ and $\operatorname{len}(A) - 1$ coefficients to S and T, respectively. Note that, in fact, $\operatorname{len}(S) \leq \operatorname{len}(B) - \operatorname{len}(G)$ and $\operatorname{len}(T) \leq \operatorname{len}(A) - \operatorname{len}(G)$.

Both S and T must have space for at least 2 coefficients.

No aliasing of input and output operands is permitted.

```
void nmod_poly_xgcd_hgcd(nmod_poly_t G, nmod_poly_t S,
   nmod_poly_t T, const nmod_poly_t A, const nmod_poly_t B)
```

Computes the GCD of A and B. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

Polynomials `S` and `T` are computed such that `S*A + T*B = G`. The length of `S` will be at most `lenB` and the length of `T` will be at most `lenA`.

```
slong _nmod_poly_xgcd(mp_ptr G, mp_ptr S, mp_ptr T,
   mp_srcptr A, slong lenA, mp_srcptr B, slong lenB, nmod_t
   mod)
```

Computes the GCD of A and B, where $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$, together with cofactors S and T such that $SA + TB = G$. Returns the length of G.

No attempt is made to make the GCD monic.

Requires that G have space for $\operatorname{len}(B)$ coefficients. Writes $\operatorname{len}(B) - 1$ and $\operatorname{len}(A) - 1$ coefficients to S and T, respectively. Note that, in fact, $\operatorname{len}(S) \leq \operatorname{len}(B) - \operatorname{len}(G)$ and $\operatorname{len}(T) \leq \operatorname{len}(A) - \operatorname{len}(G)$.

No aliasing of input and output operands is permitted.

```
void nmod_poly_xgcd(nmod_poly_t G, nmod_poly_t S,
   nmod_poly_t T, const nmod_poly_t A, const nmod_poly_t B)
```

Computes the GCD of A and B. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

The polynomials `S` and `T` are set such that `S*A + T*B = G`. The length of `S` will be at most `lenB` and the length of `T` will be at most `lenA`.

```
mp_limb_t _nmod_poly_resultant_euclidean(mp_srcptr poly1,
   slong len1, mp_srcptr poly2, slong len2, nmod_t mod)
```

Returns the resultant of `(poly1, len1)` and `(poly2, len2)` using the Euclidean algorithm.

Assumes that `len1 >= len2 > 0`.

Asumes that the modulus is prime.

```
mp_limb_t nmod_poly_resultant_euclidean(const nmod_poly_t
   f, const nmod_poly_t g)
```

Computes the resultant of f and g using the Euclidean algorithm.

For two non-zero polynomials $f(x) = a_m x^m + \cdots + a_0$ and $g(x) = b_n x^n + \cdots + b_0$ of degrees m and n, the resultant is defined to be

$$a_m^n b_n^m \prod_{(x,y):f(x)=g(y)=0} (x - y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

```
mp_limb_t _nmod_poly_resultant(mp_srcptr poly1, slong len1,
   mp_srcptr poly2, slong len2, nmod_t mod)
```

Returns the resultant of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `len1 >= len2 > 0`.

Asumes that the modulus is prime.

```
mp_limb_t nmod_poly_resultant(const nmod_poly_t f, const
   nmod_poly_t g)
```

Computes the resultant of f and g.

For two non-zero polynomials $f(x) = a_m x^m + \cdots + a_0$ and $g(x) = b_n x^n + \cdots + b_0$ of degrees m and n, the resultant is defined to be

$$a_m^n b_n^m \prod_{(x,y):f(x)=g(y)=0} (x - y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

```
slong _nmod_poly_gcdinv(mp_ptr G, mp_ptr S, mp_srcptr A,
   slong lenA, mp_srcptr B, slong lenB, const nmod_t mod)
```

Computes `(G, lenA)`, `(S, lenB-1)` such that $G \cong SA \pmod{B}$, returning the actual length of G.

Assumes that $0 < \operatorname{len}(A) < \operatorname{len}(B)$.

```
void nmod_poly_gcdinv(nmod_poly_t G, nmod_poly_t S, const
   nmod_poly_t A, const nmod_poly_t B)
```

Computes polynomials G and S, both reduced modulo B, such that $G \cong SA \pmod{B}$, where B is assumed to have $\operatorname{len}(B) \geq 2$.

In the case that $A = 0 \pmod{B}$, returns $G = S = 0$.

```
int _nmod_poly_invmod(mp_ptr A, mp_srcptr B, slong lenB,
   mp_srcptr P, slong lenP, const nmod_t mod)
```

Attempts to set `(A, lenP-1)` to the inverse of `(B, lenB)` modulo the polynomial `(P, lenP)`. Returns 1 if `(B, lenB)` is invertible and 0 otherwise.

Assumes that $0 < \text{len}(B) < \text{len}(P)$, and hence also $\text{len}(P) \geq 2$, but supports zero-padding in `(B, lenB)`.

Does not support aliasing.

Assumes that *mod* is a prime number.

```
int nmod_poly_invmod(nmod_poly_t A, const nmod_poly_t B,
    const nmod_poly_t P)
```

Attempts to set A to the inverse of B modulo P in the polynomial ring $(\mathbf{Z}/p\mathbf{Z})[X]$, where we assume that p is a prime number.

If $\deg(P) < 2$, raises an exception.

If the greatest common divisor of B and P is 1, returns 1 and sets A to the inverse of B. Otherwise, returns 0 and the value of A on exit is undefined.

30.28 Power series composition

```
void _nmod_poly_compose_series_horner(mp_ptr res, mp_srcptr
    poly1, slong len1, mp_srcptr poly2, slong len2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

Assumes that `len1, len2, n > 0`, that `len1, len2 <= n`, and that `(len1-1)* (len2-1)+ 1 <= n`, and that `res` has space for `n` coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses the Horner scheme.

```
void nmod_poly_compose_series_horner(nmod_poly_t res, const
    nmod_poly_t poly1, const nmod_poly_t poly2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

This implementation uses the Horner scheme.

```
void _nmod_poly_compose_series_brent_kung(mp_ptr res,
    mp_srcptr poly1, slong len1, mp_srcptr poly2, slong
    len2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

Assumes that `len1, len2, n > 0`, that `len1, len2 <= n`, and that `(len1-1)* (len2-1)+ 1 <= n`, and that `res` has space for `n` coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses Brent-Kung algorithm 2.1 [6].

```
void nmod_poly_compose_series_brent_kung(nmod_poly_t res,
    const nmod_poly_t poly1, const nmod_poly_t poly2, slong
    n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

This implementation uses Brent-Kung algorithm 2.1 [6].

```
void _nmod_poly_compose_series_divconquer(mp_ptr res,
    mp_srcptr poly1, slong len1, mp_srcptr poly2, slong
    len2, slong N, nmod_t mod)
```

Composes `poly1` of length ℓ_1 with `poly2` of length ℓ_2 modulo x^N and sets `res` to the result, i.e. evaluates `poly1` at `poly2`.

Writes $\min\{(\ell_1 - 1)(\ell_2 - 2) + 1, N\}$ coefficients to the vector `res`.

The algorithm used is the divide and conquer algorithm. It is assumed that $0 < \ell_1$ and $0 < \ell_2 \leq N$.

Does not support aliasing between the inputs and the output.

```
void nmod_poly_compose_series_divconquer(nmod_poly_t res,
    const nmod_poly_t poly1, const nmod_poly_t poly2, slong
    N)
```

Composes `poly1` with `poly2` modulo x^N and sets `res` to the result, i.e. evaluates `poly1` at `poly2`.

The algorithm used is the divide and conquer algorithm.

```
void _nmod_poly_compose_series(mp_ptr res, mp_srcptr poly1,
    slong len1, mp_srcptr poly2, slong len2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

Assumes that `len1, len2, n > 0`, that `len1, len2 <= n`, and that `(len1-1)* (len2-1)+ 1 <= n`, and that `res` has space for `n` coefficients. Does not support aliasing between any of the inputs and the output.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs.

```
void nmod_poly_compose_series(nmod_poly_t res, const
    nmod_poly_t poly1, const nmod_poly_t poly2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n, where the constant term of `poly2` is required to be zero.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs.

30.29 Power series reversion

```
void _nmod_poly_revert_series_lagrange(mp_ptr Qinv,
    mp_srcptr Q, slong n, nmod_t mod)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. The arguments must both have length `n` and may not be aliased.

It is required that $Q_0 = 0$ and that Q_1 as well as the integers $1, 2, \ldots, n-1$ are invertible modulo the modulus.

This implementation uses the Lagrange inversion formula.

```
void nmod_poly_revert_series_lagrange(nmod_poly_t Qinv,
    const nmod_poly_t Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$.

It is required that $Q_0 = 0$ and that Q_1 as well as the integers $1, 2, \ldots, n-1$ are invertible modulo the modulus.

This implementation uses the Lagrange inversion formula.

```
void _nmod_poly_revert_series_lagrange_fast(mp_ptr Qinv,
    mp_srcptr Q, slong n, nmod_t mod)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. The arguments must both have length `n` and may not be aliased.

It is required that $Q_0 = 0$ and that Q_1 as well as the integers $1, 2, \ldots, n-1$ are invertible modulo the modulus.

This implementation uses a reduced-complexity implementation of the Lagrange inversion formula.

```
void nmod_poly_revert_series_lagrange_fast(nmod_poly_t
    Qinv, const nmod_poly_t Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$.

It is required that $Q_0 = 0$ and that Q_1 as well as the integers $1, 2, \ldots, n-1$ are invertible modulo the modulus.

This implementation uses a reduced-complexity implementation of the Lagrange inversion formula.

```
void _nmod_poly_revert_series_newton(mp_ptr Qinv, mp_srcptr
    Q, slong n, nmod_t mod)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. The arguments must both have length `n` and may not be aliased.

It is required that $Q_0 = 0$ and that Q_1 as well as the integers $1, 2, \ldots, n-1$ are invertible modulo the modulus.

This implementation uses Newton iteration [6].

```
void nmod_poly_revert_series_newton(nmod_poly_t Qinv, const
    nmod_poly_t Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$.

It is required that $Q_0 = 0$ and that Q_1 as well as the integers $1, 2, \ldots, n-1$ are invertible modulo the modulus.

This implementation uses Newton iteration [6].

```
void _nmod_poly_revert_series(mp_ptr Qinv, mp_srcptr Q,
    slong n, nmod_t mod)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. The arguments must both have length `n` and may not be aliased.

It is required that $Q_0 = 0$ and that Q_1 as well as the integers $1, 2, \ldots, n-1$ are invertible modulo the modulus.

This implementation automatically chooses between the Lagrange inversion formula and Newton iteration based on the size of the input.

```
void nmod_poly_revert_series(nmod_poly_t Qinv, const
    nmod_poly_t Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$.

It is required that $Q_0 = 0$ and that Q_1 as well as the integers $1, 2, \ldots, n-1$ are invertible modulo the modulus.

This implementation automatically chooses between the Lagrange inversion formula and Newton iteration based on the size of the input.

30.30 Square roots

The series expansions for $\sqrt{h}$ and $1/\sqrt{h}$ are defined by means of the generalised binomial theorem

$$h^r = (1+y)^r = \sum_{k=0}^{\infty} \binom{r}{k} y^k.$$

It is assumed that h has constant term 1 and that the coefficients 2^{-k} exist in the coefficient ring (i.e. 2 must be invertible).

```
void _nmod_poly_invsqrt_series(mp_ptr g, mp_srcptr h, slong
    n, nmod_t mod)
```

Set the first n terms of g to the series expansion of $1/\sqrt{h}$. It is assumed that $n > 0$, that h has constant term 1 and that h is zero-padded as necessary to length n. Aliasing is not permitted.

```
void nmod_poly_invsqrt_series(nmod_poly_t g, const
    nmod_poly_t h, slong n)
```

Set g to the series expansion of $1/\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
void _nmod_poly_sqrt_series(mp_ptr g, mp_srcptr h, slong n,
    nmod_t mod)
```

Set the first n terms of g to the series expansion of $\sqrt{h}$. It is assumed that $n > 0$, that h has constant term 1 and that h is zero-padded as necessary to length n. Aliasing is not permitted.

```
void nmod_poly_sqrt_series(nmod_poly_t g, const nmod_poly_t
    h, slong n)
```

Set g to the series expansion of $\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
int _nmod_poly_sqrt(mp_ptr s, mp_srcptr p, slong n, nmod_t
    mod)
```

If `(p, n)` is a perfect square, sets `(s, n / 2 + 1)` to a square root of p and returns 1. Otherwise returns 0.

```
int nmod_poly_sqrt(nmod_poly_t s, const nmod_poly_t p)
```

If p is a perfect square, sets s to a square root of p and returns 1. Otherwise returns 0.

30.31 Transcendental functions

The elementary transcendental functions of a formal power series h are defined as

$$\exp(h(x)) = \sum_{k=0}^{\infty} \frac{(h(x))^k}{k!}$$

$$\log(h(x)) = \int_0^x \frac{h'(t)}{h(t)} dt$$

$$\operatorname{atan}(h(x)) = \int_0^x \frac{h'(t)}{1 + (h(t))^2} dt$$

$$\operatorname{atanh}(h(x)) = \int_0^x \frac{h'(t)}{1 - (h(t))^2} dt$$

$$\operatorname{asin}(h(x)) = \int_0^x \frac{h'(t)}{\sqrt{1 - (h(t))^2}} dt$$

$$\operatorname{asinh}(h(x)) = \int_0^x \frac{h'(t)}{\sqrt{1 + (h(t))^2}} dt$$

The functions sin, cos, tan, etc. are defined using standard inverse or functional relations.

The logarithm function assumes that h has constant term 1. All other functions assume that h has constant term 0.

All functions assume that the coefficient $1/k$ or $1/k!$ exists for all indices k. When computing to order $O(x^n)$, the modulus p must therefore be a prime satisfying $p \geq n$. Further, we always require that $p > 2$ in order to be able to multiply by $1/2$ for internal purposes.

If the input does not satisfy all these conditions, results are undefined.

Except where otherwise noted, functions are implemented with optimal (up to constants) complexity $O(M(n))$, where $M(n)$ is the cost of polynomial multiplication.

```
void _nmod_poly_log_series_monomial_ui(mp_ptr g, mp_limb_t
   c, ulong r, slong n, nmod_t mod)
```

Set $g = \log(1 + cx^r) + O(x^n)$. Assumes $n > 0$, $r > 0$, and that the coefficient is reduced by the modulus. Works efficiently in linear time.

```
void nmod_poly_log_series_monomial_ui(nmod_poly_t g,
   mp_limb_t c, ulong r, slong n)
```

Set $g = \log(1 + cx^r) + O(x^n)$. Works efficiently in linear time.

```
void _nmod_poly_log_series(mp_ptr g, mp_srcptr h, slong n,
   nmod_t mod)
```

Set $g = \log(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed.

```
void nmod_poly_log_series(nmod_poly_t g, const nmod_poly_t
   h, slong n)
```

Set $g = \log(h) + O(x^n)$. The case $h = 1 + cx^r$ is automatically detected and handled efficiently.

```
void _nmod_poly_exp_series_monomial_ui(mp_ptr g, mp_limb_t
    c, ulong r, slong n, nmod_t mod)
```

Set $g = \exp(cx^r) + O(x^n)$. Assumes $n > 0$, $r > 0$, and that the coefficient is reduced by the modulus. Works efficiently in linear time.

```
void nmod_poly_exp_series_monomial_ui(nmod_poly_t g,
    mp_limb_t c, ulong r, slong n)
```

Set $g = \exp(cx^r) + O(x^n)$. Works efficiently in linear time.

```
void _nmod_poly_exp_series_basecase(mp_ptr g, mp_srcptr h,
    slong hlen, slong n, nmod_t mod)
```

Set $g = \exp(h) + O(x^n)$ using a simple $O(n^2)$ algorithm. Assumes $n > 0$ and hlen > 0. Only the first hlen coefficients of h will be read. Aliasing of f and h is allowed.

```
void nmod_poly_exp_series_basecase(nmod_poly_t g, const
    nmod_poly_t h, slong n)
```

Set $g = \exp(h) + O(x^n)$ using a simple $O(n^2)$ algorithm.

```
void _nmod_poly_exp_series(mp_ptr g, mp_srcptr h, slong n,
    nmod_t mod)
```

Set $g = \exp(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is not allowed.

Uses Newton iteration (the version given in [15]). For small n, falls back to the basecase algorithm.

```
void  _nmod_poly_exp_expinv_series(mp_ptr f, mp_ptr g,
    mp_srcptr h, slong n, nmod_t mod)
```

Set $f = \exp(h) + O(x^n)$ and $g = \exp(-h) + O(x^n)$, more efficiently for large n than performing a separate inversion to obtain g. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing is not allowed.

Uses Newton iteration (the version given in [15]). For small n, falls back to the basecase algorithm.

```
void nmod_poly_exp_series(nmod_poly_t g, const nmod_poly_t
    h, slong n)
```

Set $g = \exp(h) + O(x^n)$. The case $h = cx^r$ is automatically detected and handled efficiently. Otherwise this function automatically uses the basecase algorithm for small n and Newton iteration otherwise.

```
void _nmod_poly_atan_series(mp_ptr g, mp_srcptr h, slong n,
    nmod_t mod)
```

Set $g = \operatorname{atan}(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed.

```
void nmod_poly_atan_series(nmod_poly_t g, const nmod_poly_t
    h, slong n)
```

Set $g = \operatorname{atan}(h) + O(x^n)$.

```
void _nmod_poly_atanh_series(mp_ptr g, mp_srcptr h, slong
    n, nmod_t mod)
```

Set $g = \operatorname{atanh}(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed.

```
void nmod_poly_atanh_series(nmod_poly_t g, const
   nmod_poly_t h, slong n)
```

Set $g = \operatorname{atanh}(h) + O(x^n)$.

```
void _nmod_poly_asin_series(mp_ptr g, mp_srcptr h, slong n,
   nmod_t mod)
```

Set $g = \operatorname{asin}(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed.

```
void nmod_poly_asin_series(nmod_poly_t g, const nmod_poly_t
   h, slong n)
```

Set $g = \operatorname{asin}(h) + O(x^n)$.

```
void _nmod_poly_asinh_series(mp_ptr g, mp_srcptr h, slong
   n, nmod_t mod)
```

Set $g = \operatorname{asinh}(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed.

```
void nmod_poly_asinh_series(nmod_poly_t g, const
   nmod_poly_t h, slong n)
```

Set $g = \operatorname{asinh}(h) + O(x^n)$.

```
void _nmod_poly_sin_series(mp_ptr g, mp_srcptr h, slong n,
   nmod_t mod)
```

Set $g = \sin(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed. The value is computed using the identity $\sin(x) = 2\tan(x/2))/(1 + \tan^2(x/2))$.

```
void nmod_poly_sin_series(nmod_poly_t g, const nmod_poly_t
   h, slong n)
```

Set $g = \sin(h) + O(x^n)$.

```
void _nmod_poly_cos_series(mp_ptr g, mp_srcptr h, slong n,
   nmod_t mod)
```

Set $g = \cos(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is allowed. The value is computed using the identity $\cos(x) = (1 - \tan^2(x/2))/(1 + \tan^2(x/2))$.

```
void nmod_poly_cos_series(nmod_poly_t g, const nmod_poly_t
   h, slong n)
```

Set $g = \cos(h) + O(x^n)$.

```
void _nmod_poly_tan_series(mp_ptr g, mp_srcptr h, slong n,
   nmod_t mod)
```

Set $g = \tan(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is not allowed. Uses Newton iteration to invert the atan function.

```
void nmod_poly_tan_series(nmod_poly_t g, const nmod_poly_t
   h, slong n)
```

Set $g = \tan(h) + O(x^n)$.

```
void _nmod_poly_sinh_series(mp_ptr g, mp_srcptr h, slong n,
    nmod_t mod)
```

Set $g = \sinh(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is not allowed. Uses the identity $\sinh(x) = (e^x - e^{-x})/2$.

```
void nmod_poly_sinh_series(nmod_poly_t g, const nmod_poly_t
    h, slong n)
```

Set $g = \sinh(h) + O(x^n)$.

```
void _nmod_poly_cosh_series(mp_ptr g, mp_srcptr h, slong n,
    nmod_t mod)
```

Set $g = \cos(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Aliasing of g and h is not allowed. Uses the identity $\cosh(x) = (e^x + e^{-x})/2$.

```
void nmod_poly_cosh_series(nmod_poly_t g, const nmod_poly_t
    h, slong n)
```

Set $g = \cosh(h) + O(x^n)$.

```
void _nmod_poly_tanh_series(mp_ptr g, mp_srcptr h, slong n,
    nmod_t mod)
```

Set $g = \tanh(h) + O(x^n)$. Assumes $n > 0$ and that h is zero-padded as necessary to length n. Uses the identity $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$.

```
void nmod_poly_tanh_series(nmod_poly_t g, const nmod_poly_t
    h, slong n)
```

Set $g = \tanh(h) + O(x^n)$.

30.32 Products

```
void _nmod_poly_product_roots_nmod_vec(mp_ptr poly,
    mp_srcptr xs, slong n, nmod_t mod)
```

Sets `(poly, n + 1)` to the monic polynomial which is the product of $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$, the roots x_i being given by `xs`.

Aliasing of the input and output is not allowed.

```
void nmod_poly_product_roots_nmod_vec(nmod_poly_t poly,
    mp_srcptr xs, slong n)
```

Sets `poly` to the monic polynomial which is the product of $(x-x_0)(x-x_1)\cdots(x-x_{n-1})$, the roots x_i being given by `xs`.

30.33 Subproduct trees

```
mp_ptr * _nmod_poly_tree_alloc(slong len)
```

Allocates space for a subproduct tree of the given length, having linear factors at the lowest level.

Entry i in the tree is a pointer to a single array of limbs, capable of storing $\lfloor n/2^i \rfloor$ subproducts of degree 2^i adjacently, plus a trailing entry if $n/2^i$ is not an integer.

For example, a tree of length 7 built from monic linear factors has the following structure, where spaces have been inserted for illustrative purposes:

```
X1 X1 X1 X1 X1 X1 X1
XX1   XX1   XX1   X1
XXXX1       XX1   X1
XXXXXXX1
```

```
void _nmod_poly_tree_free(mp_ptr * tree, slong len)
```

Free the allocated space for the subproduct.

```
void _nmod_poly_tree_build(mp_ptr * tree, mp_srcptr roots,
   slong len, nmod_t mod)
```

Builds a subproduct tree in the preallocated space from the `len` monic linear factors $(x - r_i)$. The top level product is not computed.

30.34 Inflation and deflation

```
void nmod_poly_inflate(nmod_poly_t result, const
   nmod_poly_t input, ulong inflation)
```

Sets `result` to the inflated polynomial $p(x^n)$ where p is given by `input` and n is given by `deflation`.

```
void nmod_poly_deflate(nmod_poly_t result, const
   nmod_poly_t input, ulong deflation)
```

Sets `result` to the deflated polynomial $p(x^{1/n})$ where p is given by `input` and n is given by `deflation`. Requires $n > 0$.

```
ulong nmod_poly_deflation(const nmod_poly_t input)
```

Returns the largest integer by which `input` can be deflated. As special cases, returns 0 if `input` is the zero polynomial and 1 of `input` is a constant polynomial.

§31. nmod_poly_factor

Factorisation of polynomials over $\mathbf{Z}/n\mathbf{Z}$ for word-sized moduli

The `nmod_poly_factor` module is included automatically with `nmod_poly.h`. One should not try to include `nmod_poly_factor.h` directly.

31.1 Factorisation

```
void nmod_poly_factor_init(nmod_poly_factor_t fac)
```

Initialises `fac` for use. An `nmod_poly_factor_t` represents a polynomial in factorised form as a product of polynomials with associated exponents.

```
void nmod_poly_factor_clear(nmod_poly_factor_t fac)
```

Frees all memory associated with `fac`.

```
void nmod_poly_factor_realloc(nmod_poly_factor_t fac, slong
    alloc)
```

Reallocates the factor structure to provide space for precisely `alloc` factors.

```
void nmod_poly_factor_fit_length(nmod_poly_factor_t fac,
    slong len)
```

Ensures that the factor structure has space for at least `len` factors. This functions takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

```
void nmod_poly_factor_set(nmod_poly_factor_t res, const
    nmod_poly_factor_t fac)
```

Sets `res` to the same factorisation as `fac`.

```
void nmod_poly_factor_print(const nmod_poly_factor_t fac)
```

Prints the entries of `fac` to standard output.

```
void nmod_poly_factor_insert(nmod_poly_factor_t fac, const
    nmod_poly_t poly, slong exp)
```

Inserts the factor `poly` with multiplicity `exp` into the factorisation `fac`.

If `fac` already contains `poly`, then `exp` simply gets added to the exponent of the existing entry.

```
void nmod_poly_factor_concat(nmod_poly_factor_t res, const
    nmod_poly_factor_t fac)
```

Concatenates two factorisations.

This is equivalent to calling `nmod_poly_factor_insert()` repeatedly with the individual factors of `fac`.

Does not support aliasing between `res` and `fac`.

```
void nmod_poly_factor_pow(nmod_poly_factor_t fac, slong exp)
```

Raises `fac` to the power `exp`.

```
ulong nmod_poly_remove(nmod_poly_t f, const nmod_poly_t p)
```

Removes the highest possible power of `p` from `f` and returns the exponent.

```
int nmod_poly_is_irreducible(const nmod_poly_t f)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0.

```
int nmod_poly_is_irreducible_ddf(const nmod_poly_t f)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

```
int nmod_poly_is_irreducible_rabin(const nmod_poly_t f)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses Rabin irreducibility test.

```
int _nmod_poly_is_squarefree(mp_srcptr f, slong len, nmod_t
    mod)
```

Returns 1 if `(f, len)` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

```
int nmod_poly_is_squarefree(const nmod_poly_t f)
```

Returns 1 if `f` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

```
void nmod_poly_factor_squarefree(nmod_poly_factor_t res,
    const nmod_poly_t f)
```

Sets `res` to a square-free factorization of `f`.

```
int nmod_poly_factor_equal_deg_prob(nmod_poly_t factor,
    flint_rand_t state, const nmod_poly_t pol, slong d)
```

Probabilistic equal degree factorisation of `pol` into irreducible factors of degree `d`. If it passes, a factor is placed in factor and 1 is returned, otherwise 0 is returned and the value of factor is undetermined.

Requires that `pol` be monic, non-constant and squarefree.

```
void nmod_poly_factor_equal_deg(nmod_poly_factor_t factors,
    const nmod_poly_t pol, slong d)
```

Assuming `pol` is a product of irreducible factors all of degree `d`, finds all those factors and places them in factors. Requires that `pol` be monic, non-constant and squarefree.

```
void nmod_poly_factor_distinct_deg(nmod_poly_factor_t res,
    const nmod_poly_t poly, slong * const *degs)
```

Factorises a monic non-constant squarefree polymnomial `poly` of degree n into factors $f[d]$ such that for $1 \leq d \leq n$ $f[d]$ is the product of the monic irreducible factors of `poly` of degree d. Factors $f[d]$ are stored in `res`, and the degree d of the irreducible factors is stored in `degs` in the same order as the factors.

Requires that `degs` has enough space for $(n/2) + 1 * sizeof(slong)$.

```
void nmod_poly_factor_cantor_zassenhaus(nmod_poly_factor_t
   res, const nmod_poly_t f)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the Cantor-Zassenhaus algorithm.

```
void nmod_poly_factor_berlekamp(nmod_poly_factor_t res,
   const nmod_poly_t f)
```

Factorises a non-constant, squarefree polynomial `f` into monic irreducible factors using the Berlekamp algorithm.

```
void nmod_poly_factor_kaltofen_shoup(nmod_poly_factor_t
   res, const nmod_poly_t poly)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a âĂIJbaby step/giant stepâĂİ strategy for the distinct-degree factorization step.

```
mp_limb_t
   nmod_poly_factor_with_berlekamp(nmod_poly_factor_t res,
   const nmod_poly_t f)
```

Factorises a general polynomial `f` into monic irreducible factors and returns the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Berlekamp on all the individual square-free factors.

```
mp_limb_t
   nmod_poly_factor_with_cantor_zassenhaus(nmod_poly_factor_t
   res, const nmod_poly_t f)
```

Factorises a general polynomial `f` into monic irreducible factors and returns the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

```
mp_limb_t
   nmod_poly_factor_with_kaltofen_shoup(nmod_poly_factor_t
   res, const nmod_poly_t f)
```

Factorises a general polynomial `f` into monic irreducible factors and returns the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

```
mp_limb_t nmod_poly_factor(nmod_poly_factor_t res, const
   nmod_poly_t f)
```

Factorises a general polynomial `f` into monic irreducible factors and returns the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs either Cantor-Zassenhaus or Berlekamp on all the individual square-free factors. Currently Cantor-Zassenhaus is used by default unless the modulus is 2, in which case Berlekamp is used.

§32. nmod_mat

Matrices over $\mathbf{Z}/n\mathbf{Z}$ for word-sized moduli

32.1 Introduction

An `nmod_mat_t` represents a matrix of integers modulo n, for any non-zero modulus n that fits in a single limb, up to $2^{32} - 1$ or $2^{64} - 1$.

The `nmod_mat_t` type is defined as an array of `nmod_mat_struct`'s of length one. This permits passing parameters of type `nmod_mat_t` by reference.

An `nmod_mat_t` internally consists of a single array of `mp_limb_t`'s, representing a dense matrix in row-major order. This array is only directly indexed during memory allocation and deallocation. A separate array holds pointers to the start of each row, and is used for all indexing. This allows the rows of a matrix to be permuted quickly by swapping pointers.

Matrices having zero rows or columns are allowed.

The shape of a matrix is fixed upon initialisation. The user is assumed to provide input and output variables whose dimensions are compatible with the given operation.

It is assumed that all matrices passed to a function have the same modulus. The modulus is assumed to be a prime number in functions that perform some kind of division, solving, or Gaussian elimination (including computation of rank and determinant), but can be composite in functions that only perform basic manipulation and ring operations (e.g. transpose and matrix multiplication).

The user can manipulate matrix entries directly, but must assume responsibility for normalising all values to the range $[0, n)$.

32.2 Memory management

```
void nmod_mat_init(nmod_mat_t mat, slong rows, slong cols,
    mp_limb_t n)
```

Initialises `mat` to a `rows`-by-`cols` matrix with coefficients modulo n, where n can be any nonzero integer that fits in a limb. All elements are set to zero.

```
void nmod_mat_init_set(nmod_mat_t mat, nmod_mat_t src)
```

Initialises `mat` and sets its dimensions, modulus and elements to those of `src`.

```
void nmod_mat_clear(nmod_mat_t mat)
```

Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an `nmod_mat_t` object.

```
void nmod_mat_set(nmod_mat_t mat, nmod_mat_t src)
```

Sets `mat` to a copy of `src`. It is assumed that `mat` and `src` have identical dimensions.

32.3 Basic properties and manipulation

```
MACRO nmod_mat_entry(nmod_mat_t mat, slong i, slong j)
```

Directly accesses the entry in `mat` in row i and column j, indexed from zero. No bounds checking is performed. This macro can be used both for reading and writing coefficients.

```
slong nmod_mat_nrows(nmod_mat_t mat)
```

Returns the number of rows in `mat`. This function is implemented as a macro.

```
slong nmod_mat_ncols(nmod_mat_t mat)
```

Returns the number of columns in `mat`. This function is implemented as a macro.

32.4 Printing

```
void nmod_mat_print_pretty(nmod_mat_t mat)
```

Pretty-prints `mat` to `stdout`. A header is printed followed by the rows enclosed in brackets. Each column is right-aligned to the width of the modulus written in decimal, and the columns are separated by spaces. For example:

```
<2 x 3 integer matrix mod 2903>
[   0    0 2607]
[ 622    0    0]
```

32.5 Random matrix generation

```
void nmod_mat_randtest(nmod_mat_t mat, flint_rand_t state)
```

Sets the elements to a random matrix with entries between 0 and $m-1$ inclusive, where m is the modulus of `mat`. A sparse matrix is generated with increased probability.

```
void nmod_mat_randfull(nmod_mat_t mat, flint_rand_t state)
```

Sets the element to random numbers likely to be close to the modulus of the matrix. This is used to test potential overflow-related bugs.

```
int nmod_mat_randpermdiag(nmod_mat_t mat, mp_limb_t * diag,
   slong n, flint_rand_t state)
```

Sets `mat` to a random permutation of the diagonal matrix with n leading entries given by the vector `diag`. It is assumed that the main diagonal of `mat` has room for at least n entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.

```
void nmod_mat_randrank(nmod_mat_t mat, slong rank,
   flint_rand_t state)
```

Sets `mat` to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random integers between 0 and $m - 1$ inclusive, where m is the modulus of `mat`.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling `nmod_mat_randops()`.

```
void nmod_mat_randops(nmod_mat_t mat, slong count,
    flint_rand_t state)
```

Randomises `mat` by performing elementary row or column operations. More precisely, at most `count` random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

```
void nmod_mat_randtril(nmod_mat_t mat, flint_rand_t state,
    int unit)
```

Sets `mat` to a random lower triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

```
void nmod_mat_randtriu(nmod_mat_t mat, flint_rand_t state,
    int unit)
```

Sets `mat` to a random upper triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

32.6 Comparison

```
int nmod_mat_equal(nmod_mat_t mat1, nmod_mat_t mat2)
```

Returns nonzero if mat1 and mat2 have the same dimensions and elements, and zero otherwise. The moduli are ignored.

32.7 Transpose

```
void nmod_mat_transpose(nmod_mat_t B, nmod_mat_t A)
```

Sets B to the transpose of A. Dimensions must be compatible. B and A may be the same object if and only if the matrix is square.

32.8 Addition and subtraction

```
void nmod_mat_add(nmod_mat_t C, nmod_mat_t A, nmod_mat_t B)
```

Computes $C = A + B$. Dimensions must be identical.

```
void nmod_mat_sub(nmod_mat_t C, nmod_mat_t A, nmod_mat_t B)
```

Computes $C = A - B$. Dimensions must be identical.

```
void nmod_mat_neg(nmod_mat_t A, nmod_mat_t B)
```

Sets $B = -A$. Dimensions must be identical.

32.9 Matrix-scalar arithmetic

```
void nmod_mat_scalar_mul(nmod_mat_t B, const nmod_mat_t A,
    mp_limb_t c)
```

Sets $B = cA$, where the scalar c is assumed to be reduced modulo the modulus. Dimensions of A and B must be identical.

32.10 Matrix multiplication

```
void nmod_mat_mul(nmod_mat_t C, nmod_mat_t A, nmod_mat_t B)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. This function automatically chooses between classical and Strassen multiplication.

```
void nmod_mat_mul_classical(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Uses classical matrix multiplication, creating a temporary transposed copy of B to improve memory locality if the matrices are large enough, and packing several entries of B into each word if the modulus is very small.

```
void nmod_mat_mul_strassen(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Uses Strassen multiplication (the Strassen-Winograd variant).

```
void nmod_mat_addmul(nmod_mat_t D, const nmod_mat_t C,
    const nmod_mat_t A, const nmod_mat_t B)
```

Sets $D = C + AB$. C and D may be aliased with each other but not with A or B. Automatically selects between classical and Strassen multiplication.

```
void nmod_mat_submul(nmod_mat_t D, const nmod_mat_t C,
    const nmod_mat_t A, const nmod_mat_t B)
```

Sets $D = C + AB$. C and D may be aliased with each other but not with A or B.

32.11 Trace

```
mp_limb_t nmod_mat_trace(const nmod_mat_t mat)
```

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

32.12 Determinant and rank

```
mp_limb_t nmod_mat_det(nmod_mat_t A)
```

Returns the determinant of A. The modulus of A must be a prime number.

```
slong nmod_mat_rank(nmod_mat_t A)
```

Returns the rank of A. The modulus of A must be a prime number.

32.13 Inverse

```
int nmod_mat_inv(nmod_mat_t B, nmod_mat_t A)
```

Sets $B = A^{-1}$ and returns 1 if A is invertible. If A is singular, returns 0 and sets the elements of B to undefined values.

A and B must be square matrices with the same dimensions and modulus. The modulus must be prime.

32.14 Triangular solving

```
void nmod_mat_solve_tril(nmod_mat_t X, const nmod_mat_t L,
    const nmod_mat_t B, int unit)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

```
void nmod_mat_solve_tril_classical(nmod_mat_t X, const
    nmod_mat_t L, const nmod_mat_t B, int unit)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

```
void nmod_mat_solve_tril_recursive(nmod_mat_t X, const
    nmod_mat_t L, const nmod_mat_t B, int unit)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$\begin{pmatrix} A & 0 \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}X \\ D^{-1}(Y - CA^{-1}X) \end{pmatrix}$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

```
void nmod_mat_solve_triu(nmod_mat_t X, const nmod_mat_t U,
    const nmod_mat_t B, int unit)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

```
void nmod_mat_solve_triu_classical(nmod_mat_t X, const
    nmod_mat_t U, const nmod_mat_t B, int unit)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

```
void nmod_mat_solve_triu_recursive(nmod_mat_t X, const
    nmod_mat_t U, const nmod_mat_t B, int unit)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$\begin{pmatrix} A & B \\ 0 & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}(X - BD^{-1}Y) \\ D^{-1}Y \end{pmatrix}$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

32.15 Nonsingular square solving

```
int nmod_mat_solve(nmod_mat_t X, nmod_mat_t A, nmod_mat_t B)
```

Solves the matrix-matrix equation $AX = B$ over $\mathbf{Z}/p\mathbf{Z}$ where p is the modulus of X which must be a prime number. X, A, and B should have the same moduli.

Returns 1 if A has full rank; otherwise returns 0 and sets the elements of X to undefined values.

```
int nmod_mat_solve_vec(mp_limb_t * x, nmod_mat_t A,
    mp_limb_t * b)
```

Solves the matrix-vector equation $Ax = b$ over $\mathbf{Z}/p\mathbf{Z}$ where p is the modulus of A which must be a prime number.

Returns 1 if A has full rank; otherwise returns 0 and sets the elements of x to undefined values.

32.16 LU decomposition

```
slong nmod_mat_lu(slong * P, nmod_mat_t A, int rank_check)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A.

If A is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r, U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and return 0 if A is detected to be rank-deficient.

This function calls `nmod_mat_lu_recursive`.

```
slong nmod_mat_lu_classical(slong * P, nmod_mat_t A, int
    rank_check)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A. The behavior of this function is identical to that of `nmod_mat_lu`. Uses Gaussian elimination.

```
slong nmod_mat_lu_recursive(slong * P, nmod_mat_t A, int
    rank_check)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A. The behavior of this function is identical to that of `nmod_mat_lu`. Uses recursive block decomposition, switching to classical Gaussian elimination for sufficiently small blocks.

32.17 Reduced row echelon form

`slong nmod_mat_rref(nmod_mat_t A)`

Puts A in reduced row echelon form and returns the rank of A.

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

32.18 Nullspace

`slong nmod_mat_nullspace(nmod_mat_t X, const nmod_mat_t A)`

Computes the nullspace of A and returns the nullity.

More precisely, this function sets X to a maximum rank matrix such that $AX = 0$ and returns the rank of X. The columns of X will form a basis for the nullspace of A.

X must have sufficient space to store all basis vectors in the nullspace.

This function computes the reduced row echelon form and then reads off the basis vectors.

§33. nmod_poly_mat

Matrices over $\mathbf{Z}/n\mathbf{Z}[x]$ for word-sized moduli

---

The `nmod_poly_mat_t` data type represents matrices whose entries are polynomials having coefficients in $\mathbf{Z}/n\mathbf{Z}$. We generally assume that n is a prime number.

The `nmod_poly_mat_t` type is defined as an array of `nmod_poly_mat_struct`'s of length one. This permits passing parameters of type `nmod_poly_mat_t` by reference.

A matrix internally consists of a single array of `nmod_poly_struct`'s, representing a dense matrix in row-major order. This array is only directly indexed during memory allocation and deallocation. A separate array holds pointers to the start of each row, and is used for all indexing. This allows the rows of a matrix to be permuted quickly by swapping pointers.

Matrices having zero rows or columns are allowed.

The shape of a matrix is fixed upon initialisation. The user is assumed to provide input and output variables whose dimensions are compatible with the given operation.

33.1 Memory management

```
void nmod_poly_mat_init(nmod_poly_mat_t mat, slong rows,
    slong cols, mp_limb_t n)
```

Initialises a matrix with the given number of rows and columns for use. The modulus is set to n.

```
void nmod_poly_mat_init_set(nmod_poly_mat_t mat, const
    nmod_poly_mat_t src)
```

Initialises a matrix `mat` of the same dimensions and modulus as `src`, and sets it to a copy of `src`.

```
void nmod_poly_mat_clear(nmod_poly_mat_t mat)
```

Frees all memory associated with the matrix. The matrix must be reinitialised if it is to be used again.

33.2 Basic properties

```
slong nmod_poly_mat_nrows(const nmod_poly_mat_t mat)
```

Returns the number of rows in mat.

```
slong nmod_poly_mat_ncols(const nmod_poly_mat_t mat)
```

Returns the number of columns in mat.

```
mp_limb_t nmod_poly_mat_modulus(const nmod_poly_mat_t mat)
```

Returns the modulus of mat.

33.3 Basic assignment and manipulation

```
MACRO nmod_poly_mat_entry(mat,i,j)
```

Gives a reference to the entry at row i and column j. The reference can be passed as an input or output variable to any nmod_poly function for direct manipulation of the matrix element. No bounds checking is performed.

```
void nmod_poly_mat_set(nmod_poly_mat_t mat1, const
   nmod_poly_mat_t mat2)
```

Sets mat1 to a copy of mat2.

```
void nmod_poly_mat_swap(nmod_poly_mat_t mat1,
   nmod_poly_mat_t mat2)
```

Swaps mat1 and mat2 efficiently.

33.4 Input and output

```
void nmod_poly_mat_print(const nmod_poly_mat_t mat, const
   char * x)
```

Prints the matrix mat to standard output, using the variable x.

33.5 Random matrix generation

```
void nmod_poly_mat_randtest(nmod_poly_mat_t mat,
   flint_rand_t state, slong len)
```

This is equivalent to applying nmod_poly_randtest to all entries in the matrix.

```
void nmod_poly_mat_randtest_sparse(nmod_poly_mat_t A,
   flint_rand_t state, slong len, float density)
```

Creates a random matrix with the amount of nonzero entries given approximately by the density variable, which should be a fraction between 0 (most sparse) and 1 (most dense).

The nonzero entries will have random lengths between 1 and len.

33.6 Special matrices

```
void nmod_poly_mat_zero(nmod_poly_mat_t mat)
```

Sets mat to the zero matrix.

```
void nmod_poly_mat_one(nmod_poly_mat_t mat)
```

Sets `mat` to the unit or identity matrix of given shape, having the element 1 on the main diagonal and zeros elsewhere. If `mat` is nonsquare, it is set to the truncation of a unit matrix.

33.7 Basic comparison and properties

```
int nmod_poly_mat_equal(const nmod_poly_mat_t mat1, const
    nmod_poly_mat_t mat2)
```

Returns nonzero if `mat1` and `mat2` have the same shape and all their entries agree, and returns zero otherwise.

```
int nmod_poly_mat_is_zero(const nmod_poly_mat_t mat)
```

Returns nonzero if all entries in `mat` are zero, and returns zero otherwise.

```
int nmod_poly_mat_is_one(const nmod_poly_mat_t mat)
```

Returns nonzero if all entry of `mat` on the main diagonal are the constant polynomial 1 and all remaining entries are zero, and returns zero otherwise. The matrix need not be square.

```
int nmod_poly_mat_is_empty(const nmod_poly_mat_t mat)
```

Returns a non-zero value if the number of rows or the number of columns in `mat` is zero, and otherwise returns zero.

```
int nmod_poly_mat_is_square(const nmod_poly_mat_t mat)
```

Returns a non-zero value if the number of rows is equal to the number of columns in `mat`, and otherwise returns zero.

33.8 Norms

```
slong nmod_poly_mat_max_length(const nmod_poly_mat_t A)
```

Returns the maximum polynomial length among all the entries in `A`.

33.9 Evaluation

```
void nmod_poly_mat_evaluate_nmod(nmod_mat_t B, const
    nmod_poly_mat_t A, mp_limb_t x)
```

Sets the `nmod_mat_t` `B` to `A` evaluated entrywise at the point `x`.

33.10 Arithmetic

```
void nmod_poly_mat_scalar_mul_nmod_poly(nmod_poly_mat_t B,
    const nmod_poly_mat_t A, const nmod_poly_t c)
```

Sets `B` to `A` multiplied entrywise by the polynomial `c`.

```
void nmod_poly_mat_scalar_mul_nmod(nmod_poly_mat_t B, const
    nmod_poly_mat_t A, mp_limb_t c)
```

Sets `B` to `A` multiplied entrywise by the coefficient `c`, which is assumed to be reduced modulo the modulus.

```
void nmod_poly_mat_add(nmod_poly_mat_t C, const
   nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the sum of `A` and `B`. All matrices must have the same shape. Aliasing is allowed.

```
void nmod_poly_mat_sub(nmod_poly_mat_t C, const
   nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the sum of `A` and `B`. All matrices must have the same shape. Aliasing is allowed.

```
void nmod_poly_mat_neg(nmod_poly_mat_t B, const
   nmod_poly_mat_t A)
```

Sets `B` to the negation of `A`. The matrices must have the same shape. Aliasing is allowed.

```
void nmod_poly_mat_mul(nmod_poly_mat_t C, const
   nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the matrix product of `A` and `B`. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical, KS and evaluation-interpolation multiplication.

```
void nmod_poly_mat_mul_classical(nmod_poly_mat_t C, const
   nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the matrix product of `A` and `B`, computed using the classical algorithm. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

```
void nmod_poly_mat_mul_KS(nmod_poly_mat_t C, const
   nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the matrix product of `A` and `B`, computed using Kronecker segmentation. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

```
void nmod_poly_mat_mul_interpolate(nmod_poly_mat_t C, const
   nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Sets `C` to the matrix product of `A` and `B`, computed through evaluation and interpolation. The matrices must have compatible dimensions for matrix multiplication. For interpolation to be well-defined, we require that the modulus is a prime at least as large as $m + n - 1$ where m and n are the maximum lengths of polynomials in the input matrices. Aliasing is allowed.

```
void nmod_poly_mat_sqr(nmod_poly_mat_t B, const
   nmod_poly_mat_t A)
```

Sets `B` to the square of `A`, which must be a square matrix. Aliasing is allowed. This function automatically chooses between classical and KS squaring.

```
void nmod_poly_mat_sqr_classical(nmod_poly_mat_t B, const
   nmod_poly_mat_t A)
```

Sets `B` to the square of `A`, which must be a square matrix. Aliasing is allowed. This function uses direct formulas for very small matrices, and otherwise classical matrix multiplication.

```
void nmod_poly_mat_sqr_KS(nmod_poly_mat_t B, const
   nmod_poly_mat_t A)
```

Sets `B` to the square of `A`, which must be a square matrix. Aliasing is allowed. This function uses Kronecker segmentation.

```
void nmod_poly_mat_sqr_interpolate(nmod_poly_mat_t B, const
    nmod_poly_mat_t A)
```

Sets `B` to the square of `A`, which must be a square matrix, computed through evaluation and interpolation. For interpolation to be well-defined, we require that the modulus is a prime at least as large as $2n - 1$ where n is the maximum length of polynomials in the input matrix. Aliasing is allowed.

```
void nmod_poly_mat_pow(nmod_poly_mat_t B, const
    nmod_poly_mat_t A, ulong exp)
```

Sets `B` to `A` raised to the power `exp`, where `A` is a square matrix. Uses exponentiation by squaring. Aliasing is allowed.

33.11 Row reduction

```
slong nmod_poly_mat_find_pivot_any(const nmod_poly_mat_t
    mat, slong start_row, slong end_row, slong c)
```

Attempts to find a pivot entry for row reduction. Returns a row index r between `start_row` (inclusive) and `stop_row` (exclusive) such that column c in `mat` has a nonzero entry on row r, or returns -1 if no such entry exists.

This implementation simply chooses the first nonzero entry from it encounters. This is likely to be a nearly optimal choice if all entries in the matrix have roughly the same size, but can lead to unnecessary coefficient growth if the entries vary in size.

```
slong nmod_poly_mat_find_pivot_partial(const
    nmod_poly_mat_t mat, slong start_row, slong end_row,
    slong c)
```

Attempts to find a pivot entry for row reduction. Returns a row index r between `start_row` (inclusive) and `stop_row` (exclusive) such that column c in `mat` has a nonzero entry on row r, or returns -1 if no such entry exists.

This implementation searches all the rows in the column and chooses the nonzero entry of smallest degree. This heuristic typically reduces coefficient growth when the matrix entries vary in size.

```
slong nmod_poly_mat_fflu(nmod_poly_mat_t B, nmod_poly_t
    den, slong * perm, const nmod_poly_mat_t A, int
    rank_check)
```

Uses fraction-free Gaussian elimination to set (`B`, `den`) to a fraction-free LU decomposition of `A` and returns the rank of `A`. Aliasing of `A` and `B` is allowed.

Pivot elements are chosen with `nmod_poly_mat_find_pivot_partial`. If `perm` is non-`NULL`, the permutation of rows in the matrix will also be applied to `perm`.

If `rank_check` is set, the function aborts and returns 0 if the matrix is detected not to have full rank without completing the elimination.

The denominator `den` is set to $\pm \det(A)$, where the sign is decided by the parity of the permutation. Note that the determinant is not generally the minimal denominator.

```
slong nmod_poly_mat_rref(nmod_poly_mat_t B, nmod_poly_t
    den, const nmod_poly_mat_t A)
```

Sets (`B`, `den`) to the reduced row echelon form of `A` and returns the rank of `A`. Aliasing of `A` and `B` is allowed.

The denominator `den` is set to $\pm \det(A)$. Note that the determinant is not generally the minimal denominator.

33.12 Trace

```
void nmod_poly_mat_trace(nmod_poly_t trace, const
    nmod_poly_mat_t mat)
```

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

33.13 Determinant and rank

```
void nmod_poly_mat_det(nmod_poly_t det, const
    nmod_poly_mat_t A)
```

Sets `det` to the determinant of the square matrix `A`. Uses a direct formula, fraction-free LU decomposition, or interpolation, depending on the size of the matrix.

```
void nmod_poly_mat_det_fflu(nmod_poly_t det, const
    nmod_poly_mat_t A)
```

Sets `det` to the determinant of the square matrix `A`. The determinant is computed by performing a fraction-free LU decomposition on a copy of `A`.

```
void nmod_poly_mat_det_interpolate(nmod_poly_t det, const
    nmod_poly_mat_t A)
```

Sets `det` to the determinant of the square matrix `A`. The determinant is computed by determing a bound n for its length, evaluating the matrix at n distinct points, computing the determinant of each coefficient matrix, and forming the interpolating polynomial.

If the coefficient ring does not contain n distinct points (that is, if working over $\mathbb{Z}/p\mathbb{Z}$ where $p < n$), this function automatically falls back to `nmod_poly_mat_det_fflu`.

```
slong nmod_poly_mat_rank(const nmod_poly_mat_t A)
```

Returns the rank of `A`. Performs fraction-free LU decomposition on a copy of `A`.

33.14 Inverse

```
int nmod_poly_mat_inv(nmod_poly_mat_t Ainv, nmod_poly_t
    den, const nmod_poly_mat_t A)
```

Sets (`Ainv`, `den`) to the inverse matrix of `A`. Returns 1 if `A` is nonsingular and 0 if `A` is singular. Aliasing of `Ainv` and `A` is allowed.

More precisely, `det` will be set to the determinant of `A` and `Ainv` will be set to the adjugate matrix of `A`. Note that the determinant is not necessarily the minimal denominator.

Uses fraction-free LU decomposition, followed by solving for the identity matrix.

33.15 Nullspace

```
slong nmod_poly_mat_nullspace(nmod_poly_mat_t res, const
    nmod_poly_mat_t mat)
```

Computes the right rational nullspace of the matrix `mat` and returns the nullity.

More precisely, assume that `mat` has rank r and nullity n. Then this function sets the first n columns of `res` to linearly independent vectors spanning the nullspace of `mat`. As a result, we always have $\operatorname{rank}(\texttt{res}) = n$, and `mat` $\times$ `res` is the zero matrix.

The computed basis vectors will not generally be in a reduced form. In general, the polynomials in each column vector in the result will have a nontrivial common GCD.

33.16 Solving

```
int nmod_poly_mat_solve(nmod_poly_mat_t X, nmod_poly_t den,
    const nmod_poly_mat_t A, const nmod_poly_mat_t B)
```

Solves the equation $AX = B$ for nonsingular A. More precisely, computes (`X`, `den`) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

```
int nmod_poly_mat_solve_fflu(nmod_poly_mat_t X, nmod_poly_t
    den, const nmod_poly_mat_t A, const nmod_poly_mat_t B);
```

Solves the equation $AX = B$ for nonsingular A. More precisely, computes (`X`, `den`) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

```
void nmod_poly_mat_solve_fflu_precomp(nmod_poly_mat_t X,
    const slong * perm, const nmod_poly_mat_t FFLU, const
    nmod_poly_mat_t B);
```

Performs fraction-free forward and back substitution given a precomputed fraction-free LU decomposition and corresponding permutation.

§34. fmpz_mod_poly

Polynomials over $\mathbf{Z}/n\mathbf{Z}$ for general moduli

34.1 Introduction

The `fmpz_mod_poly_t` data type represents elements of $\mathbf{Z}/n\mathbf{Z}[x]$ for a fixed modulus n. The `fmpz_mod_poly` module provides routines for memory management, basic arithmetic and some higher level functions such as GCD, etc.

Each coefficient of an `fmpz_mod_poly_t` is of type `fmpz` and represents an integer reduced modulo the fixed modulus n in the range $[0, n)$.

Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

34.2 Simple example

The following example computes the square of the polynomial $5x^3 + 6$ in $\mathbf{Z}/7\mathbf{Z}[x]$.

```
#include "fmpz_mod_poly.h"
...
fmpz_t n;
fmpz_mod_poly_t x, y;

fmpz_init_set_ui(n, 7);
fmpz_mod_poly_init(x, n);
fmpz_mod_poly_init(y, n);
fmpz_mod_poly_set_coeff_ui(x, 3, 5);
fmpz_mod_poly_set_coeff_ui(x, 0, 6);
fmpz_mod_poly_sqr(y, x);
fmpz_mod_poly_print(x); flint_printf("\n");
fmpz_mod_poly_print(y); flint_printf("\n");
fmpz_mod_poly_clear(x);
fmpz_mod_poly_clear(y);
fmpz_clear(n);
```

The output is:

```
4 7  6 0 0 5
7 7  1 0 0 4 0 0 4
```

34.3 Definition of the fmpz_mod_poly_t type

The `fmpz_mod_poly_t` type is a typedef for an array of length 1 of `fmpz_mod_poly_struct`'s. This permits passing parameters of type `fmpz_mod_poly_t` by reference.

In reality one never deals directly with the `struct` and simply deals with objects of type `fmpz_mod_poly_t`. For simplicity we will think of an `fmpz_mod_poly_t` as a `struct`, though in practice to access fields of this `struct`, one needs to dereference first, e.g. to access the `length` field of an `fmpz_mod_poly_t` called `poly1` one writes `poly1->length`.

An `fmpz_mod_poly_t` is said to be *normalised* if either `length` is zero, or if the leading coefficient of the polynomial is non-zero. All `fmpz_mod_poly` functions expect their inputs to be normalised and all coefficients to be reduced modulo n, and unless otherwise specified they produce output that is normalised with coefficients reduced modulo n.

It is recommended that users do not access the fields of an `fmpz_mod_poly_t` or its coefficient data directly, but make use of the functions designed for this purpose, detailed below.

Functions in `fmpz_mod_poly` do all the memory management for the user. One does not need to specify the maximum length in advance before using a polynomial object. FLINT reallocates space automatically as the computation proceeds, if more space is required.

We now describe the functions available in `fmpz_mod_poly`.

34.4 Memory management

```
void fmpz_mod_poly_init(fmpz_mod_poly_t poly, const fmpz_t
   p)
```

Initialises `poly` for use over $\mathbf{Z}/p\mathbf{Z}$, setting its length to zero.

A corresponding call to `fmpz_mod_poly_clear()` must be made after finishing with the `fmpz_mod_poly_t` to free the memory used by the polynomial. The user is also responsible to clearing the integer p.

```
void fmpz_mod_poly_init2(fmpz_mod_poly_t poly, const fmpz_t
   p, slong alloc)
```

Initialises `poly` with space for at least `alloc` coefficients and sets the length to zero. The allocated coefficients are all set to zero.

```
void fmpz_mod_poly_clear(fmpz_mod_poly_t poly)
```

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

```
void fmpz_mod_poly_realloc(fmpz_mod_poly_t poly, slong
   alloc)
```

Reallocates the given polynomial to have space for `alloc` coefficients. If `alloc` is zero the polynomial is cleared and then reinitialised. If the current length is greater than `alloc` the polynomial is first truncated to length `alloc`.

```
void fmpz_mod_poly_fit_length(fmpz_mod_poly_t poly, slong
   len)
```

If `len` is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least `len` coefficients. No data is lost when calling this function.

The function efficiently deals with the case where it is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

```
void _fmpz_mod_poly_normalise(fmpz_mod_poly_t poly)
```

Sets the length of `poly` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

```
void _fmpz_mod_poly_set_length(fmpz_mod_poly_t poly, slong
    len)
```

Demotes the coefficients of `poly` beyond `len` and sets the length of `poly` to `len`.

```
void fmpz_mod_poly_truncate(fmpz_mod_poly_t poly, slong len)
```

If the current length of `poly` is greater than `len`, it is truncated to have the given length. Discarded coefficients are not necessarily set to zero.

34.5 Randomisation

```
void fmpz_mod_poly_randtest(fmpz_mod_poly_t f, flint_rand_t
    state, slong len)
```

Sets the polynomial f to a random polynomial of length up `len`.

```
void fmpz_mod_poly_randtest_irreducible(fmpz_mod_poly_t f,
    flint_rand_t state, slong len)
```

Sets the polynomial f to a random irreducible polynomial of length up `len`, assuming `len` is positive.

```
void fmpz_mod_poly_randtest_not_zero(fmpz_mod_poly_t f,
    flint_rand_t state, slong len)
```

Sets the polynomial f to a random polynomial of length up `len`, assuming `len` is positive.

```
void fmpz_mod_poly_randtest_monic(fmpz_mod_poly_t poly,
    flint_rand_t state, slong len)
```

Generates a random monic polynomial with length `len`.

```
void
    fmpz_mod_poly_randtest_monic_irreducible(fmpz_mod_poly_t
    poly, flint_rand_t state, slong len)
```

Generates a random monic irreducible polynomial with length `len`.

```
void fmpz_mod_poly_randtest_trinomial(fmpz_mod_poly_t poly,
    flint_rand_t state, slong len)
```

Generates a random monic trinomial of length `len`.

```
int
    fmpz_mod_poly_randtest_trinomial_irreducible(fmpz_mod_poly_t
    poly, flint_rand_t state, slong len, slong max_attempts)
```

Attempts to set `poly` to a monic irreducible trinomial of length `len`. It will generate up to `max_attempts` trinomials in attempt to find an irreducible one. If `max_attempts` is `0`, then it will keep generating trinomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.

```
void fmpz_mod_poly_randtest_pentomial(fmpz_mod_poly_t poly,
   flint_rand_t state, slong len)
```

Generates a random monic pentomial of length `len`.

```
int
   fmpz_mod_poly_randtest_pentomial_irreducible(fmpz_mod_poly_t
   poly, flint_rand_t state, slong len, slong max_attempts)
```

Attempts to set `poly` to a monic irreducible pentomial of length `len`. It will generate up to `max_attempts` pentomials in attempt to find an irreducible one. If `max_attempts` is `0`, then it will keep generating pentomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.

```
void
   fmpz_mod_poly_randtest_sparse_irreducible(fmpz_mod_poly_t
   poly, flint_rand_t state, slong len)
```

Attempts to set `poly` to a sparse, monic irreducible polynomial with length `len`. It attempts to find an irreducible trinomial. If that does not succeed, it attempts to find a irreducible pentomial. If that fails, then `poly` is just set to a random monic irreducible polynomial.

34.6 Attributes

```
fmpz * fmpz_mod_poly_modulus(const fmpz_mod_poly_t poly)
```

Returns the modulus of this polynomial. This function is implemented as a macro.

```
slong fmpz_mod_poly_degree(const fmpz_mod_poly_t poly)
```

Returns the degree of the polynomial. The degree of the zero polynomial is defined to be -1.

```
slong fmpz_mod_poly_length(const fmpz_mod_poly_t poly)
```

Returns the length of the polynomial, which is one more than its degree.

```
fmpz * fmpz_mod_poly_lead(const fmpz_mod_poly_t poly)
```

Returns a pointer to the first leading coefficient of `poly` if this is non-zero, otherwise returns `NULL`.

34.7 Assignment and basic manipulation

```
void fmpz_mod_poly_set(fmpz_mod_poly_t poly1, const
   fmpz_mod_poly_t poly2)
```

Sets the polynomial `poly1` to the value of `poly2`.

```
void fmpz_mod_poly_swap(fmpz_mod_poly_t poly1,
   fmpz_mod_poly_t poly2)
```

Swaps the two polynomials. This is done efficiently by swapping pointers rather than individual coefficients.

```
void fmpz_mod_poly_zero(fmpz_mod_poly_t poly)
```

Sets `poly` to the zero polynomial.

```
void fmpz_mod_poly_zero_coeffs(fmpz_mod_poly_t poly, slong
   i, slong j)
```

Sets the coefficients of X^k for $k \in [i, j)$ in the polynomial to zero.

```
void fmpz_mod_poly_reverse(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly, slong n)
```

This function considers the polynomial `poly` to be of length n, notionally truncating and zero padding if required, and reverses the result. Since the function normalises its result `res` may be of length less than n.

34.8 Conversion

```
void fmpz_mod_poly_set_ui(fmpz_mod_poly_t f, ulong c)
```

Sets the polynomial f to the constant c reduced modulo p.

```
void fmpz_mod_poly_set_fmpz(fmpz_mod_poly_t f, const fmpz_t
   c)
```

Sets the polynomial f to the constant c reduced modulo p.

```
void fmpz_mod_poly_set_fmpz_poly(fmpz_mod_poly_t f, const
   fmpz_poly_t g)
```

Sets f to g reduced modulo p, where p is the modulus that is part of the data structure of f.

```
void fmpz_mod_poly_get_fmpz_poly(fmpz_poly_t f, const
   fmpz_mod_poly_t g)
```

Sets f to g. This is done simply by lifting the coefficients of g taking representatives $[0, p) \subset \mathbf{Z}$.

34.9 Comparison

```
int fmpz_mod_poly_equal(const fmpz_mod_poly_t poly1, const
   fmpz_mod_poly_t poly2)
```

Returns non-zero if the two polynomials are equal.

```
int fmpz_mod_poly_is_zero(const fmpz_mod_poly_t poly)
```

Returns non-zero if the polynomial is zero.

34.10 Getting and setting coefficients

```
void fmpz_mod_poly_set_coeff_fmpz(fmpz_mod_poly_t poly,
   slong n, const fmpz_t x)
```

Sets the coefficient of X^n in the polynomial to x, assuming $n \geq 0$.

```
void fmpz_mod_poly_set_coeff_ui(fmpz_mod_poly_t poly, slong
   n, ulong x)
```

Sets the coefficient of X^n in the polynomial to x, assuming $n \geq 0$.

```
void fmpz_mod_poly_get_coeff_fmpz(fmpz_t x, const
   fmpz_mod_poly_t poly, slong n)
```

Sets x to the coefficient of X^n in the polynomial, assumng $n \geq 0$.

```
void fmpz_mod_poly_set_coeff_mpz(fmpz_mod_poly_t poly,
   slong n, const mpz_t x)
```

Sets the coefficient of X^n in the polynomial to x, assuming $n \geq 0$.

```
void fmpz_mod_poly_get_coeff_mpz(mpz_t x, const
   fmpz_mod_poly_t poly, slong n)
```

Sets x to the coefficient of X^n in the polynomial, assumng $n \geq 0$.

34.11 Shifting

```
void _fmpz_mod_poly_shift_left(fmpz * res, const fmpz *
   poly, slong len, slong n)
```

Sets `(res, len + n)` to `(poly, len)` shifted left by n coefficients.

Inserts zero coefficients at the lower end. Assumes that `len` and n are positive, and that `res` fits `len + n` elements. Supports aliasing between `res` and `poly`.

```
void fmpz_mod_poly_shift_left(fmpz_mod_poly_t f, const
   fmpz_mod_poly_t g, slong n)
```

Sets `res` to `poly` shifted left by n coeffs. Zero coefficients are inserted.

```
void _fmpz_mod_poly_shift_right(fmpz * res, const fmpz *
   poly, slong len, slong n)
```

Sets `(res, len - n)` to `(poly, len)` shifted right by n coefficients.

Assumes that `len` and n are positive, that `len > n`, and that `res` fits `len - n` elements. Supports aliasing between `res` and `poly`, although in this case the top coefficients of `poly` are not set to zero.

```
void fmpz_mod_poly_shift_right(fmpz_mod_poly_t f, const
   fmpz_mod_poly_t g, slong n)
```

Sets `res` to `poly` shifted right by n coefficients. If n is equal to or greater than the current length of `poly`, `res` is set to the zero polynomial.

34.12 Addition and subtraction

```
void _fmpz_mod_poly_add(fmpz *res, const fmpz *poly1, slong
   len1, const fmpz *poly2, slong len2, const fmpz_t p)
```

Sets `res` to the sum of `(poly1, len1)` and `(poly2, len2)`. It is assumed that `res` has sufficient space for the longer of the two polynomials.

```
void fmpz_mod_poly_add(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2)
```

Sets `res` to the sum of `poly1` and `poly2`.

```
void _fmpz_mod_poly_sub(fmpz *res, const fmpz *poly1, slong
   len1, const fmpz *poly2, slong len2, const fmpz_t p)
```

Sets `res` to `(poly1, len1)` minus `(poly2, len2)`. It is assumed that `res` has sufficient space for the longer of the two polynomials.

```
void fmpz_mod_poly_sub(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2)
```

Sets `res` to `poly1` minus `poly2`.

```
void _fmpz_mod_poly_neg(fmpz *res, const fmpz *poly, slong
   len, const fmpz_t p)
```

Sets `(res, len)` to the negative of `(poly, len)` modulo p.

```
void fmpz_mod_poly_neg(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly)
```

Sets `res` to the negative of `poly` modulo p.

34.13 Scalar multiplication

```
void _fmpz_mod_poly_scalar_mul_fmpz(fmpz *res, const fmpz
   *poly, slong len, const fmpz_t x, const fmpz_t p)
```

Sets `(res, len)` to `(poly, len)` multiplied by x, reduced modulo p.

```
void fmpz_mod_poly_scalar_mul_fmpz(fmpz_mod_poly_t res,
   const fmpz_mod_poly_t poly, const fmpz_t x)
```

Sets `res` to `poly` multiplied by x.

34.14 Multiplication

```
void _fmpz_mod_poly_mul(fmpz *res, const fmpz *poly1, slong
   len1, const fmpz *poly2, slong len2, const fmpz_t p)
```

Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2)`. Assumes `len1 >= len2 > 0`. Allows zero-padding of the two input polynomials.

```
void fmpz_mod_poly_mul(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _fmpz_mod_poly_mullow(fmpz *res, const fmpz *poly1,
   slong len1, const fmpz *poly2, slong len2, const fmpz_t
   p, slong n)
```

Sets `(res, n)` to the lowest n coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes `len1 >= len2 > 0` and `0 < n <= len1 + len2 - 1`. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

```
void fmpz_mod_poly_mullow(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2,
   slong n)
```

Sets `res` to the lowest n coefficients of the product of `poly1` and `poly2`.

```
void _fmpz_mod_poly_sqr(fmpz *res, const fmpz *poly, slong
   len, const fmpz_t p)
```

Sets `res` to the square of `poly`.

```
void fmpz_mod_poly_sqr(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly)
```

Computes `res` as the square of `poly`.

```
void _fmpz_mod_poly_mulmod(fmpz * res, const fmpz * poly1,
   slong len1, const fmpz * poly2, slong len2, const fmpz *
   f, slong lenf, const fmpz_t p)
```

Sets `res, len1 + len2 - 1` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_fmpz_mod_poly_mul` instead.

Aliasing of `f` and `res` is not permitted.

```
void fmpz_mod_poly_mulmod(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2,
   const fmpz_mod_poly_t f)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

```
void _fmpz_mod_poly_mulmod_preinv(fmpz * res, const fmpz *
   poly1, slong len1, const fmpz * poly2, slong len2, const
   fmpz * f, slong lenf, const fmpz* finv, slong lenfinv,
   const fmpz_t p)
```

Sets `res, len1 + len2 - 1` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `finv` is the inverse of the reverse of `f` mod `x^lenf`. It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. It is required that `len1 < lenf` and `len2 < lenf`. Otherwise, simply use `_fmpz_mod_poly_mul` instead.

Aliasing of `f` or `finv` and `res` is not permitted.

```
void fmpz_mod_poly_mulmod_preinv(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2,
   const fmpz_mod_poly_t f, const fmpz_mod_poly_t finv)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`. `finv` is the inverse of the reverse of `f`. It is required that `poly1` and `poly2` are reduced modulo `f`.

34.15 Powering

```
void _fmpz_mod_poly_pow(fmpz *rop, const fmpz *op, slong
   len, ulong e, const fmpz_t p)
```

Sets `res = poly^e`, assuming that $e > 1$ and `elen > 0`, and that `res` has space for `e*(len - 1)+ 1` coefficients. Does not support aliasing.

```
void fmpz_mod_poly_pow(fmpz_mod_poly_t rop, const
   fmpz_mod_poly_t op, ulong e)
```

Computes `res = poly^e`. If e is zero, returns one, so that in particular `0^0 = 1`.

```
void _fmpz_mod_poly_pow_trunc(fmpz * res, const fmpz *
   poly, ulong e, slong trunc, const fmpz_t p)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc > 0` and that `e > 1`. Aliasing is not permitted.

```
void fmpz_mod_poly_pow_trunc(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly, ulong e, slong trunc)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation.

```
void _fmpz_mod_poly_pow_trunc_binexp(fmpz * res, const fmpz
   * poly, ulong e, slong trunc, const fmpz_t p)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc > 0` and that `e > 1`. Aliasing is not permitted. Uses the binary exponentiation method.

```
void fmpz_mod_poly_pow_trunc_binexp(fmpz_mod_poly_t res,
   const fmpz_mod_poly_t poly, ulong e, slong trunc)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation. Uses the binary exponentiation method.

```
void _fmpz_mod_poly_powmod_ui_binexp(fmpz * res, const fmpz
   * poly, ulong e, const fmpz * f, slong lenf, const
   fmpz_t p)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fmpz_mod_poly_powmod_ui_binexp(fmpz_mod_poly_t res,
   const fmpz_mod_poly_t poly, ulong e, const
   fmpz_mod_poly_t f)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void _fmpz_mod_poly_powmod_ui_binexp_preinv(fmpz * res,
   const fmpz * poly, ulong e, const fmpz * f, slong lenf,
   const fmpz * finv, slong lenfinv, const fmpz_t p)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fmpz_mod_poly_powmod_ui_binexp_preinv(fmpz_mod_poly_t
   res, const fmpz_mod_poly_t poly, ulong e, const
   fmpz_mod_poly_t f, const fmpz_mod_poly_t finv)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fmpz_mod_poly_powmod_fmpz_binexp(fmpz * res, const
    fmpz * poly, const fmpz_t e, const fmpz * f, slong lenf,
    const fmpz_t p)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fmpz_mod_poly_powmod_fmpz_binexp(fmpz_mod_poly_t res,
    const fmpz_mod_poly_t poly, const fmpz_t e, const
    fmpz_mod_poly_t f)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void _fmpz_mod_poly_powmod_fmpz_binexp_preinv(fmpz * res,
    const fmpz * poly, const fmpz_t e, const fmpz * f, slong
    lenf, const fmpz* finv, slong lenfinv, const fmpz_t p)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void
    fmpz_mod_poly_powmod_fmpz_binexp_preinv(fmpz_mod_poly_t
    res, const fmpz_mod_poly_t poly, const fmpz_t e, const
    fmpz_mod_poly_t f, const fmpz_mod_poly_t finv)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fmpz_mod_poly_powmod_x_fmpz_preinv(fmpz * res, const
    fmpz_t e, const fmpz * f, slong lenf, const fmpz* finv,
    slong lenfinv, const fmpz_t p)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 2`. The output `res` must have room for `lenf - 1` coefficients.

```
void fmpz_mod_poly_powmod_x_fmpz_preinv(fmpz_mod_poly_t
    res, const fmpz_t e, const fmpz_mod_poly_t f, const
    fmpz_mod_poly_t finv)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

34.16 Division

```
void _fmpz_mod_poly_divrem_basecase(fmpz * Q, fmpz * R,
    const fmpz * A, slong lenA, const fmpz * B, slong lenB,
    const fmpz_t invB, const fmpz_t p)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible modulo p, and that `invB` is the inverse.

Assumes that $\text{len}(A), \text{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fmpz_mod_poly_divrem_basecase(fmpz_mod_poly_t Q,
   fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const
   fmpz_mod_poly_t B)
```

Computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible modulo p.

```
void _fmpz_mod_poly_divrem_newton_n_preinv (fmpz* Q, fmpz*
   R, const fmpz* A, slong lenA, const fmpz* B, slong lenB,
   const fmpz* Binv, slong lenBinv, const fmpz_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where A is of length `lenA` and B is of length `lenB`. We require that Q have space for `lenA - lenB + 1` coefficients. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$. The algorithm used is to call `div_newton_n_preinv()` and then multiply out and compute the remainder.

```
void fmpz_mod_poly_divrem_newton_n_preinv(fmpz_mod_poly_t
   Q, fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const
   fmpz_mod_poly_t B, const fmpz_mod_poly_t Binv)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to call `div_newton_n()` and then multiply out and compute the remainder.

```
void _fmpz_mod_poly_div_basecase(fmpz * Q, fmpz * R, const
   fmpz * A, slong lenA, const fmpz * B, slong lenB, const
   fmpz_t invB, const fmpz_t p)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets `(Q, lenA - lenB + 1)`.

Requires temporary space `(R, lenA)`. Allows aliasing only between A and R. Allows zero-padding in A but not in B. Assumes that the leading coefficient of B is a unit modulo p.

```
void fmpz_mod_poly_div_basecase(fmpz_mod_poly_t Q, const
   fmpz_mod_poly_t A, const fmpz_mod_poly_t B)
```

Notationally, computes Q, R such that $A = BQ+R$ with $0 \leq \text{len}(R) < \text{len}(B)$ assuming that the leading term of B is a unit.

```
void _fmpz_mod_poly_div_newton_n_preinv (fmpz* Q, const
   fmpz* A, slong lenA, const fmpz* B, slong lenB, const
   fmpz* Binv, slong lenBinv, const fmpz_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ+R$ with $\text{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`, but return only Q.

We require that Q have space for `lenA - lenB + 1` coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void fmpz_mod_poly_div_newton_n_preinv(fmpz_mod_poly_t Q,
    const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const
    fmpz_mod_poly_t Binv)
```

Notionally computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q.

We assume that the leading coefficient of B is a unit and that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
ulong fmpz_mod_poly_remove(fmpz_mod_poly_t f, const
    fmpz_mod_poly_t g)
```

Removes the highest possible power of `g` from `f` and returns the exponent.

```
void _fmpz_mod_poly_rem_basecase(fmpz * R, const fmpz * A,
    slong lenA, const fmpz * B, slong lenB, const fmpz_t
    invB, const fmpz_t p)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets `(R, lenA)`.

Allows aliasing only between A and R. Allows zero-padding in A but not in B. Assumes that the leading coefficient of B is a unit modulo p.

```
void fmpz_mod_poly_rem_basecase(fmpz_mod_poly_t R, const
    fmpz_mod_poly_t A, const fmpz_mod_poly_t B)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ assuming that the leading term of B is a unit.

```
void _fmpz_mod_poly_divrem_divconquer_recursive(fmpz * Q,
    fmpz * BQ, fmpz * W, const fmpz * A, const fmpz * B,
    slong lenB, const fmpz_t invB, const fmpz_t p)
```

Computes `(Q, lenB)`, `(BQ, 2 lenB - 1)` such that $BQ = B \times Q$ and $A = BQ + R$ where $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible modulo p, and that `invB` is the inverse.

Assumes $\text{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. Requires a temporary array `(W, 2 lenB - 1)`. No aliasing of input and output operands is allowed.

This function does not read the bottom $\text{len}(B) - 1$ coefficients from A, which means that they might not even need to exist in allocated memory.

```
void _fmpz_mod_poly_divrem_divconquer(fmpz * Q, fmpz * R,
    const fmpz * A, slong lenA, const fmpz * B, slong lenB,
    const fmpz_t invB, const fmpz_t p)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ and $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible modulo p, and that `invB` is the inverse.

Assumes $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. No aliasing of input and output operands is allowed.

```
void fmpz_mod_poly_divrem_divconquer(fmpz_mod_poly_t Q,
   fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const
   fmpz_mod_poly_t B)
```

Computes Q, R such that $A = BQ + R$ and $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that B is non-zero and that the leading coefficient of B is invertible modulo p.

```
void _fmpz_mod_poly_divrem(fmpz * Q, fmpz * R, const fmpz *
   A, slong lenA, const fmpz * B, slong lenB, const fmpz_t
   invB, const fmpz_t p)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ and $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that B is non-zero, that the leading coefficient of B is invertible modulo p and that `invB` is the inverse.

Assumes $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. No aliasing of input and output operands is allowed.

```
void fmpz_mod_poly_divrem(fmpz_mod_poly_t Q,
   fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const
   fmpz_mod_poly_t B)
```

Computes Q, R such that $A = BQ + R$ and $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that B is non-zero and that the leading coefficient of B is invertible modulo p.

```
void fmpz_mod_poly_divrem_f(fmpz_t f, fmpz_mod_poly_t Q,
   fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const
   fmpz_mod_poly_t B)
```

Either finds a non-trivial factor f of the modulus p, or computes Q, R such that $A = BQ + R$ and $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

If the leading coefficient of B is invertible in $\mathbf{Z}/(p)$, the division with remainder operation is carried out, Q and R are computed correctly, and f is set to 1. Otherwise, f is set to a non-trivial factor of p and Q and R are not touched.

Assumes that B is non-zero.

```
void _fmpz_mod_poly_rem(fmpz *R, const fmpz *A, slong lenA,
   const fmpz *B, slong lenB, const fmpz_t invB, const
   fmpz_t p)
```

Notationally, computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ and $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$, returning only the remainder part.

Assumes that B is non-zero, that the leading coefficient of B is invertible modulo p and that `invB` is the inverse.

Assumes $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. No aliasing of input and output operands is allowed.

```
void fmpz_mod_poly_rem(fmpz_mod_poly_t R, const
   fmpz_mod_poly_t A, const fmpz_mod_poly_t B)
```

Notationally, computes Q, R such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$, returning only the remainder part.

Assumes that B is non-zero and that the leading coefficient of B is invertible modulo p.

34.17 Power series inversion

```
void _fmpz_mod_poly_inv_series_newton(fmpz * Qinv, const
   fmpz * Q, slong n, const fmpz_t cinv, const fmpz_t p)
```

Sets `(Qinv, n)` to the inverse of `(Q, n)` modulo x^n, where $n \geq 1$, assuming that the bottom coefficient of Q is invertible modulo p and that its inverse is `cinv`.

```
void fmpz_mod_poly_inv_series_newton(fmpz_mod_poly_t Qinv,
   const fmpz_mod_poly_t Q, slong n)
```

Sets `Qinv` to the inverse of `Q` modulo x^n, where $n \geq 1$, assuming that the bottom coefficient of Q is a unit.

34.18 Greatest common divisor

```
void fmpz_mod_poly_make_monic(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly)
```

If `poly` is non-zero, sets `res` to `poly` divided by its leading coefficient. This assumes that the leading coefficient of `poly` is invertible modulo p.

Otherwise, if `poly` is zero, sets `res` to zero.

```
slong _fmpz_mod_poly_gcd_euclidean(fmpz *G, const fmpz *A,
   slong lenA, const fmpz *B, slong lenB, const fmpz_t
   invB, const fmpz_t p)
```

Sets G to the greatest common divisor of $(A, \text{len}(A))$ and $(B, \text{len}(B))$ and returns its length.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$ and that the vector G has space for sufficiently many coefficients.

Assumes that `invB` is the inverse of the leading coefficients of B modulo the prime number p.

```
void fmpz_mod_poly_gcd_euclidean(fmpz_mod_poly_t G, const
   fmpz_mod_poly_t A, const fmpz_mod_poly_t B)
```

Sets G to the greatest common divisor of A and B.

The algorithm used to compute G is the classical Euclidean algorithm.

In general, the greatest common divisor is defined in the polynomial ring $(\mathbf{Z}/(p\mathbf{Z}))[X]$ if and only if p is a prime number. Thus, this function assumes that p is prime.

```
slong _fmpz_mod_poly_gcd(fmpz *G, const fmpz *A, slong
   lenA, const fmpz *B, slong lenB, const fmpz_t invB,
   const fmpz_t p)
```

Sets G to the greatest common divisor of $(A, \text{len}(A))$ and $(B, \text{len}(B))$ and returns its length.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$ and that the vector G has space for sufficiently many coefficients.

Assumes that `invB` is the inverse of the leading coefficients of B modulo the prime number p.

```
void fmpz_mod_poly_gcd(fmpz_mod_poly_t G, const
   fmpz_mod_poly_t A, const fmpz_mod_poly_t B)
```

Sets G to the greatest common divisor of A and B.

In general, the greatest common divisor is defined in the polynomial ring $(\mathbf{Z}/(p\mathbf{Z}))[X]$ if and only if p is a prime number. Thus, this function assumes that p is prime.

```
slong _fmpz_mod_poly_gcd_euclidean_f(fmpz_t f, fmpz *G,
   const fmpz *A, slong lenA, const fmpz *B, slong lenB,
   const fmpz_t p)
```

Either sets $f = 1$ and G to the greatest common divisor of $(A, \operatorname{len}(A))$ and $(B, \operatorname{len}(B))$ and returns its length, or sets $f \in (1, p)$ to a non-trivial factor of p and leaves the contents of the vector $(G, lenB)$ undefined.

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$ and that the vector G has space for sufficiently many coefficients.

Does not support aliasing of any of the input arguments with any of the output argument.

```
void fmpz_mod_poly_gcd_euclidean_f(fmpz_t f,
   fmpz_mod_poly_t G, const fmpz_mod_poly_t A, const
   fmpz_mod_poly_t B)
```

Either sets $f = 1$ and G to the greatest common divisor of A and B, or $\in (1, p)$ to a non-trivial factor of p.

In general, the greatest common divisor is defined in the polynomial ring $(\mathbf{Z}/(p\mathbf{Z}))[X]$ if and only if p is a prime number.

```
slong _fmpz_mod_poly_gcd_f(fmpz_t f, fmpz *G, const fmpz
   *A, slong lenA, const fmpz *B, slong lenB, const fmpz_t
   p)
```

Either sets $f = 1$ and G to the greatest common divisor of $(A, \operatorname{len}(A))$ and $(B, \operatorname{len}(B))$ and returns its length, or sets $f \in (1, p)$ to a non-trivial factor of p and leaves the contents of the vector $(G, lenB)$ undefined.

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$ and that the vector G has space for sufficiently many coefficients.

Does not support aliasing of any of the input arguments with any of the output argument.

```
void fmpz_mod_poly_gcd_f(fmpz_t f, fmpz_mod_poly_t G, const
   fmpz_mod_poly_t A, const fmpz_mod_poly_t B)
```

Either sets $f = 1$ and G to the greatest common divisor of A and B, or $\in (1, p)$ to a non-trivial factor of p.

In general, the greatest common divisor is defined in the polynomial ring $(\mathbf{Z}/(p\mathbf{Z}))[X]$ if and only if p is a prime number.

```
slong _fmpz_mod_poly_xgcd_euclidean(fmpz *G, fmpz *S, fmpz
   *T, const fmpz *A, slong lenA, const fmpz *B, slong
   lenB, const fmpz_t invB, const fmpz_t p)
```

Computes the GCD of A and B together with cofactors S and T such that $SA+TB = G$. Returns the length of G.

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) \geq 1$ and $(\operatorname{len}(A), \operatorname{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that G have space for $\operatorname{len}(B)$ coefficients. Writes $\operatorname{len}(B) - 1$ and $\operatorname{len}(A) - 1$ coefficients to S and T, respectively. Note that, in fact, $\operatorname{len}(S) \leq \max(\operatorname{len}(B) - \operatorname{len}(G), 1)$ and $\operatorname{len}(T) \leq \max(\operatorname{len}(A) - \operatorname{len}(G), 1)$.

No aliasing of input and output operands is permitted.

```
void fmpz_mod_poly_xgcd_euclidean(fmpz_mod_poly_t G,
   fmpz_mod_poly_t S, fmpz_mod_poly_t T, const
   fmpz_mod_poly_t A, const fmpz_mod_poly_t B)
```

Computes the GCD of A and B. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

Polynomials `S` and `T` are computed such that `S*A + T*B = G`. The length of `S` will be at most `lenB` and the length of `T` will be at most `lenA`.

```
slong _fmpz_mod_poly_xgcd(fmpz *G, fmpz *S, fmpz *T, const
   fmpz *A, slong lenA, const fmpz *B, slong lenB, const
   fmpz_t invB, const fmpz_t p)
```

Computes the GCD of A and B together with cofactors S and T such that $SA+TB = G$. Returns the length of G.

Assumes that $\operatorname{len}(A) \geq \operatorname{len}(B) \geq 1$ and $(\operatorname{len}(A), \operatorname{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that G have space for $\operatorname{len}(B)$ coefficients. Writes $\operatorname{len}(B) - 1$ and $\operatorname{len}(A) - 1$ coefficients to S and T, respectively. Note that, in fact, $\operatorname{len}(S) \leq \max(\operatorname{len}(B) - \operatorname{len}(G), 1)$ and $\operatorname{len}(T) \leq \max(\operatorname{len}(A) - \operatorname{len}(G), 1)$.

No aliasing of input and output operands is permitted.

```
void fmpz_mod_poly_xgcd(fmpz_mod_poly_t G, fmpz_mod_poly_t
   S, fmpz_mod_poly_t T, const fmpz_mod_poly_t A, const
   fmpz_mod_poly_t B)
```

Computes the GCD of A and B. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

Polynomials `S` and `T` are computed such that `S*A + T*B = G`. The length of `S` will be at most `lenB` and the length of `T` will be at most `lenA`.

```
slong _fmpz_mod_poly_gcdinv(fmpz *G, fmpz *S, const fmpz
   *A, slong lenA, const fmpz *B, slong lenB, const fmpz_t
   p)
```

Computes `(G, lenA)`, `(S, lenB-1)` such that $G \cong SA \pmod{B}$, returning the actual length of G.

Assumes that $0 < \operatorname{len}(A) < \operatorname{len}(B)$.

```
void fmpz_mod_poly_gcdinv(fmpz_mod_poly_t G,
   fmpz_mod_poly_t S, const fmpz_mod_poly_t A, const
   fmpz_mod_poly_t B)
```

Computes polynomials G and S, both reduced modulo B, such that $G \cong SA \pmod{B}$, where B is assumed to have $\operatorname{len}(B) \geq 2$.

In the case that $A = 0 \pmod{B}$, returns $G = S = 0$.

```
int _fmpz_mod_poly_invmod(fmpz *A, const fmpz *B, slong
   lenB, const fmpz *P, slong lenP, const fmpz_t p)
```

Attempts to set `(A, lenP-1)` to the inverse of `(B, lenB)` modulo the polynomial `(P, lenP)`. Returns 1 if `(B, lenB)` is invertible and 0 otherwise.

Assumes that $0 < \operatorname{len}(B) < \operatorname{len}(P)$, and hence also $\operatorname{len}(P) \geq 2$, but supports zero-padding in `(B, lenB)`.

Does not support aliasing.

Assumes that p is a prime number.

```
int fmpz_mod_poly_invmod(fmpz_mod_poly_t A, const
   fmpz_mod_poly_t B, const fmpz_mod_poly_t P)
```

Attempts to set A to the inverse of B modulo P in the polynomial ring $(\mathbf{Z}/p\mathbf{Z})[X]$, where we assume that p is a prime number.

If $\deg(P) < 2$, raises an exception.

If the greatest common divisor of B and P is 1, returns 1 and sets A to the inverse of B. Otherwise, returns 0 and the value of A on exit is undefined.

34.19 Derivative

```
void _fmpz_mod_poly_derivative(fmpz *res, const fmpz *poly,
   slong len, const fmpz_t p)
```

Sets `(res, len - 1)` to the derivative of `(poly, len)`. Also handles the cases where `len` is 0 or 1 correctly. Supports aliasing of `res` and `poly`.

```
void fmpz_mod_poly_derivative(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly)
```

Sets `res` to the derivative of `poly`.

34.20 Evaluation

```
void _fmpz_mod_poly_evaluate_fmpz(fmpz_t res, const fmpz
   *poly, slong len, const fmpz_t a, const fmpz_t p)
```

Evaluates the polynomial `(poly, len)` at the integer a and sets `res` to the result. Aliasing between `res` and a or any of the coefficients of `poly` is not supported.

```
void fmpz_mod_poly_evaluate_fmpz(fmpz_t res, const
   fmpz_mod_poly_t poly, const fmpz_t a)
```

Evaluates the polynomial `poly` at the integer a and sets `res` to the result.

As expected, aliasing between `res` and a is supported. However, `res` may not be aliased with a coefficient of `poly`.

34.21 Multipoint evaluation

```
void _fmpz_mod_poly_evaluate_fmpz_vec_iter(fmpz * ys, const
   fmpz * coeffs, slong len, const fmpz * xs, slong n,
   const fmpz_t mod)
```

Evaluates (`coeffs`, `len`) at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

Uses Horner's method iteratively.

```
void fmpz_mod_poly_evaluate_fmpz_vec_iter(fmpz * ys, const
   fmpz_mod_poly_t poly, const fmpz * xs, slong n)
```

Evaluates `poly` at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

Uses Horner's method iteratively.

```
void _fmpz_mod_poly_evaluate_fmpz_vec_fast_precomp(fmpz *
   vs, const fmpz * poly, slong plen, fmpz_poly_struct *
   const * tree, slong len, const fmpz_t mod)
```

Evaluates (`poly`, `plen`) at the `len` values given by the precomputed subproduct tree `tree`.

```
void _fmpz_mod_poly_evaluate_fmpz_vec_fast(fmpz * ys, const
   fmpz * poly, slong plen, const fmpz * xs, slong n, const
   fmpz_t mod)
```

Evaluates (`coeffs`, `len`) at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

Uses fast multipoint evaluation, building a temporary subproduct tree.

```
void fmpz_mod_poly_evaluate_fmpz_vec_fast(fmpz * ys, const
   fmpz_mod_poly_t poly, const fmpz * xs, slong n)
```

Evaluates `poly` at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

Uses fast multipoint evaluation, building a temporary subproduct tree.

```
void _fmpz_mod_poly_evaluate_fmpz_vec(fmpz * ys, const fmpz
   * coeffs, slong len, const fmpz * xs, slong n, const
   fmpz_t mod)
```

Evaluates (`coeffs`, `len`) at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

```
void fmpz_mod_poly_evaluate_fmpz_vec(fmpz * ys, const
   fmpz_mod_poly_t poly, const fmpz * xs, slong n)
```

Evaluates `poly` at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

34.22 Composition

```
void _fmpz_mod_poly_compose_horner(fmpz *res, const fmpz
   *poly1, slong len1, const fmpz *poly2, slong len2, const
   fmpz_t p)
```

Sets `res` to the composition of `(poly1, len1)` and `(poly2, len2)` using Horner's algorithm.

Assumes that `res` has space for `(len1-1)*(len2-1)+ 1` coefficients, although in $\mathbf{Z}_p[X]$ this might not actually be the length of the resulting polynomial when p is not a prime.

Assumes that `poly1` and `poly2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fmpz_mod_poly_compose_horner(fmpz_mod_poly_t res,
   const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2)
```

Sets `res` to the composition of `poly1` and `poly2` using Horner's algorithm.

To be precise about the order of composition, denoting `res`, `poly1`, and `poly2` by f, g, and h, respectively, sets $f(t) = g(h(t))$.

```
void _fmpz_mod_poly_compose_divconquer(fmpz *res, const
   fmpz *poly1, slong len1, const fmpz *poly2, slong len2,
   const fmpz_t p)
```

Sets `res` to the composition of `(poly1, len1)` and `(poly2, len2)` using a divide and conquer algorithm which takes out factors of `poly2` raised to 2^i where possible.

Assumes that `res` has space for `(len1-1)*(len2-1)+ 1` coefficients, although in $\mathbf{Z}_p[X]$ this might not actually be the length of the resulting polynomial when p is not a prime.

Assumes that `poly1` and `poly2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fmpz_mod_poly_compose_divconquer(fmpz_mod_poly_t res,
   const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2)
```

Sets `res` to the composition of `poly1` and `poly2` using a divide and conquer algorithm which takes out factors of `poly2` raised to 2^i where possible.

To be precise about the order of composition, denoting `res`, `poly1`, and `poly2` by f, g, and h, respectively, sets $f(t) = g(h(t))$.

```
void _fmpz_mod_poly_compose(fmpz *res, const fmpz *poly1,
   slong len1, const fmpz *poly2, slong len2, const fmpz_t
   p)
```

Sets `res` to the composition of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `res` has space for `(len1-1)*(len2-1)+ 1` coefficients, although in $\mathbf{Z}_p[X]$ this might not actually be the length of the resulting polynomial when p is not a prime.

Assumes that `poly1` and `poly2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fmpz_mod_poly_compose(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2)
```

Sets `res` to the composition of `poly1` and `poly2`.

To be precise about the order of composition, denoting `res`, `poly1`, and `poly2` by f, g, and h, respectively, sets $f(t) = g(h(t))$.

34.23 Modular composition

```
void _fmpz_mod_poly_compose_mod(fmpz * res, const fmpz * f,
   slong lenf, const fmpz * g, const fmpz * h, slong lenh,
   const fmpz_t p)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```
void fmpz_mod_poly_compose_mod(fmpz_mod_poly_t res, const
   fmpz_mod_poly_t f, const fmpz_mod_poly_t g, const
   fmpz_mod_poly_t h)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero.

```
void _fmpz_mod_poly_compose_mod_horner(fmpz * res, const
   fmpz * f, slong lenf, const fmpz * g, const fmpz * h,
   slong lenh, const fmpz_t p)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fmpz_mod_poly_compose_mod_horner(fmpz_mod_poly_t res,
   const fmpz_mod_poly_t f, const fmpz_mod_poly_t g, const
   fmpz_mod_poly_t h)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero. The algorithm used is Horner's rule.

```
void _fmpz_mod_poly_compose_mod_brent_kung(fmpz * res,
   const fmpz * f, slong len1, const fmpz * g, const fmpz *
   h, slong len3, const fmpz_t p)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fmpz_mod_poly_compose_mod_brent_kung( fmpz_mod_poly_t
   res, const fmpz_mod_poly_t f, const fmpz_mod_poly_t g,
   const fmpz_mod_poly_t h)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. The algorithm used is the Brent-Kung matrix algorithm.

```
void _fmpz_mod_poly_reduce_matrix_mod_poly (fmpz_mat_t A,
   const fmpz_mat_t B, const fmpz_mod_poly_t f)
```

Sets the ith row of `A` to the reduction of the ith row of B modulo f for $i = 1, \ldots, \sqrt{\deg(f)}$. We require B to be at least a $\sqrt{\deg(f)} \times \deg(f)$ matrix and f to be nonzero.

```
void _fmpz_mod_poly_precompute_matrix (fmpz_mat_t A, const
   fmpz * f, const fmpz * g, slong leng, const fmpz * ginv,
   slong lenginv, const fmpz_t p)
```

Sets the ith row of `A` to f^i modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of `g` and g to be nonzero.

```
void fmpz_mod_poly_precompute_matrix(fmpz_mat_t A, const
   fmpz_mod_poly_t f, const fmpz_mod_poly_t g, const
   fmpz_mod_poly_t ginv)
```

Sets the ith row of `A` to f^i modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of `g`.

```
void
   _fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv(fmpz
   * res, const fmpz * f, slong lenf, const fmpz_mat_t A,
   const fmpz * h, slong lenh, const fmpz * hinv, slong
   lenhinv, const fmpz_t p)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero. We require that the ith row of A contains g^i for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void
    fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv(fmpz_mod_poly_t
    res, const fmpz_mod_poly_t f, const fmpz_mat_t A, const
    fmpz_mod_poly_t h, const fmpz_mod_poly_t hinv)
```

Sets `res` to the composition $f(g)$ modulo h. We require that the ith row of A contains g^i for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo h for fixed g and h.

```
void _fmpz_mod_poly_compose_mod_brent_kung_preinv(fmpz *
    res, const fmpz * f, slong lenf, const fmpz * g, const
    fmpz * h, slong lenh, const fmpz * hinv, slong lenhinv,
    const fmpz_t p)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void
    fmpz_mod_poly_compose_mod_brent_kung_preinv(fmpz_mod_poly_t
    res, const fmpz_mod_poly_t f, const fmpz_mod_poly_t g,
    const fmpz_mod_poly_t h, const fmpz_mod_poly_t hinv)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The algorithm used is the Brent-Kung matrix algorithm.

34.24 Subproduct trees

```
fmpz_poly_struct ** _fmpz_mod_poly_tree_alloc(slong len)
```

Allocates space for a subproduct tree of the given length, having linear factors at the lowest level.

```
void _fmpz_mod_poly_tree_free(fmpz_poly_struct ** tree,
    slong len)
```

Free the allocated space for the subproduct.

```
void _fmpz_mod_poly_tree_build(fmpz_poly_struct ** tree,
    const fmpz * roots, slong len, const fmpz_t mod)
```

Builds a subproduct tree in the preallocated space from the `len` monic linear factors $(x - r_i)$ where r_i are given by `roots`. The top level product is not computed.

34.25 Radix conversion

The following functions provide the functionality to solve the radix conversion problems for polynomials, which is to express a polynomial $f(X)$ with respect to a given radix $r(X)$ as

$$f(X) = \sum_{i=0}^{N} b_i(X) r(X)^i$$

where $N = \lfloor \deg(f) / \deg(r) \rfloor$.

The algorithm implemented here is a recursive one, which performs Euclidean divisions by powers of r of the form r^{2^i}, and it has time complexity $\Theta(\deg(f) \log \deg(f))$.

It facilitates the repeated use of precomputed data, namely the powers of r and their power series inverses. This data is stored in objects of type `fmpz_mod_poly_radix_t` and it is computed using the function `fmpz_mod_poly_radix_init()`, which only depends on r and an upper bound on the degree of f.

```
void _fmpz_mod_poly_radix_init(fmpz **Rpow, fmpz **Rinv,
    const fmpz *R, slong lenR, slong k, const fmpz_t invL,
    const fmpz_t p)
```

Computes powers of R of the form R^{2^i} and their Newton inverses modulo $x^{2^i \deg(R)}$ for $i = 0, \ldots, k-1$.

Assumes that the vectors `Rpow[i]` and `Rinv[i]` have space for $2^i \deg(R) + 1$ and $2^i \deg(R)$ coefficients, respectively.

Assumes that the polynomial R is non-constant, i.e. $\deg(R) \geq 1$.

Assumes that the leading coefficient of R is a unit and that the argument `invL` is the inverse of the coefficient modulo p.

The argument p is the modulus, which in p-adic applications is typically a prime power, although this is not necessary. Here, we only assume that $p \geq 2$.

Note that this precomputed data can be used for any F such that $\text{len}(F) \leq 2^k \deg(R)$.

```
void fmpz_mod_poly_radix_init(fmpz_mod_poly_radix_t D,
    const fmpz_mod_poly_t R, slong degF)
```

Carries out the precomputation necessary to perform radix conversion to radix R for polynomials F of degree at most `degF`.

Assumes that R is non-constant, i.e. $\deg(R) \geq 1$, and that the leading coefficient is a unit.

```
void _fmpz_mod_poly_radix(fmpz **B, const fmpz *F, fmpz
    **Rpow, fmpz **Rinv, slong degR, slong k, slong i, fmpz
    *W, const fmpz_t p)
```

This is the main recursive function used by the function `fmpz_mod_poly_radix()`.

Assumes that, for all $i = 0, \ldots, N$, the vector `B[i]` has space for $\deg(R)$ coefficients.

The variable k denotes the factors of r that have previously been counted for the polynomial F, which is assumed to have length $2^{i+1} \deg(R)$, possibly including zero-padding.

Assumes that W is a vector providing temporary space of length $\text{len}(F) = 2^{i+1} \deg(R)$.

The entire computation takes place over $\mathbf{Z}/p\mathbf{Z}$, where $p \geq 2$ is a natural number.

Thus, the top level call will have F as in the original problem, and $k = 0$.

```
void fmpz_mod_poly_radix(fmpz_mod_poly_struct **B, const
   fmpz_mod_poly_t F, const fmpz_mod_poly_radix_t D)
```

Given a polynomial F and the precomputed data D for the radix R, computes polynomials $B_0, \ldots, B_N$ of degree less than $\deg(R)$ such that

$$F = B_0 + B_1 R + \cdots + B_N R^N,$$

where necessarily $N = \lfloor \deg(F) / \deg(R) \rfloor$.

Assumes that R is non-constant, i.e. $\deg(R) \geq 1$, and that the leading coefficient is a unit.

34.26 Input and output

The printing options supported by this module are very similar to what can be found in the two related modules `fmpz_poly` and `nmod_poly`.

Consider, for example, the polynomial $f(x) = 5x^3 + 2x + 1$ in $(\mathbf{Z}/6\mathbf{Z})[x]$. Its simple string representation is `"4 6 1 2 0 5"`, where the first two numbers denote the length of the polynomial and the modulus. The pretty string representation is `"5*x^3+2*x+1"`.

```
int _fmpz_mod_poly_fprint(FILE * file, const fmpz *poly,
   slong len, const fmpz_t p)
```

Prints the polynomial `(poly, len)` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_mod_poly_fprint(FILE * file, const fmpz_mod_poly_t
   poly)
```

Prints the polynomial to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_mod_poly_fprint_pretty(FILE * file, const
   fmpz_mod_poly_t poly, const char * x)
```

Prints the pretty representation of `(poly, len)` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_mod_poly_print(const fmpz_mod_poly_t poly)
```

Prints the polynomial to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_mod_poly_print_pretty(const fmpz_mod_poly_t poly,
   const char * x)
```

Prints the pretty representation of `poly` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

§35. fmpz_mod_poly_factor

Factorisation of polynomials over $\mathbf{Z}/n\mathbf{Z}$ for general moduli

---

<<<<<<< HEAD The `fmpz_mod_poly_factor` module is included automatically with `fmpz_mod_poly.h`. One should not try to include ======= The `fmpz_mod_poly_factor` module is included automatically when one includes `fmpz_mod_poly.h`. One should not try to include >>>>>>> mhansen/fq `fmpz_mod_poly_factor.h` directly.

35.1 Factorisation

```
void fmpz_mod_poly_factor_init(fmpz_mod_poly_factor_t fac)
```

Initialises `fac` for use. An `fmpz_mod_poly_factor_t` represents a polynomial in factorised form as a product of polynomials with associated exponents.

```
void fmpz_mod_poly_factor_clear(fmpz_mod_poly_factor_t fac)
```

Frees all memory associated with `fac`.

```
void fmpz_mod_poly_factor_realloc(fmpz_mod_poly_factor_t
   fac, slong alloc)
```

Reallocates the factor structure to provide space for precisely `alloc` factors.

```
void fmpz_mod_poly_factor_fit_length(fmpz_mod_poly_factor_t
   fac, slong len)
```

Ensures that the factor structure has space for at least `len` factors. This function takes care of the case of repeated calls by always, at least doubling the number of factors the structure can hold.

```
void fmpz_mod_poly_factor_set(fmpz_mod_poly_factor_t res,
   const fmpz_mod_poly_factor_t fac)
```

Sets `res` to the same factorisation as `fac`.

```
void fmpz_mod_poly_factor_print(const
   fmpz_mod_poly_factor_t fac)
```

Prints the entries of `fac` to standard output.

```
void fmpz_mod_poly_factor_insert(fmpz_mod_poly_factor_t
   fac, const fmpz_mod_poly_t poly, slong exp)
```

Inserts the factor `poly` with multiplicity `exp` into the factorisation `fac`.

If `fac` already contains `poly`, then `exp` simply gets added to the exponent of the existing entry.

```
void fmpz_mod_poly_factor_concat(fmpz_mod_poly_factor_t
   res, const fmpz_mod_poly_factor_t fac)
```

Concatenates two factorisations.

This is equivalent to calling `fmpz_mod_poly_factor_insert()` repeatedly with the individual factors of `fac`.

Does not support aliasing between `res` and `fac`.

```
void fmpz_mod_poly_factor_pow(fmpz_mod_poly_factor_t fac,
   slong exp)
```

Raises `fac` to the power `exp`.

```
int fmpz_mod_poly_is_irreducible(const fmpz_mod_poly_t f)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0.

```
int fmpz_mod_poly_is_irreducible_ddf(const fmpz_mod_poly_t
   f)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

```
int fmpz_mod_poly_is_irreducible_rabin(const
   fmpz_mod_poly_t f)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses Rabin irreducibility test.

```
int _fmpz_mod_poly_is_squarefree(const fmpz * f, slong len,
   const fmpz_t p)
```

Returns 1 if `(f, len)` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

```
int fmpz_mod_poly_is_squarefree(const fmpz_mod_poly_t f)
```

Returns 1 if `f` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

```
int fmpz_mod_poly_factor_equal_deg_prob(fmpz_mod_poly_t
   factor, flint_rand_t state, const fmpz_mod_poly_t pol,
   slong d)
```

Probabilistic equal degree factorisation of `pol` into irreducible factors of degree `d`. If it passes, a factor is placed in `factor` and 1 is returned, otherwise 0 is returned and the value of factor is undetermined.

Requires that `pol` be monic, non-constant and squarefree.

```
void fmpz_mod_poly_factor_equal_deg(fmpz_mod_poly_factor_t
   factors, const fmpz_mod_poly_t pol, slong d)
```

Assuming `pol` is a product of irreducible factors all of degree `d`, finds all those factors and places them in factors. Requires that `pol` be monic, non-constant and squarefree.

```
void
    fmpz_mod_poly_factor_distinct_deg(fmpz_mod_poly_factor_t
    res, const fmpz_mod_poly_t poly, slong * const *degs)
```

Factorises a monic non-constant squarefree polynomial `poly` of degree n into factors $f[d]$ such that for $1 \leq d \leq n$ $f[d]$ is the product of the monic irreducible factors of `poly` of degree d. Factors $f[d]$ are stored in `res`, and the degree d of the irreducible factors is stored in `degs` in the same order as the factors.

Requires that `degs` has enough space for $(n/2) + 1 * sizeof(slong)$.

```
void fmpz_mod_poly_factor_squarefree(fmpz_mod_poly_factor_t
    res, const fmpz_mod_poly_t f)
```

Sets `res` to a squarefree factorization of `f`.

```
void fmpz_mod_poly_factor(fmpz_mod_poly_factor_t res, const
    fmpz_mod_poly_t f)
```

Factorises a non-constant polynomial `f` into monic irreducible factors choosing the best algorithm for given modulo and degree. Choise is based on heuristic measurments.

```
void
    fmpz_mod_poly_factor_cantor_zassenhaus(fmpz_mod_poly_factor_t
    res, const fmpz_mod_poly_t f)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the Cantor-Zassenhaus algorithm.

```
void
    fmpz_mod_poly_factor_kaltofen_shoup(fmpz_mod_poly_factor_t
    res, const fmpz_mod_poly_t poly)
```

Factorises a non-constant polynomial `poly` into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a "baby step/giant step"İ strategy for the distinct-degree factorization step.

```
void fmpz_mod_poly_factor_berlekamp(fmpz_mod_poly_factor_t
    factors, const fmpz_mod_poly_t f)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the Berlekamp algorithm.

§36. fq

Finite fields of arbitrary characteristic

We represent an element of the finite field $\mathbb{F}_{p^n} \cong \mathbb{F}_p[X]/(f(X))$, where $f(X) \in \mathbb{F}_p[X]$ is a monic, irreducible polynomial of degree n, as a polynomial in $\mathbb{F}_p[X]$ of degree less than n. The underlying data structure is an `fmpz_poly_t`.

The default choice for $f(X)$ is the Conway polynomial for the pair (p, n). Frank Luebeck's data base of Conway polynomials is made available in the file `qadic/CPimport.txt`. If a Conway polynomial is not available, then a random irreducible polynomial will be chosen for $f(X)$. Additionally, the user is able to supply their own $f(X)$.

36.1 Context Management

```
void fq_ctx_init(fq_ctx_t ctx, const fmpz_t p, slong d,
    const char *var)
```

Initialises the context for prime p and extension degree d, with name `var` for the generator. By default, it will try use a Conway polynomial; if one is not available, a random irreducible polynomial will be used.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

```
int _fq_ctx_init_conway(fq_ctx_t ctx, const fmpz_t p, slong
    d, const char *var)
```

Attempts to initialise the context for prime p and extension degree d, with name `var` for the generator using a Conway polynomial for the modulus.

Returns 1 if the Conway polynomial is in the database for the given size and the initialization is successful; otherwise, returns 0.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

```
void fq_ctx_init_conway(fq_ctx_t ctx, const fmpz_t p, slong
    d, const char *var)
```

Initialises the context for prime p and extension degree d, with name `var` for the generator using a Conway polynomial for the modulus.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

```
void fq_ctx_init_modulus(fq_ctx_t ctx, fmpz_mod_poly_t
   modulus, const char *var)
```

Initialises the context for given `modulus` with name `var` for the generator.

Assumes that `modulus` is and irreducible polynomial over $\mathbf{F}_p$.

Assumes that the string `var` is a null-terminated string of length at least one.

```
void fq_ctx_clear(fq_ctx_t ctx)
```

Clears all memory that has been allocated as part of the context.

```
long fq_ctx_degree(const fq_ctx_t ctx)
```

Returns the degree of the field extension $[\mathbf{F}_q : \mathbf{F}_p]$, which is equal to $\log_p q$.

```
fmpz * fq_ctx_prime(const fq_ctx_t ctx)
```

Returns a pointer to the prime p in the context.

```
void fq_ctx_order(fmpz_t f, const fq_ctx_t ctx)
```

Sets f to be the size of the finite field.

```
int fq_ctx_fprint(FILE * file, const fq_ctx_t ctx)
```

Prints the context information to `file`. Returns 1 for a success and a negative number for an error.

```
void fq_ctx_print(const fq_ctx_t ctx)
```

Prints the context information to `stdout`.

```
void fq_ctx_randtest(fq_ctx_t ctx)
```

Initializes `ctx` to a random finite field. Assumes that `fq_ctx_init` as not been called on `ctx` already.

```
void fq_ctx_randtest_reducible(fq_ctx_t ctx)
```

Initializes `ctx` to a random extension of a prime field. The modulus may or may not be irreducible. Assumes that `fq_ctx_init` as not been called on `ctx` already.

36.2 Memory management

```
void fq_init(fq_t rop, const fq_ctx_t ctx)
```

Initialises the element `rop`, setting its value to 0.

```
void fq_init2(fq_t rop, const fq_ctx_t ctx)
```

Initialises `poly` with at least enough space for it to be an element of `ctx` and sets it to 0.

```
void fq_clear(fq_t rop, const fq_ctx_t ctx)
```

Clears the element `rop`.

```
void _fq_sparse_reduce(fmpz *R, slong lenR, const fq_ctx_t
   ctx)
```

Reduces `(R, lenR)` modulo the polynomial f given by the modulus of `ctx`.

```
void _fq_dense_reduce(fmpz *R, slong lenR, const fq_ctx_t
    ctx)
```

Reduces `(R, lenR)` modulo the polynomial f given by the modulus of `ctx` using Newton division.

```
void _fq_reduce(fmpz *r, slong lenR, const fq_ctx_t ctx)
```

Reduces `(R, lenR)` modulo the polynomial f given by the modulus of `ctx`. Does either sparse or dense reduction based on `ctx->sparse_modulus`.

```
void fq_reduce(fq_t rop, const fq_ctx_t ctx)
```

Reduces the polynomial `rop` as an element of $\mathbf{F}_p[X]/(f(X))$.

36.3 Basic arithmetic

```
void fq_add(fq_t rop, const fq_t op1, const fq_t op2, const
    fq_ctx_t ctx)
```

Sets `rop` to the sum of `op1` and `op2`.

```
void fq_sub(fq_t rop, const fq_t op1, const fq_t op2, const
    fq_ctx_t ctx)
```

Sets `rop` to the difference of `op1` and `op2`.

```
void fq_sub_one(fq_t rop, const fq_t op1, const fq_ctx_t
    ctx)
```

Sets `rop` to the difference of `op1` and 1.

```
void fq_neg(fq_t rop, const fq_t op, const fq_ctx_t ctx)
```

Sets `rop` to the negative of `op`.

```
void fq_mul(fq_t rop, const fq_t op1, const fq_t op2, const
    fq_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`, reducing the output in the given context.

```
void fq_mul_fmpz(fq_t rop, const fq_t op, const fmpz_t x,
    const fq_ctx_t ctx)
```

Sets `rop` to the product of `op` and x, reducing the output in the given context.

```
void fq_mul_si(fq_t rop, const fq_t op, slong x, const
    fq_ctx_t ctx)
```

Sets `rop` to the product of `op` and x, reducing the output in the given context.

```
void fq_mul_ui(fq_t rop, const fq_t op, ulong x, const
    fq_ctx_t ctx)
```

Sets `rop` to the product of `op` and x, reducing the output in the given context.

```
void fq_sqr(fq_t rop, const fq_t op, const fq_ctx_t ctx)
```

Sets `rop` to the square of `op`, reducing the output in the given context.

```
void _fq_inv(fmpz *rop, const fmpz *op, slong len, const
    fq_ctx_t ctx)
```

Sets `(rop, d)` to the inverse of the non-zero element `(op, len)`.

```
void fq_inv(fq_t rop, const fq_t op, const fq_ctx_t ctx)
```

Sets `rop` to the inverse of the non-zero element `op`.

```
void _fq_pow(fmpz *rop, const fmpz *op, slong len, const
   fmpz_t e, const fq_ctx_t ctx)
```

Sets `(rop, 2*d-1)` to `(op,len)` raised to the power e, reduced modulo $f(X)$, the modulus of `ctx`.

Assumes that $e \geq 0$ and that `len` is positive and at most d.

Although we require that `rop` provides space for $2d - 1$ coefficients, the output will be reduced modulo $f(X)$, which is a polynomial of degree d.

Does not support aliasing.

```
void fq_pow(fq_t rop, const fq_t op, const fmpz_t e, const
   fq_ctx_t ctx)
```

Sets `rop` the `op` raised to the power e.

Currently assumes that $e \geq 0$.

Note that for any input `op`, `rop` is set to 1 whenever $e = 0$.

```
void fq_pow_ui(fq_t rop, const fq_t op, const ulong e,
   const fq_ctx_t ctx)
```

Sets `rop` the `op` raised to the power e.

Currently assumes that $e \geq 0$.

Note that for any input `op`, `rop` is set to 1 whenever $e = 0$.

36.4 Roots

```
void fq_pth_root(fq_t rop, const fq_t op1, const fq_ctx_t
   ctx)
```

Sets `rop` to a p^{th} root root of `op1`. Currently, this computes the root by raising `op1` to p^{d-1} where d is the degree of the extension.

36.5 Output

```
int fq_fprint_pretty(FILE *file, const fq_t op, const
   fq_ctx_t ctx)
```

Prints a pretty representation of `op` to `file`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

```
int fq_print_pretty(const fq_t op, const fq_ctx_t ctx)
```

Prints a pretty representation of `op` to `stdout`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

```
void fq_fprint(FILE * file, const fq_t op, const fq_ctx_t
    ctx)
```

Prints a representation of `op` to `file`.

For further details on the representation used, see `fmpz_mod_poly_fprint()`.

```
void fq_print(const fq_t op, const fq_ctx_t ctx)
```

Prints a representation of `op` to `stdout`.

For further details on the representation used, see `fmpz_mod_poly_print()`.

```
char * fq_get_str(const fq_t op, const fq_ctx_t ctx)
```

Returns the plain FLINT string representation of the element `op`.

```
char * fq_get_str_pretty(const fq_t op, const fq_ctx_t ctx)
```

Returns a pretty representation of the element `op` using the null-terminated string `x` as the variable name.

36.6 Randomisation

```
void fq_randtest(fq_t rop, flint_rand_t state, const
    fq_ctx_t ctx)
```

Generates a random element of $\mathbb{F}_q$.

```
void fq_randtest_not_zero(fq_t rop, flint_rand_t state,
    const fq_ctx_t ctx)
```

Generates a random non-zero element of $\mathbb{F}_q$.

```
void fq_randtest_dense(fq_t rop, flint_rand_t state, const
    fq_ctx_t ctx)
```

Generates a random element of $\mathbb{F}_q$ which has an underlying polynomial with dense coefficients.

36.7 Assignments and conversions

```
void fq_set(fq_t rop, const fq_t op, const fq_ctx_t ctx)
```

Sets `rop` to `op`.

```
void fq_set_ui(fq_t rop, const ulong x, const fq_ctx_t ctx)
```

Sets `rop` to `x`, considered as an element of $\mathbb{F}_p$.

```
void fq_set_fmpz(fq_t rop, const fmpz_t x, const fq_ctx_t
    ctx)
```

Sets `rop` to `x`, considered as an element of $\mathbb{F}_p$.

```
void fq_swap(fq_t op1, fq_t op2, const fq_ctx_t ctx)
```

Swaps the two elements `op1` and `op2`.

```
void fq_zero(fq_t rop, const fq_ctx_t ctx)
```

Sets `rop` to zero.

```
void fq_one(fq_t rop, const fq_ctx_t ctx)
```

Sets `rop` to one, reduced in the given context.

```
void fq_gen(fq_t rop, const fq_ctx_t ctx)
```

Sets `rop` to a multiplicative generator for the finite field.

36.8 Comparison

```
int fq_is_zero(const fq_t op, const fq_ctx_t ctx)
```

Returns whether `op` is equal to zero.

```
int fq_is_one(const fq_t op, const fq_ctx_t ctx)
```

Returns whether `op` is equal to one.

```
int fq_equal(const fq_t op1, const fq_t op2, const fq_ctx_t
    ctx)
```

Returns whether `op1` and `op2` are equal.

```
int fq_is_invertible(const fq_t op, const fq_ctx_t ctx)
```

Returns whether `op` is an invertible element.

```
int fq_is_invertible_f(fq_t f, const fq_t op, const
    fq_ctx_t ctx)
```

Returns whether `op` is an invertible element. If it is not, then `f` is set of a factor of the modulus.

36.9 Special functions

```
void _fq_trace(fmpz_t rop, const fmpz *op, slong len, const
    fq_ctx_t ctx)
```

Sets `rop` to the trace of the non-zero element `(op, len)` in $\mathbf{F}_q$.

```
void fq_trace(fq_t rop, const fq_t op, const fq_ctx_t ctx)
```

Sets `rop` to the trace of `op`.

For an element $a \in \mathbb{F}_q$, multiplication by a defines a $\mathbb{F}_p$-linear map on $\mathbb{F}_q$. We define the trace of a as the trace of this map. Equivalently, if Σ generates $\mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)$ then the trace of a is equal to $\sum_{i=0}^{d-1} \Sigma^i(a)$, where $d = \log_p q$.

```
void _fq_norm(fmpz_t rop, const fmpz *op, slong len, const
    fq_ctx_t ctx)
```

Sets `rop` to the norm of the non-zero element `(op, len)` in $\mathbf{F}_q$.

```
void fq_norm(fq_t rop, const fq_t op, const fq_ctx_t ctx)
```

Computes the norm of `op`.

For an element $a \in \mathbb{F}_q$, multiplication by a defines a $\mathbb{F}_p$-linear map on $\mathbb{F}_q$. We define the norm of a as the determinant of this map. Equivalently, if Σ generates $\mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)$ then the trace of a is equal to $\prod_{i=0}^{d-1} \Sigma^i(a)$, where $d = \dim_{\mathbb{F}_p}(\mathbb{F}_q)$.

Algorithm selection is automatic depending on the input.

```
void _fq_frobenius(fmpz *rop, const fmpz *op, slong len,
    slong e, const fq_ctx_t ctx)
```

Sets `(rop, 2d-1)` to the image of `(op, len)` under the Frobenius operator raised to the e-th power, assuming that neither `op` nor `e` are zero.

```
void fq_frobenius(fq_t rop, const fq_t op, slong e, const
    fq_ctx_t ctx)
```

Evaluates the homomorphism Σ^e at `op`.

Recall that $\mathbb{F}_q/\mathbb{F}_p$ is Galois with Galois group $\langle\sigma\rangle$, which is also isomorphic to $\mathbf{Z}/d\mathbf{Z}$, where $\sigma \in \mathrm{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ is the Frobenius element $\sigma\colon x \mapsto x^p$.

36.10 Bit packing

```
void fq_bit_pack(fmpz_t f, const fq_t op, mp_bitcnt_t
    bit_size, const fq_ctx_t ctx)
```

Packs `op` into bitfields of size `bit_size`, writing the result to `f`.

```
void fq_bit_unpack(fq_t rop, const fmpz_t f, mp_bitcnt_t
    bit_size, const fq_ctx_t ctx)
```

Unpacks into `rop` the element with coefficients packed into fields of size `bit_size` as represented by the integer `f`.

§37. fq_vec

Vectors over finite fields of arbitrary characteristic

37.1 Memory management

```
fq_struct * _fq_vec_init(slong len, const fq_ctx_t ctx)
```

Returns an initialised vector of `fq`'s of given length.

```
void _fq_vec_clear(fq * vec, slong len, const fq_ctx_t ctx)
```

Clears the entries of `(vec, len)` and frees the space allocated for `vec`.

37.2 Randomisation

```
void _fq_vec_randtest(fq_struct * f, flint_rand_t state,
   slong len, const fq_ctx_t ctx)
```

Sets the entries of a vector of the given length to elements of the finite field.

37.3 Input and output

```
int _fq_vec_fprint(FILE * file, const fq_struct * vec,
   slong len, const fq_ctx_t ctx)
```

Prints the vector of given length to the stream `file`. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_vec_print(const fq_struct * vec, slong len, const
   fq_ctx_t ctx)
```

Prints the vector of given length to `stdout`.

For further details, see `_fq_vec_fprint()`.

37.4 Assignment and basic manipulation

```
void _fq_vec_set(fq_struct * vec1, const fq_struct * vec2,
   slong len2, const fq_ctx_t ctx)
```

Makes a copy of `(vec2, len2)` into `vec1`.

```
void _fq_vec_swap(fq_struct * vec1, fq_struct * vec2, slong
   len2, const fq_ctx_t ctx)
```

Swaps the elements in `(vec1, len2)` and `(vec2, len2)`.

```
void _fq_vec_zero(fq_struct * vec, slong len, const
   fq_ctx_t ctx)
```

Zeros the entries of `(vec, len)`.

```
void _fq_vec_neg(fq_struct * vec1, const fq_struct * vec2,
   slong len2, const fq_ctx_t ctx)
```

Negates `(vec2, len2)` and places it into `vec1`.

37.5 Comparison

```
int _fq_vec_equal(const fq_struct * vec1, const fq_struct *
   vec2, slong len, const fq_ctx_t ctx)
```

Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

```
int _fq_vec_is_zero(const fq_struct * vec, slong len, const
   ctx_ctx)
```

Returns 1 if `(vec, len)` is zero, and 0 otherwise.

37.6 Addition and subtraction

```
void _fq_vec_add(fq_struct * res, const fq_struct * vec1,
   const fq_struct * vec2, slong len2, const fq_ctx_t ctx)
```

Sets `(res, len2)` to the sum of `(vec1, len2)` and `(vec2, len2)`.

```
void _fq_vec_sub(fq_struct * res, const fq_struct * vec1,
   const fq_struct * vec2, slong len2, const fq_ctx_t ctx)
```

Sets `(res, len2)` to `(vec1, len2)` minus `(vec2, len2)`.

37.7 Scalar multiplication and division

```
void _fq_vec_scalar_addmul_fq(fq_struct * vec1, const
   fq_struct * vec2, slong len2, const fq_t c, const
   fq_ctx_t ctx)
```

Adds `(vec2, len2)` times c to `(vec1, len2)`, where c is a `fq_t`.

```
void _fq_vec_scalar_submul_fq(fq_struct * vec1, const
   fq_struct * vec2, slong len2, const fq_t c, const
   fq_ctx_t ctx)
```

Subtracts `(vec2, len2)` times c from `(vec1, len2)`, where c is a `fq_t`.

37.8 Dot products

```
void _fq_vec_dot(fq_t res, const fq_struct * vec1, const
   fq_struct * vec2, slong len2, const fq_ctx_t ctx)
```

Sets `res` to the dot product of (`vec1`, `len`) and (`vec2`, `len`).

§38. fq_mat

Matrices over finite fields of arbitrary characteristic

38.1 Memory management

```
void fq_mat_init(fq_mat_t mat, slong rows, slong cols,
    const fq_ctx_t ctx)
```

Initialises `mat` to a `rows`-by-`cols` matrix with coefficients in $\mathbb{F}_q$ given by `ctx`. All elements are set to zero.

```
void fq_mat_init_set(fq_mat_t mat, fq_mat_t src, const
    fq_ctx_t ctx)
```

Initialises `mat` and sets its dimensions and elements to those of `src`.

```
void fq_mat_clear(fq_mat_t mat, const fq_ctx_t ctx)
```

Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an `fq_mat_t` object.

```
void fq_mat_set(fq_mat_t mat, fq_mat_t src, const fq_ctx_t
    ctx)
```

Sets `mat` to a copy of `src`. It is assumed that `mat` and `src` have identical dimensions.

38.2 Basic properties and manipulation

```
fq_struct * fq_mat_entry(fq_mat_t mat, slong i, slong j)
```

Directly accesses the entry in `mat` in row i and column j, indexed from zero. No bounds checking is performed.

```
fq_struct * fq_mat_entry_set(fq_mat_t mat, slong i, slong
    j, fq_t x, const fq_ctx_t ctx)
```

Sets the entry in `mat` in row i and column j to `x`.

```
slong fq_mat_nrows(fq_mat_t mat, const fq_ctx_t ctx)
```

Returns the number of rows in `mat`.

```
slong fq_mat_ncols(fq_mat_t mat, const fq_ctx_t ctx)
```

Returns the number of columns in `mat`.

```
void fq_mat_swap(fq_mat_t mat1, fq_mat_t mat2, const
    fq_ctx_t ctx)
```

Swaps two matrices. The dimensions of `mat1` and `mat2` are allowed to be different.

```
void fq_mat_zero(fq_mat_t mat, const fq_ctx_t ctx)
```

Sets all entries of `mat` to 0.

38.3 Printing

```
void fq_mat_print_pretty(const fq_mat_t mat, const fq_ctx_t
    ctx)
```

Pretty-prints `mat` to `stdout`. A header is printed followed by the rows enclosed in brackets.

```
int fq_mat_fprint_pretty(FILE * file, const fq_mat_t mat,
    const fq_ctx_t ctx)
```

Pretty-prints `mat` to `file`. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
void fq_mat_print(const fq_mat_t mat, const fq_ctx_t ctx)
```

Prints `mat` to `stdout`. A header is printed followed by the rows enclosed in brackets.

```
int fq_mat_fprint(FILE * file, const fq_mat_t mat, const
    fq_ctx_t ctx)
```

Prints `mat` to `file`. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

38.4 Window

```
void fq_mat_window_init(fq_mat_t window, const fq_mat_t
    mat, slong r1, slong c1, slong r2, slong c2, const
    fq_ctx_t ctx)
```

Initializes the matrix `window` to be an `r2 - r1` by `c2 - c1` submatrix of `mat` whose `(0,0)` entry is the `(r1, c1)` entry of `mat`. The memory for the elements of `window` is shared with `mat`.

```
void fq_mat_window_clear(fq_mat_t window, const fq_ctx_t
    ctx)
```

Clears the matrix `window` and releases any memory that it uses. Note that the memory to the underlying matrix that `window` points to is not freed.

38.5 Random matrix generation

```
void fq_mat_randtest(fq_mat_t mat, flint_rand_t state,
    const fq_ctx_t ctx)
```

Sets the elements of `mat` to random elements of $\mathbb{F}_q$, given by `ctx`.

```
int fq_mat_randpermdiag(fq_mat_t mat, fq_struct * diag,
    slong n, flint_rand_t state, const fq_ctx_t ctx)
```

Sets `mat` to a random permutation of the diagonal matrix with n leading entries given by the vector `diag`. It is assumed that the main diagonal of `mat` has room for at least n entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.

```
void fq_mat_randrank(fq_mat_t mat, slong rank, flint_rand_t
    state, const fq_ctx_t ctx)
```

Sets `mat` to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random elements of $\mathbb{F}_q$.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling `fq_mat_randops()`.

```
void fq_mat_randops(fq_mat_t mat, slong count, flint_rand_t
    state, const fq_ctx_t ctx)
```

Randomises `mat` by performing elementary row or column operations. More precisely, at most `count` random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

```
void fq_mat_randtril(fq_mat_t mat, flint_rand_t state, int
    unit, const fq_ctx_t ctx)
```

Sets `mat` to a random lower triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

```
void fq_mat_randtriu(fq_mat_t mat, flint_rand_t state, int
    unit, const fq_ctx_t ctx)
```

Sets `mat` to a random upper triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

38.6 Comparison

```
int fq_mat_equal(fq_mat_t mat1, fq_mat_t mat2, const
    fq_ctx_t ctx)
```

Returns nonzero if mat1 and mat2 have the same dimensions and elements, and zero otherwise.

```
int fq_mat_is_zero(const fq_mat_t mat, const fq_ctx_t ctx)
```

Returns a non-zero value if all entries `mat` are zero, and otherwise returns zero.

```
int fq_mat_is_empty(const fq_mat_t mat, const fq_ctx_t ctx)
```

Returns a non-zero value if the number of rows or the number of columns in `mat` is zero, and otherwise returns zero.

```
int fq_mat_is_square(const fq_mat_t mat, const fq_ctx_t ctx)
```

Returns a non-zero value if the number of rows is equal to the number of columns in `mat`, and otherwise returns zero.

38.7 Addition and subtraction

```
void fq_mat_add(fq_mat_t C, const fq_mat_t A, const
   fq_mat_t B, const fq_ctx_t ctx)
```

Computes $C = A + B$. Dimensions must be identical.

```
void fq_mat_sub(fq_mat_t C, const fq_mat_t A, const
   fq_mat_t B, const fq_ctx_t ctx)
```

Computes $C = A - B$. Dimensions must be identical.

```
void fq_mat_neg(fq_mat_t A, const fq_mat_t B, const
   fq_ctx_t ctx)
```

Sets $B = -A$. Dimensions must be identical.

38.8 Matrix multiplication

```
void fq_mat_mul(fq_mat_t C, const fq_mat_t A, const
   fq_mat_t B, const fq_ctx_t ctx)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. This function automatically chooses between classical and KS multiplication.

```
void fq_mat_mul_classical(fq_mat_t C, const fq_mat_t A,
   const fq_mat_t B, const fq_ctx_t ctx)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Uses classical matrix multiplication.

```
void fq_mat_mul_KS(fq_mat_t C, const fq_mat_t A, const
   fq_mat_t B, const fq_ctx_t ctx)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Uses Kronecker substitution to perform the multiplication over the integers.

```
void fq_mat_submul(fq_mat_t D, const fq_mat_t C, const
   fq_mat_t A, const fq_mat_t B, const fq_ctx_t ctx)
```

Sets $D = C + AB$. C and D may be aliased with each other but not with A or B.

38.9 LU decomposition

```
slong fq_mat_lu(slong * P, fq_mat_t A, int rank_check,
   const fq_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A.

If A is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r, U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and return 0 if A is detected to be rank-deficient.

This function calls `fq_mat_lu_recursive`.

```
slong fq_mat_lu_classical(slong * P, fq_mat_t A, int
   rank_check, const fq_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A. The behavior of this function is identical to that of `fq_mat_lu`. Uses Gaussian elimination.

```
slong fq_mat_lu_recursive(slong * P, fq_mat_t A, int
   rank_check, const fq_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A. The behavior of this function is identical to that of `fq_mat_lu`. Uses recursive block decomposition, switching to classical Gaussian elimination for sufficiently small blocks.

38.10 Reduced row echelon form

```
slong fq_mat_rref(fq_mat_t A, const fq_ctx_t ctx)
```

Puts A in reduced row echelon form and returns the rank of A.

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

38.11 Triangular solving

```
void fq_mat_solve_tril(fq_mat_t X, const fq_mat_t L, const
   fq_mat_t B, int unit, const fq_ctx_t ctx)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

```
void fq_mat_solve_tril_classical(fq_mat_t X, const fq_mat_t
   L, const fq_mat_t B, int unit, const fq_ctx_t ctx)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

```
void fq_mat_solve_tril_recursive(fq_mat_t X, const fq_mat_t
   L, const fq_mat_t B, int unit, const fq_ctx_t ctx)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$\begin{pmatrix} A & 0 \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}X \\ D^{-1}(Y - CA^{-1}X) \end{pmatrix}$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

```
void fq_mat_solve_triu(fq_mat_t X, const fq_mat_t U, const
   fq_mat_t B, int unit, const fq_ctx_t ctx)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

```
void fq_mat_solve_triu_classical(fq_mat_t X, const fq_mat_t
   U, const fq_mat_t B, int unit, const fq_ctx_t ctx)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

```
void fq_mat_solve_triu_recursive(fq_mat_t X, const fq_mat_t
   U, const fq_mat_t B, int unit, const fq_ctx_t ctx)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$\begin{pmatrix} A & B \\ 0 & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}(X - BD^{-1}Y) \\ D^{-1}Y \end{pmatrix}$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

§39. fq_poly

polynomials over finite fields of arbitrary characteristic

We represent a polynomial in $\mathbf{F}_q[X]$ as a `struct` which includes an array `coeffs` with the coefficients, as well as the length `length` and the number `alloc` of coefficients for which memory has been allocated.

As a data structure, we call this polynomial *normalised* if the top coefficient is non-zero.

Unless otherwise stated here, all functions that deal with polynomials assume that the $\mathbf{F}_q$ context of said polynomials are compatible, i.e., it assumes that the fields are generated by the same polynomial.

39.1 Memory management

```
void fq_poly_init(fq_poly_t poly, const fq_ctx_t ctx)
```

Initialises `poly` for use, with context ctx, and setting its length to zero. A corresponding call to `fq_poly_clear()` must be made after finishing with the `fq_poly_t` to free the memory used by the polynomial.

```
void fq_poly_init2(fq_poly_t poly, slong alloc, const
    fq_ctx_t ctx)
```

Initialises `poly` with space for at least `alloc` coefficients and sets the length to zero. The allocated coefficients are all set to zero. A corresponding call to `fq_poly_clear()` must be made after finishing with the `fq_poly_t` to free the memory used by the polynomial.

```
void fq_poly_realloc(fq_poly_t poly, slong alloc, const
    fq_ctx_t ctx)
```

Reallocates the given polynomial to have space for `alloc` coefficients. If `alloc` is zero the polynomial is cleared and then reinitialised. If the current length is greater than `alloc` the polynomial is first truncated to length `alloc`.

```
void fq_poly_fit_length(fq_poly_t poly, slong len, const
    fq_ctx_t ctx)
```

If `len` is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least `len` coefficients. No data is lost when calling this function.

The function efficiently deals with the case where `fit_length` is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

```
void _fq_poly_set_length(fq_poly_t poly, slong newlen,
    const fq_ctx_t ctx)
```

Sets the coefficients of `poly` beyond `len` to zero and sets the length of `poly` to `len`.

```
void fq_poly_clear(fq_poly_t poly, const fq_ctx_t ctx)
```

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

```
void _fq_poly_normalise(fq_poly_t poly, const fq_ctx_t ctx)
```

Sets the length of `poly` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

```
void _fq_poly_normalise2(fq_struct *poly, slong *length,
    const fq_ctx_t ctx)
```

Sets the length `length` of `(poly,length)` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

```
void fq_poly_truncate(fq_poly_t poly, slong newlen, const
    fq_ctx_t ctx)
```

Truncates the polynomial to length at most n.

```
void _fq_poly_reverse(fq_struct* output, const fq_struct*
    input, slong len, slong m, const fq_ctx_t ctx)
```

Sets `output` to the reverse of `input`, which is of length `len`, but thinking of it as a polynomial of length `m`, notionally zero-padded if necessary. The length `m` must be non-negative, but there are no other restrictions. The polynomial `output` must have space for `m` coefficients.

```
void fq_poly_reverse(fq_poly_t output, const fq_poly_t
    input, slong m, const fq_ctx_t ctx)
```

Sets `output` to the reverse of `input`, thinking of it as a polynomial of length `m`, notionally zero-padded if necessary). The length `m` must be non-negative, but there are no other restrictions. The output polynomial will be set to length `m` and then normalised.

39.2 Polynomial parameters

```
long fq_poly_degree(fq_poly_t poly, const fq_ctx_t ctx)
```

Returns the degree of the polynomial `poly`.

```
long fq_poly_length(fq_poly_t poly, const fq_ctx_t ctx)
```

Returns the length of the polynomial `poly`.

```
fq_struct * fq_poly_lead(const fq_poly_t poly, const
    fq_ctx_t ctx)
```

Returns a pointer to the leading coefficient of `poly`, or `NULL` if `poly` is the zero polynomial.

39.3 Randomisation

```
void fq_poly_randtest(fq_poly_t f, flint_rand_t state,
   slong len, const fq_ctx_t ctx)
```

Sets f to a random polynomial of length at most `len` with entries in the field described by `ctx`.

```
void fq_poly_randtest_not_zero(fq_poly_t f, flint_rand_t
   state, slong len, const fq_ctx_t ctx)
```

Same as `fq_poly_randtest` but guarantees that the polynomial is not zero.

```
void fq_poly_randtest_monic(fq_poly_t f, flint_rand_t
   state, slong len, const fq_ctx_t ctx)
```

Sets f to a random monic polynomial of length `len` with entries in the field described by `ctx`.

```
void fq_poly_randtest_irreducible(fq_poly_t f, flint_rand_t
   state, slong len, const fq_ctx_t ctx)
```

Sets f to a random monic, irreducible polynomial of length `len` with entries in the field described by `ctx`.

39.4 Assignment and basic manipulation

```
void _fq_poly_set(fq_struct *rop, const fq_struct *op,
   slong len, const fq_ctx_t ctx)
```

Sets `(rop, len)` to `(op, len)`.

```
void fq_poly_set(fq_poly_t poly1, const fq_poly_t poly2,
   const fq_ctx_t ctx)
```

Sets the polynomial `poly1` to the polynomial `poly2`.

```
void fq_poly_set_fq(fq_poly_t poly, const fq_t c, const
   fq_ctx_t ctx)
```

Sets the polynomial `poly` to `c`.

```
void fq_poly_swap(fq_poly_t op1, fq_poly_t op2, const
   fq_ctx_t ctx)
```

Swaps the two polynomials `op1` and `op2`.

```
void _fq_poly_zero(fq_struct *rop, slong len, const
   fq_ctx_t ctx)
```

Sets `(rop, len)` to the zero polynomial.

```
void fq_poly_zero(fq_poly_t poly, const fq_ctx_t ctx)
```

Sets `poly` to the zero polynomial.

```
void void fq_poly_one(fq_poly_t poly, const fq_ctx_t ctx)
```

Sets `poly` to the constant polynomial 1.

```
void void fq_poly_gen(fq_poly_t poly, const fq_ctx_t ctx)
```

Sets `poly` to the polynomial x.

```
void fq_poly_make_monic(fq_poly_t rop, const fq_poly_t op,
    const fq_ctx_t ctx)
```

Sets `rop` to `op`, normed to have leading coefficient 1.

```
void _fq_poly_make_monic(fq_struct *rop, const fq_struct
    *op, slong length, const fq_ctx_t ctx)
```

Sets `rop` to `(op,length)`, normed to have leading coefficient 1. Assumes that `rop` has enough space for the polynomial, assumes that `op` is not zero (and thus has an invertible leading coefficient).

39.5 Getting and setting coefficients

```
void fq_poly_get_coeff(fq_t x, const fq_poly_t poly, slong
    n, const fq_ctx_t ctx)
```

Sets x to the coefficient of X^n in `poly`.

```
void fq_poly_set_coeff(fq_poly_t poly, slong n, const fq_t
    x, const fq_ctx_t ctx)
```

Sets the coefficient of X^n in `poly` to x.

```
void fq_poly_set_coeff_fmpz(fq_poly_t poly, slong n, const
    fmpz_t x, const fq_ctx_t ctx)
```

Sets the coefficient of X^n in the polynomial to x, assuming $n \geq 0$.

39.6 Comparison

```
int fq_poly_equal(const fq_poly_t poly1, const fq_poly_t
    poly2, const fq_ctx_t ctx)
```

Returns whether the two polynomials `poly1` and `poly2` are equal.

```
int fq_poly_is_zero(const fq_poly_t poly, const fq_ctx_t
    ctx)
```

Returns whether the polynomial `poly` is the zero polynomial.

```
int fq_poly_is_one(const fq_poly_t op)
```

Returns whether the polynomial `poly` is equal to the constant polynomial 1.

```
int fq_poly_is_gen(const fq_poly_t op, const fq_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal to the polynomial x.

```
int fq_poly_is_unit(const fq_poly_t op, const fq_ctx_t ctx)
```

Returns whether the polynomial `poly` is a unit in the polynomial ring $\mathbf{F}_q[X]$, i.e. if it has degree 0 and is non-zero.

```
int fq_poly_equal_fq(const fq_poly_t poly, const fq_t c,
    const fq_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal the (constant) $\mathbf{F}_q$ element `c`

39.7 Addition and subtraction

```
void _fq_poly_add(fq_struct *res, const fq_struct *poly1,
    slong len1, const fq_struct *poly2, slong len2, const
    fq_ctx_t ctx)
```

Sets `res` to the sum of `(poly1,len1)` and `(poly2,len2)`.

```
void fq_poly_add(fq_poly_t res, const fq_poly_t poly1,
    const fq_poly_t poly2, const fq_ctx_t ctx)
```

Sets `res` to the sum of `poly1` and `poly2`.

```
void _fq_poly_sub(fq_struct *res, const fq_struct *poly1,
    slong len1, const fq_struct *poly2, slong len2, const
    fq_ctx_t ctx)
```

Sets `res` to the difference of `(poly1,len1)` and `(poly2,len2)`.

```
void fq_poly_sub(fq_poly_t res, const fq_poly_t poly1,
    const fq_poly_t poly2, const fq_ctx_t ctx)
```

Sets `res` to the difference of `poly1` and `poly2`.

```
void _fq_poly_neg(fq_struct *rop, const fq_struct *op,
    slong len, const fq_ctx_t ctx)
```

Sets `res` to the additive inverse of `(poly,len)`.

```
void fq_poly_neg(fq_poly_t res, const fq_poly_t poly, const
    fq_ctx_t ctx)
```

Sets `res` to the additive inverse of `poly`.

39.8 Scalar multiplication and division

```
void _fq_poly_scalar_mul_fq(fq_struct *rop, const fq_struct
    *op, slong len, const fq_t x, const fq_ctx_t ctx)
```

Sets `(rop,len)` to the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`.

```
void fq_poly_scalar_mul_fq(fq_poly_t rop, const fq_poly_t
    op, const fq_t x, const fq_ctx_t ctx)
```

Sets `(rop,len)` to the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`.

```
void _fq_poly_scalar_addmul_fq(fq_struct *rop, const
    fq_struct *op, slong len, const fq_t x, const fq_ctx_t
    ctx)
```

Adds to `(rop,len)` the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`. In particular, assumes the same length for `op` and `rop`.

```
void fq_poly_scalar_addmul_fq(fq_poly_t rop, const
    fq_poly_t op, const fq_t x, const fq_ctx_t ctx)
```

Adds to `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.

```
void _fq_poly_scalar_submul_fq(fq_struct *rop, const
   fq_struct *op, slong len, const fq_t x, const fq_ctx_t
   ctx)
```

Substracts from `(rop,len)` the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`. In particular, assumes the same length for `op` and `rop`.

```
void fq_poly_scalar_submul_fq(fq_poly_t rop, const
   fq_poly_t op, const fq_t x, const fq_ctx_t ctx)
```

Substracts from `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.

39.9 Multiplication

```
void _fq_poly_mul_classical(fq_struct *rop, const fq_struct
   *op1, slong len1, const fq_struct *op2, slong len2,
   const fq_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, assuming that `len1` is at least `len2` and neither is zero.

Permits zero padding. Does not support aliasing of `rop` with either `op1` or `op2`.

```
void fq_poly_mul_classical(fq_poly_t rop, const fq_poly_t
   op1, const fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2` using classical polynomial multiplication.

```
void _fq_poly_mul_reorder(fq_struct *rop, const fq_struct
   *op1, slong len1, const fq_struct *op2, slong len2,
   const fq_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, assuming that `len1` and `len2` are non-zero.

Permits zero padding. Supports aliasing.

```
void fq_poly_mul_reorder(fq_poly_t rop, const fq_poly_t
   op1, const fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`, reordering the two indeterminates X and Y when viewing the polynomials as elements of $\mathbf{F}_p[X, Y]$.

Suppose $\mathbf{F}_q = \mathbf{F}_p[X]/(f(X))$ and recall that elements of $\mathbf{F}_q$ are internally represented by elements of type `fmpz_poly`. For small degree extensions but polynomials in $\mathbf{F}_q[Y]$ of large degree n, we change the representation to

$$\begin{aligned} g(Y) &= \sum_{i=0}^{n} a_i(X) Y^i \\ &= \sum_{j=0}^{d} \sum_{i=0}^{n} \operatorname{Coeff}(a_i(X), j) Y^i. \end{aligned}$$

This allows us to use a poor algorithm (such as classical multiplication) in the X-direction and leverage the existing fast integer multiplication routines in the Y-direction where the polynomial degree n is large.

```
void _fq_poly_mul_KS(fq_struct *rop, const fq_struct *op1,
   slong len1, const fq_struct *op2, slong len2, const
   fq_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`.

Permits zero padding and places no assumptions on the lengths `len1` and `len2`. Supports aliasing.

```
void fq_poly_mul_KS(fq_poly_t rop, const fq_poly_t op1,
   const fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2` using Kronecker substitution, that is, by encoding each coefficient in $\mathbf{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

```
void _fq_poly_mul(fq_struct *rop, const fq_struct *op1,
   slong len1, const fq_struct *op2, slong len2, const
   fq_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

```
void fq_poly_mul(fq_poly_t rop, const fq_poly_t op1, const
   fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`, choosing an appropriate algorithm.

```
void _fq_poly_mullow_classical(fq_struct *rop, const
   fq_struct *op1, slong len1, const fq_struct *op2, slong
   len2, slong n, const fq_ctx_t ctx)
```

Sets `(res, n)` to the first n coefficients of `(poly1, len1)` multiplied by `(poly2, len2)`.

Assumes `0 < n <= len1 + len2 - 1`. Assumes neither `len1` nor `len2` is zero.

```
void fq_poly_mullow_classical(fq_poly_t rop, const
   fq_poly_t op1, const fq_poly_t op2, slong n, const
   fq_ctx_t ctx)
```

Sets `res` to the product of `poly1` and `poly2`, computed using the classical or schoolbook method.

```
void _fq_poly_mullow_KS(fq_struct *rop, const fq_struct
   *op1, slong len1, const fq_struct *op2, slong len2,
   slong n, const fq_ctx_t ctx)
```

Sets `(res, n)` to the lowest n coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `len1` and `len2` are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between `res`, `poly1` and `poly2`.

```
void fq_poly_mullow_KS(fq_poly_t rop, const fq_poly_t op1,
   const fq_poly_t op2, slong n, const fq_ctx_t ctx)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _fq_poly_mullow(fq_struct *rop, const fq_struct *op1,
   slong len1, const fq_struct *op2, slong len2, slong n,
   const fq_ctx_t ctx)
```

Sets `(res, n)` to the lowest n coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes `0 < n <= len1 + len2 - 1`. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

```
void fq_poly_mullow(fq_poly_t rop, const fq_poly_t op1,
   const fq_poly_t op2, slong n, const fq_ctx_t ctx)
```

Sets `res` to the lowest n coefficients of the product of `poly1` and `poly2`.

```
void _fq_poly_mulmod(fq_struct* res, const fq_struct*
   poly1, slong len1, const fq_struct* poly2, slong len2,
   const fq_struct* f, slong lenf, const fq_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_fq_poly_mul` instead.

Aliasing of `f` and `res` is not permitted.

```
void fq_poly_mulmod(fq_poly_t res,const fq_poly_t poly1,
   const fq_poly_t poly2, const fq_poly_t f, const fq_ctx_t
   ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

```
void _fq_poly_mulmod_preinv(fq_struct* res, const
   fq_struct* poly1, slong len1, const fq_struct* poly2,
   slong len2, const fq_struct* f, slong lenf, const
   fq_struct* finv, slong lenfinv, const fq_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `finv` is the inverse of the reverse of `f` mod `x^lenf`. It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_fq_poly_mul` instead.

Aliasing of `f` or `finv` and `res` is not permitted.

```
void fq_poly_mulmod_preinv(fq_poly_t res, const fq_poly_t
   poly1, const fq_poly_t poly2, const fq_poly_t f, const
   fq_poly_t finv, const fq_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`. `finv` is the inverse of the reverse of `f`.

39.10 Squaring

```
void _fq_poly_sqr_classical(fq_struct *rop, const fq_struct
   *op, slong len, const fq_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `(op,len)` is not zero and using classical polynomial multiplication.

Permits zero padding. Does not support aliasing of `rop` with either `op1` or `op2`.

```
void fq_poly_sqr_classical(fq_poly_t rop, const fq_poly_t
   op, const fq_ctx_t ctx)
```

Sets `rop` to the square of `op` using classical polynomial multiplication.

```
void _fq_poly_sqr_reorder(fq_struct *rop, const fq_struct
   *op, slong len, const fq_ctx_t ctx)
```

Sets `(rop, 2*len- 1)` to the square of `(op, len)`, assuming that `len` is not zero reordering the two indeterminates X and Y when viewing the polynomials as elements of $\mathbb{F}_p[X, Y]$.

Permits zero padding. Supports aliasing.

```
void fq_poly_sqr_reorder(fq_poly_t rop, const fq_poly_t op,
   const fq_ctx_t ctx)
```

Sets `rop` to the square of `op`, assuming that `len` is not zero reordering the two indeterminates X and Y when viewing the polynomials as elements of $\mathbb{F}_p[X, Y]$. See `fq_poly_mul_reorder`.

```
void _fq_poly_sqr_KS(fq_struct *rop, const fq_struct *op,
   slong len, const fq_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`.

Permits zero padding and places no assumptions on the lengths `len1` and `len2`. Supports aliasing.

```
void fq_poly_sqr_KS(fq_poly_t rop, const fq_poly_t op,
   const fq_ctx_t ctx)
```

Sets `rop` to the square `op` using Kronecker substitution, that is, by encoding each coefficient in $\mathbf{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

```
void _fq_poly_sqr(fq_struct *rop, const fq_struct *op,
   slong len, const fq_ctx_t ctx)
```

Sets `(rop, 2* len - 1)` to the square of `(op, len)`, choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

```
void fq_poly_sqr(fq_poly_t rop, const fq_poly_t op, const
   fq_ctx_t ctx)
```

Sets `rop` to the square of `op`, choosing an appropriate algorithm.

39.11 Powering

```
void _fq_poly_pow(fq_struct *rop, const fq_struct *op,
   slong len, ulong e, const fq_ctx_t ctx)
```

Sets `res = poly^e`, assuming that `e, len > 0` and that `res` has space for `e*(len - 1)+ 1` coefficients. Does not support aliasing.

```
void fq_poly_pow(fq_poly_t rop, const fq_poly_t op, ulong
   e, const fq_ctx_t ctx)
```

Computes `res = poly^e`. If e is zero, returns one, so that in particular `0^0 = 1`.

```
void _fq_poly_powmod_ui_binexp(fq_struct* res, const
   fq_struct* poly, ulong e, const fq_struct* f, slong
   lenf, const fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_poly_powmod_ui_binexp(fq_poly_t res, const
   fq_poly_t poly, ulong e, const fq_poly_t f, const
   fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void _fq_poly_powmod_ui_binexp_preinv(fq_struct* res, const
   fq_struct* poly, ulong e, const fq_struct* f, slong
   lenf, const fq_struct* finv, slong lenfinv, const
   fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_poly_powmod_ui_binexp_preinv(fq_poly_t res, const
   fq_poly_t poly, ulong e, const fq_poly_t f, const
   fq_poly_t finv, const fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fq_poly_powmod_fmpz_binexp(fq_struct* res, const
   fq_struct* poly, fmpz_t e, const fq_struct* f, slong
   lenf, const fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_poly_powmod_fmpz_binexp(fq_poly_t res, const
   fq_poly_t poly, fmpz_t e, const fq_poly_t f, const
   fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void _fq_poly_powmod_fmpz_binexp_preinv(fq_struct* res,
   const fq_struct* poly, fmpz_t e, const fq_struct* f,
   slong lenf, const fq_struct* finv, slong lenfinv, const
   fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_poly_powmod_fmpz_binexp_preinv(fq_poly_t res, const
   fq_poly_t poly, fmpz_t e, const fq_poly_t f, const
   fq_poly_t finv, const fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fq_poly_powmod_fmpz_sliding_preinv(fq_struct* res,
   const fq_struct* poly, fmpz_t e, ulong k, const
   fq_struct* f, slong lenf, const fq_struct* finv, slong
   lenfinv, const fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an "optimum" size will be selected automatically base on `e`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_poly_powmod_fmpz_sliding_preinv(fq_poly_t res,
   const fq_poly_t poly, fmpz_t e, ulong k, const fq_poly_t
   f, const fq_poly_t finv, const fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an "optimum" size will be selected automatically base on `e`.

```
void _fq_poly_powmod_x_fmpz_preinv(fq_struct * res, const
   fmpz_t e, const fq_struct * f, slong lenf, const
   fq_struct * finv, slong lenfinv, const fq_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 2`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_poly_powmod_x_fmpz_preinv(fq_poly_t res, const
   fmpz_t e, const fq_poly_t f, const fq_poly_t finv, const
   fq_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

39.12 Shifting

```
void _fq_poly_shift_left(fq_struct *rop, const fq_struct
   *op, slong len, slong n, const fq_ctx_t ctx)
```

Sets `(res, len + n)` to `(poly, len)` shifted left by n coefficients.

Inserts zero coefficients at the lower end. Assumes that `len` and n are positive, and that `res` fits `len + n` elements. Supports aliasing between `res` and `poly`.

```
void fq_poly_shift_left(fq_poly_t rop, const fq_poly_t op,
   slong n, const fq_ctx_t ctx)
```

Sets `res` to `poly` shifted left by n coeffs. Zero coefficients are inserted.

```
void _fq_poly_shift_right(fq_struct *rop, const fq_struct
   *op, slong len, slong n, const fq_ctx_t ctx)
```

Sets `(res, len - n)` to `(poly, len)` shifted right by n coefficients.

Assumes that `len` and n are positive, that `len > n`, and that `res` fits `len - n` elements. Supports aliasing between `res` and `poly`, although in this case the top coefficients of `poly` are not set to zero.

```
void fq_poly_shift_right(fq_poly_t rop, const fq_poly_t op,
   slong n, const fq_ctx_t ctx)
```

Sets `res` to `poly` shifted right by n coefficients. If n is equal to or greater than the current length of `poly`, `res` is set to the zero polynomial.

39.13 Norms

```
long _fq_poly_hamming_weight(const fq_poly *op, slong len,
   const fq_ctx_t ctx)
```

Returns the number of non-zero entries in `(op, len)`.

```
long fq_poly_hamming_weight(const fq_poly_t op, const
   fq_ctx_t ctx)
```

Returns the number of non-zero entries in the polynomial `op`.

39.14 Euclidean division

```
void _fq_poly_divrem_basecase(fq_struct *Q, fq_struct *R,
   const fq_struct *A, slong lenA, const fq_struct *B,
   slong lenB, const fq_t invB, const fq_ctx_t ctx)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ with $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is its inverse.

Assumes that $\operatorname{len}(A), \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fq_poly_divrem_basecase(fq_poly_t Q, fq_poly_t R,
   const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ with $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible. This can be taken for granted the context is for a finite field, that is, when p is prime and $f(X)$ is irreducible.

```
void _fq_poly_divrem(fq_struct *Q, fq_struct *R, const
   fq_struct *A, slong lenA, const fq_struct *B, slong
   lenB, const fq_t invB, const fq_ctx_t ctx)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ with $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is its inverse.

Assumes that $\operatorname{len}(A), \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fq_poly_divrem(fq_poly_t Q, fq_poly_t R, const
   fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible. This can be taken for granted the context is for a finite field, that is, when p is prime and $f(X)$ is irreducible.

```
void _fq_poly_rem(fq_struct *R, const fq_struct *A, slong
   lenA, const fq_struct *B, slong lenB, const fq_t invB,
   const fq_ctx_t ctx)
```

Sets `R` to the remainder of the division of `(A,lenA)` by `(B,lenB)`. Assumes that the leading coefficient of `(B,lenB)` is invertible and that `invB` is its inverse.

```
void fq_poly_rem(fq_poly_t R, const fq_poly_t A, const
   fq_poly_t B, const fq_ctx_t ctx)
```

Sets `R` to the remainder of the division of `A` by `B` in the context described by `ctx`.

```
void _fq_poly_div_basecase(fq_struct *Q, fq_struct *R,
   const fq_struct *A, slong lenA, const fq_struct *B,
   slong lenB, const fq_t invB, const fq_ctx_t ctx)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets `(Q, lenA - lenB + 1)`.

Requires temporary space `(R, lenA)`. If `R` is `NULL`, then the temporary space will be allocated. Allows aliasing only between A and R. Allows zero-padding in A but not in B. Assumes that the leading coefficient of B is a unit.

```
void fq_poly_div_basecase(fq_poly_t Q, const fq_poly_t A,
   const fq_poly_t B, const fq_ctx_t ctx)
```

Notionally finds polynomials Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only `Q`. If $\text{len}(B) = 0$ an exception is raised.

```
void _fq_poly_divrem_divconquer_recursive(fq_struct * Q,
   fq_struct * BQ, fq_struct * W, const fq_struct * A,
   const fq_struct * B, slong lenB, const fq_t invB, const
   fq_ctx_t ctx)
```

Computes `(Q, lenB)`, `(BQ, 2 lenB - 1)` such that $BQ = B \times Q$ and $A = BQ + R$ where $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is the inverse.

Assumes $\text{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. Requires a temporary array `(W, 2 lenB - 1)`. No aliasing of input and output operands is allowed.

This function does not read the bottom $\text{len}(B) - 1$ coefficients from A, which means that they might not even need to exist in allocated memory.

```
void _fq_poly_divrem_divconquer(fq_struct * Q, fq_struct *
   R, const fq_struct * A, slong lenA, const fq_struct * B,
   slong lenB, const fq_t invB, const fq_ctx_t ctx)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is the inverse.

Assumes $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. No aliasing of input and output operands is allowed.

```
void fq_poly_divrem_divconquer(fq_poly_t Q, fq_poly_t R,
   const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that B is non-zero and that the leading coefficient of B is invertible.

```
void _fq_poly_div_newton_n_preinv(fq_struct* Q, const
   fq_struct* A, slong lenA, const fq_struct* B, slong
   lenB, const fq_struct* Binv, slong lenBinv, const
   fq_struct ctx_t)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`, but return only Q.

We require that Q have space for `lenA - lenB + 1` coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void fq_poly_div_newton_n_preinv(fq_poly_t Q, const
   fq_poly_t A, const fq_poly_t B, const fq_poly_t Binv,
   const fq_ctx_t ctx)
```

Notionally computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q.

We assume that the leading coefficient of B is a unit and that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void _fq_poly_divrem_newton_n_preinv(fq_struct* Q,
   fq_struct* R, const fq_struct* A, slong lenA, const
   fq_struct* B, slong lenB, const fq_struct* Binv, slong
   lenBinv, const fq_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where A is of length `lenA` and B is of length `lenB`. We require that Q have space for `lenA - lenB + 1` coefficients. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$. The algorithm used is to call `div_newton_n_preinv()` and then multiply out and compute the remainder.

```
void fq_poly_divrem_newton_preinv(fq_poly_t Q, fq_poly_t R,
   const fq_poly_t A, const fq_poly_t B, const fq_poly_t
   Binv, const fq_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to call `div_newton_n()` and then multiply out and compute the remainder.

```
void _fq_poly_inv_series_newton(fq_struct* Qinv, const
   fq_struct* Q, slong n, const fq_ctx_t ctx)
```

Given `Q` of length `n` whose constant coefficient is invertible modulo the given modulus, find a polynomial `Qinv` of length `n` such that `Q * Qinv` is `1` modulo x^n. Requires `n > 0`. This function can be viewed as inverting a power series via Newton iteration.

```
void fq_poly_inv_series_newton(fq_poly_t Qinv, const
   fq_poly_t Q, slong n, const fq_ctx_t ctx)
```

Given `Q` find `Qinv` such that `Q * Qinv` is `1` modulo x^n. The constant coefficient of `Q` must be invertible modulo the modulus of `Q`. An exception is raised if this is not the case or if `n = 0`. This function can be viewed as inverting a power series via Newton iteration.

39.15 Greatest common divisor

```
void fq_poly_gcd(fq_poly_t rop, const fq_poly_t op1, const
   fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the greatest common divisor of `op1` and `op2`, using the either the Euclidean or HGCD algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

```
long _fq_poly_gcd(fq_struct* G,const fq_struct* A, slong
   lenA, const fq_struct* B, slong lenB, const fq_ctx_t ctx)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= lenB > 0` and sets G to it. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for `lenB` coefficients.

```
void fq_poly_gcd_euclidean(fq_poly_t rop, const fq_poly_t
   op1, const fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the greatest common divisor of `op1` and `op2`, using the Euclidean algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

```
long _fq_poly_gcd_euclidean(fq_struct* G, const fq_struct*
   A, slong lenA, const fq_struct* B, slong lenB, const
   fq_ctx_t ctx)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= lenB > 0` and sets G to it. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for `lenB` coefficients.

39.16 Divisibility testing

```
int _fq_poly_divides(fq_struct *Q, const fq_struct *A,
   slong lenA, const fq_struct *B, slong lenB, const fq_t
   invB, const fq_ctx_t ctx)
```

Returns 1 if `(B, lenB)` divides `(A, lenA)` exactly and sets Q to the quotient, otherwise returns 0.

It is assumed that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$ and that Q has space for $\operatorname{len}(A) - \operatorname{len}(B) + 1$ coefficients.

Aliasing of Q with either of the inputs is not permitted.

This function is currently unoptimised and provided for convenience only.

```
int fq_poly_divides(fq_poly_t Q, const fq_poly_t A, const
    fq_poly_t B, const fq_ctx_t ctx)
```

Returns 1 if B divides A exactly and sets Q to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.

39.17 Derivative

```
void _fq_poly_derivative(fq_struct *rop, const fq_struct
    *op, slong len, const fq_ctx_t ctx)
```

Sets `(rpoly, len - 1)` to the derivative of `(poly, len)`. Also handles the cases where `len` is 0 or 1 correctly. Supports aliasing of `rpoly` and `poly`.

```
void fq_poly_derivative(fq_poly_t rop, const fq_poly_t op,
    const fq_ctx_t ctx)
```

Sets `res` to the derivative of `poly`.

39.18 Evaluation

```
void _fq_poly_evaluate_fq(fq_t rop, const fq_struct *op,
    slong len, const fq_t a, const fq_ctx_t ctx)
```

Sets `rop` to `(op, len)` evaluated at a.

Supports zero padding. There are no restrictions on `len`, that is, `len` is allowed to be zero, too.

```
void fq_poly_evaluate_fq(fq_t rop, const fq_poly_t f, const
    fq_t a, const fq_ctx_t ctx)
```

Sets `rop` to the value of $f(a)$.

As the coefficient ring $\mathbf{F}_q$ is finite, Horner's method is sufficient.

39.19 Composition

```
void _fq_poly_compose_divconquer(fq_struct *rop, const
    fq_struct *op1, slong len1, const fq_struct *op2, slong
    len2, const fq_ctx_t ctx)
```

Computes the composition of `(op1, len1)` and `(op2, len2)` using a divide and conquer approach and places the result into `rop`, assuming `rop` can hold the output of length `(len1 - 1)* (len2 - 1)+ 1`.

Assumes `len1, len2 > 0`. Does not support aliasing between `rop` and any of `(op1, len1)` and `(op2, len2)`.

```
void fq_poly_compose_divconquer(fq_poly_t rop, const
    fq_poly_t op1, const fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be precise about the order of composition, denoting `rop`, `op1`, and `op2` by f, g, and h, respectively, sets $f(t) = g(h(t))$.

```
void _fq_poly_compose_horner(fq_struct *rop, const
    fq_struct *op1, slong len1, const fq_struct *op2, slong
    len2, const fq_ctx_t ctx)
```

Sets `rop` to the composition of `(op1, len1)` and `(op2, len2)`.

Assumes that `rop` has space for `(len1-1)*(len2-1)+ 1` coefficients. Assumes that `op1` and `op2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fq_poly_compose_horner(fq_poly_t rop, const fq_poly_t
   op1, const fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be more precise, denoting `rop`, `op1`, and `op2` by f, g, and h, sets $f(t) = g(h(t))$.

This implementation uses Horner's method.

```
void _fq_poly_compose(fq_struct *rop, const fq_struct *op1,
   slong len1, const fq_struct *op2, slong len2, const
   fq_ctx_t ctx)
```

Sets `rop` to the composition of `(op1, len1)` and `(op2, len2)`.

Assumes that `rop` has space for `(len1-1)*(len2-1)+ 1` coefficients. Assumes that `op1` and `op2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fq_poly_compose(fq_poly_t rop, const fq_poly_t op1,
   const fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be precise about the order of composition, denoting `rop`, `op1`, and `op2` by f, g, and h, respectively, sets $f(t) = g(h(t))$.

```
void _fq_poly_compose_mod_horner(fq_struct * res, const
   fq_struct * f, slong lenf, const fq_struct * g, const
   fq_struct * h, slong lenh, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_poly_compose_mod_horner(fq_poly_t res, const
   fq_poly_t f, const fq_poly_t g, const fq_poly_t h, const
   fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero. The algorithm used is Horner's rule.

```
void _fq_poly_compose_mod_horner_preinv(fq_struct * res,
   const fq_struct * f, slong lenf, const fq_struct * g,
   const fq_struct * h, slong lenh, const fq_struct * hinv,
   slong lenhiv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_poly_compose_mod_horner_preinv(fq_poly_t res, const
   fq_poly_t f, const fq_poly_t g, const fq_poly_t h, const
   fq_poly_t hinv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The algorithm used is Horner's rule.

```
void _fq_poly_compose_mod_brent_kung(fq_struct * res, const
   fq_struct * f, slong lenf, const fq_struct * g, const
   fq_struct * h, slong lenh, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_poly_compose_mod_brent_kung(fq_poly_t res, const
   fq_poly_t f, const fq_poly_t g, const fq_poly_t h, const
   fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. The algorithm used is the Brent-Kung matrix algorithm.

```
void _fq_poly_compose_mod_brent_kung_preinv(fq_struct *
   res, const fq_struct * f, slong lenf, const fq_struct *
   g, const fq_struct * h, slong lenh, const fq_struct *
   hinv, slong lenhiv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_poly_compose_mod_brent_kung_preinv(fq_poly_t res,
   const fq_poly_t f, const fq_poly_t g, const fq_poly_t h,
   const fq_poly_t hinv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The algorithm used is the Brent-Kung matrix algorithm.

```
void _fq_poly_compose_mod(fq_struct * res, const fq_struct
   * f, slong lenf, const fq_struct * g, const fq_struct *
   h, slong lenh, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```
void fq_poly_compose_mod(fq_poly_t res, const fq_poly_t f,
   const fq_poly_t g, const fq_poly_t h, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero.

```
void _fq_poly_compose_mod_preinv(fq_struct * res, const
   fq_struct * f, slong lenf, const fq_struct * g, const
   fq_struct * h, slong lenh, const fq_struct * hinv, slong
   lenhiv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

```
void fq_poly_compose_mod_preinv(fq_poly_t res, const
   fq_poly_t f, const fq_poly_t g, const fq_poly_t h, const
   fq_poly_t hinv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`.

```
void _fq_poly_reduce_matrix_mod_poly (fq_mat_t A, const
   fq_mat_t B, const fq_poly_t f, const fq_ctx_t ctx)
```

Sets the ith row of `A` to the reduction of the ith row of B modulo f for $i = 1, \ldots, \sqrt{\deg(f)}$. We require B to be at least a $\sqrt{\deg(f)} \times \deg(f)$ matrix and f to be nonzero.

```
void _fq_poly_precompute_matrix (fq_mat_t A, const
   fq_struct* f, const fq_struct* g, slong leng, const
   fq_struct* ginv, slong lenginv, const fq_ctx_t ctx)
```

Sets the ith row of `A` to f^i modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of `g` and g to be nonzero.

```
void fq_poly_precompute_matrix (fq_mat_t A, const fq_poly_t
   f, const fq_poly_t g, const fq_poly_t ginv, const
   fq_ctx_t ctx)
```

Sets the ith row of `A` to f^i modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of `g`.

```
void
   _fq_poly_compose_mod_brent_kung_precomp_preinv(fq_struct*
   res, const fq_struct* f, slong lenf, const fq_mat_t A,
   const fq_struct* h, slong h, const fq_struct* hinv,
   slong lenhinv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero. We require that the ith row of A contains g^i for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void
   fq_poly_compose_mod_brent_kung_precomp_preinv(fq_poly_t
   res, const fq_poly_t f, const fq_mat_t A, const
   fq_poly_t h, const fq_poly_t hinv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that the ith row of A contains g^i for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. This version of Brent-Kung modular composition is

particularly useful if one has to perform several modular composition of the form $f(g)$ modulo h for fixed g and h.

39.20 Output

```
int _fq_poly_fprint_pretty(FILE *file, const fq_struct
   *poly, slong len, const char *x, const fq_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_poly_fprint_pretty(FILE * file, const fq_poly_t
   poly, const char *x, const fq_ctx_t ctx)
```

Prints the pretty representation of `poly` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_poly_print_pretty(const fq_struct *poly, slong len,
   const char *x, const fq_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_poly_print_pretty(const fq_poly_t poly, const char
   *x, const fq_ctx_t ctx)
```

Prints the pretty representation of `poly` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_poly_fprint(FILE *file, const fq_struct *poly,
   slong len, const fq_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_poly_fprint(FILE * file, const fq_poly_t poly, const
   fq_ctx_t ctx)
```

Prints the pretty representation of `poly` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_poly_print(const fq_struct *poly, slong len, const
   fq_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_poly_print(const fq_poly_t poly, const fq_ctx_t ctx)
```

Prints the representation of `poly` to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
char * _fq_poly_get_str(const fq_struct * poly, slong len,
   const fq_ctx_t ctx)
```

Returns the plain FLINT string representation of the polynomial `(poly, len)`.

```
char * fq_poly_get_str(const fq_poly_t poly, const fq_ctx_t
   ctx)
```

Returns the plain FLINT string representation of the polynomial `poly`.

```
char * _fq_poly_get_str_pretty(const fq_struct * poly,
   slong len, const char * x, const fq_ctx_t ctx)
```

Returns a pretty representation of the polynomial `(poly, len)` using the null-terminated string `x` as the variable name.

```
char * fq_poly_get_str_pretty(const fq_poly_t poly, const
   char * x, const fq_ctx_t ctx)
```

Returns a pretty representation of the polynomial `poly` using the null-terminated string `x` as the variable name

39.21 Inflation and deflation

```
void fq_poly_inflate(fq_poly_t result, const fq_poly_t
   input, ulong inflation, const fq_ctx_t ctx)
```

Sets `result` to the inflated polynomial $p(x^n)$ where p is given by `input` and n is given by `inflation`.

```
void fq_poly_deflate(fq_poly_t result, const fq_poly_t
   input, ulong deflation, const fq_ctx_t ctx)
```

Sets `result` to the deflated polynomial $p(x^{1/n})$ where p is given by `input` and n is given by `deflation`. Requires $n > 0$.

```
ulong fq_poly_deflation(const fq_poly_t input, const
   fq_ctx_t ctx)
```

Returns the largest integer by which `input` can be deflated. As special cases, returns 0 if `input` is the zero polynomial and 1 of `input` is a constant polynomial.

§40. fq_poly_factor

Factorisation of polynomials over
finite fields of arbitrary characteristic

The `fq_poly_factor` module is included automatically when one includes `fq_poly.h`. One should not try to include `fq_poly_factor.h` directly.

40.1 Memory Management

```
void fq_poly_factor_init(fq_poly_factor_t fac, const
   fq_ctx_t ctx)
```

Initialises `fac` for use. An `fq_poly_factor_t` represents a polynomial in factorised form as a product of polynomials with associated exponents.

```
void fq_poly_factor_clear(fq_poly_factor_t fac, const
   fq_ctx_t ctx)
```

Frees all memory associated with `fac`.

```
void fq_poly_factor_realloc(fq_poly_factor_t fac, slong
   alloc, const fq_ctx_t ctx)
```

Reallocates the factor structure to provide space for precisely `alloc` factors.

```
void fq_poly_factor_fit_length(fq_poly_factor_t fac, slong
   len, const fq_ctx_t ctx)
```

Ensures that the factor structure has space for at least `len` factors. This functions takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

40.2 Basic Operations

```
void fq_poly_factor_set(fq_poly_factor_t res, const
   fq_poly_factor_t fac, const fq_ctx_t ctx)
```

Sets `res` to the same factorisation as `fac`.

```
void fq_poly_factor_print_pretty(const fq_poly_factor_t
   fac, const fq_ctx_t ctx)
```

Pretty-prints the entries of `fac` to standard output.

```
void fq_poly_factor_print(const fq_poly_factor_t fac, const
   fq_ctx_t ctx)
```

Prints the entries of `fac` to standard output.

```
void fq_poly_factor_insert(fq_poly_factor_t fac, const
   fq_poly_t poly, slong exp, const fq_ctx_t ctx)
```

Inserts the factor `poly` with multiplicity `exp` into the factorisation `fac`.

If `fac` already contains `poly`, then `exp` simply gets added to the exponent of the existing entry.

```
void fq_poly_factor_concat(fq_poly_factor_t res, const
   fq_poly_factor_t fac, const fq_ctx_t ctx)
```

Concatenates two factorisations.

This is equivalent to calling `fq_poly_factor_insert()` repeatedly with the individual factors of `fac`.

Does not support aliasing between `res` and `fac`.

```
void fq_poly_factor_pow(fq_poly_factor_t fac, slong exp,
   const fq_ctx_t ctx)
```

Raises `fac` to the power `exp`.

```
ulong fq_poly_remove(fq_poly_t f, const fq_poly_t p, const
   fq_ctx_t ctx)
```

Removes the highest possible power of `p` from `f` and returns the exponent.

40.3 Irreducibility Testing

```
int fq_poly_is_irreducible(const fq_poly_t f, const
   fq_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0.

```
int fq_poly_is_irreducible_ddf(const fq_poly_t f, const
   fq_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

```
int fq_poly_is_irreducible_ben_or(const fq_poly_t f, const
   fq_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses Ben-Or's irreducibility test.

```
int _fq_poly_is_squarefree(const fq_struct * f, slong len,
   const fq_ctx_t ctx)
```

Returns 1 if `(f, len)` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

```
int fq_poly_is_squarefree(const fq_poly_t f, const fq_ctx_t
   ctx)
```

Returns 1 if `f` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

40.4 Factorisation

```
int fq_poly_factor_equal_deg_prob(fq_poly_t factor,
    flint_rand_t state, const fq_poly_t pol, slong d, const
    fq_ctx_t ctx)
```

Probabilistic equal degree factorisation of `pol` into irreducible factors of degree `d`. If it passes, a factor is placed in factor and 1 is returned, otherwise 0 is returned and the value of factor is undetermined.

Requires that `pol` be monic, non-constant and squarefree.

```
void fq_poly_factor_equal_deg(fq_poly_factor_t factors,
    const fq_poly_t pol, slong d, const fq_ctx_t ctx)
```

Assuming `pol` is a product of irreducible factors all of degree `d`, finds all those factors and places them in factors. Requires that `pol` be monic, non-constant and squarefree.

```
void fq_poly_factor_distinct_deg(fq_poly_factor_t res,
    const fq_poly_t poly, slong * const *degs, const
    fq_ctx_t ctx)
```

Factorises a monic non-constant squarefree polymnomial `poly` of degree n into factors $f[d]$ such that for $1 \leq d \leq n$ $f[d]$ is the product of the monic irreducible factors of `poly` of degree d. Factors are stored in `res`, assotiated powers of irreducible polynomials are stored in `degs` in the same order as factors.

Requires that `degs` have enough space for irreducible polynomials' powers (maximum space required is $n * sizeof(slong)$).

```
void fq_poly_factor_squarefree(fq_poly_factor_t res, const
    fq_poly_t f, const fq_ctx_t ctx)
```

Sets `res` to a squarefree factorization of `f`.

```
void fq_poly_factor(fq_poly_factor_t res, const fq_poly_t
    f, const fq_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors choosing the best algorithm for given modulo and degree. Choise is based on heuristic measurments.

```
void fq_poly_factor_cantor_zassenhaus(fq_poly_factor_t res,
    const fq_poly_t f, const fq_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the Cantor-Zassenhaus algorithm.

```
void fq_poly_factor_kaltofen_shoup(fq_poly_factor_t res,
    const fq_poly_t poly, const fq_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a âĂIJbaby step/giant stepâĂİ strategy for the distinct-degree factorization step.

```
void fq_poly_factor_berlekamp(fq_poly_factor_t factors,
    const fq_poly_t f, const fq_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the Berlekamp algorithm.

```
void fq_poly_factor_with_berlekamp(fq_poly_factor_t res,
   fq_t leading_coeff, const fq_poly_t f, const fq_ctx_t)
```

Factorises a general polynomial `f` into monic irreducible factors and sets `leading_coeff` to the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Berlekamp factorisation on all the individual square-free factors.

```
void fq_poly_factor_with_cantor_zassenhaus(fq_poly_factor_t
   res, fq_t leading_coeff const fq_poly_t f, const
   fq_ctx_t ctx)
```

Factorises a general polynomial `f` into monic irreducible factors and sets `leading_coeff` to the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

```
void fq_poly_factor_with_kaltofen_shoup(fq_poly_factor_t
   res, fq_t leading_coeff, const fq_poly_t f, const
   fq_ctx_t ctx)
```

Factorises a general polynomial `f` into monic irreducible factors and sets `leading_coeff` to the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

```
void fq_poly_iterated_frobenius_preinv(fq_poly_t *rop,
   slong n, const fq_poly_t v, const fq_poly_t vinv, const
   fq_ctx_t ctx)
```

Sets `rop[i]` to be $x^{q^i} \mod v$ for $0 <= i < n$.

It is required that `vinv` is the inverse of the reverse of `v` mod `x^lenv`.

§41. fq_nmod

Finite fields of word-sized characteristic

We represent an element of the finite field $\mathbb{F}_{p^n} \cong \mathbb{F}_p[X]/(f(X))$, where $f(X) \in \mathbb{F}_p[X]$ is a monic, irreducible polynomial of degree n, as a polynomial in $\mathbb{F}_p[X]$ of degree less than n. The underlying data structure is an `nmod_poly_t`.

The default choice for $f(X)$ is the Conway polynomial for the pair (p, n). Frank Luebeck's data base of Conway polynomials is made available in the file `qadic/CPimport.txt`. If a Conway polynomial is not available, then a random irreducible polynomial will be chosen for $f(X)$. Additionally, the user is able to supply their own $f(X)$.

41.1 Context Management

```
void fq_nmod_ctx_init(fq_nmod_ctx_t ctx, const fmpz_t p,
   slong d, const char *var)
```

Initialises the context for prime p and extension degree d, with name `var` for the generator. By default, it will try use a Conway polynomial; if one is not available, a random irreducible polynomial will be used.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

```
int _fq_nmod_ctx_init_conway(fq_nmod_ctx_t ctx, const
   fmpz_t p, slong d, const char *var)
```

Attempts to initialise the context for prime p and extension degree d, with name `var` for the generator using a Conway polynomial for the modulus.

Returns 1 if the Conway polynomial is in the database for the given size and the initialization is successful; otherwise, returns 0.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

```
void fq_nmod_ctx_init_conway(fq_nmod_ctx_t ctx, const
   fmpz_t p, slong d, const char *var)
```

Initialises the context for prime p and extension degree d, with name `var` for the generator using a Conway polynomial for the modulus.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

```
void fq_nmod_ctx_init_modulus(fq_nmod_ctx_t ctx,
   nmod_poly_t modulus, const char *var)
```

Initialises the context for given `modulus` with name `var` for the generator.

Assumes that `modulus` is and irreducible polynomial over $\mathbf{F}_p$.

Assumes that the string `var` is a null-terminated string of length at least one.

```
void fq_nmod_ctx_clear(fq_nmod_ctx_t ctx)
```

Clears all memory that has been allocated as part of the context.

```
long fq_nmod_ctx_degree(const fq_nmod_ctx_t ctx)
```

Returns the degree of the field extension $[\mathbf{F}_q : \mathbf{F}_p]$, which is equal to $\log_p q$.

```
fmpz * fq_nmod_ctx_prime(const fq_nmod_ctx_t ctx)
```

Returns a pointer to the prime p in the context.

```
void fq_nmod_ctx_order(fmpz_t f, const fq_nmod_ctx_t ctx)
```

Sets f to be the size of the finite field.

```
int fq_nmod_ctx_fprint(FILE * file, const fq_nmod_ctx_t ctx)
```

Prints the context information to `file`. Returns 1 for a success and a negative number for an error.

```
void fq_nmod_ctx_print(const fq_nmod_ctx_t ctx)
```

Prints the context information to `stdout`.

```
void fq_nmod_ctx_randtest(fq_nmod_ctx_t ctx)
```

Initializes `ctx` to a random finite field. Assumes that `fq_nmod_ctx_init` as not been called on `ctx` already.

```
void fq_nmod_ctx_randtest_reducible(fq_nmod_ctx_t ctx)
```

Initializes `ctx` to a random extension of a word-sized prime field. The modulus may or may not be irreducible. Assumes that `fq_nmod_ctx_init` as not been called on `ctx` already.

41.2 Memory management

```
void fq_nmod_init(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
```

Initialises the element `rop`, setting its value to 0.

```
void fq_nmod_init2(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
```

Initialises `poly` with at least enough space for it to be an element of `ctx` and sets it to 0.

```
void fq_nmod_clear(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
```

Clears the element `rop`.

```
void _fq_nmod_sparse_reduce(mp_ptr R, slong lenR, const
   fq_nmod_ctx_t ctx)
```

Reduces `(R, lenR)` modulo the polynomial f given by the modulus of `ctx`.

```
void _fq_nmod_dense_reduce(mp_ptr R, slong lenR, const
    fq_nmod_ctx_t ctx)
```

Reduces `(R, lenR)` modulo the polynomial f given by the modulus of `ctx` using Newton division.

```
void _fq_nmod_reduce(mp_ptr r, slong lenR, const
    fq_nmod_ctx_t ctx)
```

Reduces `(R, lenR)` modulo the polynomial f given by the modulus of `ctx`. Does either sparse or dense reduction based on `ctx->sparse_modulus`.

```
void fq_nmod_reduce(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
```

Reduces the polynomial `rop` as an element of $\mathbf{F}_p[X]/(f(X))$.

41.3 Basic arithmetic

```
void fq_nmod_add(fq_nmod_t rop, const fq_nmod_t op1, const
    fq_nmod_t op2, const fq_nmod_ctx_t ctx)
```

Sets `rop` to the sum of `op1` and `op2`.

```
void fq_nmod_sub(fq_nmod_t rop, const fq_nmod_t op1, const
    fq_nmod_t op2, const fq_nmod_ctx_t ctx)
```

Sets `rop` to the difference of `op1` and `op2`.

```
void fq_nmod_sub_one(fq_nmod_t rop, const fq_nmod_t op1,
    const fq_nmod_ctx_t ctx)
```

Sets `rop` to the difference of `op1` and 1.

```
void fq_nmod_neg(fq_nmod_t rop, const fq_nmod_t op, const
    fq_nmod_ctx_t ctx)
```

Sets `rop` to the negative of `op`.

```
void fq_nmod_mul(fq_nmod_t rop, const fq_nmod_t op1, const
    fq_nmod_t op2, const fq_nmod_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`, reducing the output in the given context.

```
void fq_nmod_mul_fmpz(fq_nmod_t rop, const fq_nmod_t op,
    const fmpz_t x, const fq_nmod_ctx_t ctx)
```

Sets `rop` to the product of `op` and x, reducing the output in the given context.

```
void fq_nmod_mul_si(fq_nmod_t rop, const fq_nmod_t op,
    slong x, const fq_nmod_ctx_t ctx)
```

Sets `rop` to the product of `op` and x, reducing the output in the given context.

```
void fq_nmod_mul_ui(fq_nmod_t rop, const fq_nmod_t op,
    ulong x, const fq_nmod_ctx_t ctx)
```

Sets `rop` to the product of `op` and x, reducing the output in the given context.

```
void fq_nmod_sqr(fq_nmod_t rop, const fq_nmod_t op, const
    fq_nmod_ctx_t ctx)
```

Sets `rop` to the square of `op`, reducing the output in the given context.

```
void _fq_nmod_inv(mp_ptr *rop, mp_srcptr *op, slong len,
    const fq_nmod_ctx_t ctx)
```

Sets `(rop, d)` to the inverse of the non-zero element `(op, len)`.

```
void fq_nmod_inv(fq_nmod_t rop, const fq_nmod_t op, const
    fq_nmod_ctx_t ctx)
```

Sets `rop` to the inverse of the non-zero element `op`.

```
void _fq_nmod_pow(mp_ptr *rop, mp_srcptr *op, slong len,
    const fmpz_t e, const fq_nmod_ctx_t ctx)
```

Sets `(rop, 2*d-1)` to `(op,len)` raised to the power e, reduced modulo $f(X)$, the modulus of `ctx`.

Assumes that $e \geq 0$ and that `len` is positive and at most d.

Although we require that `rop` provides space for $2d - 1$ coefficients, the output will be reduced modulo $f(X)$, which is a polynomial of degree d.

Does not support aliasing.

```
void fq_nmod_pow(fq_nmod_t rop, const fq_nmod_t op, const
    fmpz_t e, const fq_nmod_ctx_t ctx)
```

Sets `rop` the `op` raised to the power e.

Currently assumes that $e \geq 0$.

Note that for any input `op`, `rop` is set to 1 whenever $e = 0$.

```
void fq_nmod_pow_ui(fq_nmod_t rop, const fq_nmod_t op,
    const ulong e, const fq_nmod_ctx_t ctx)
```

Sets `rop` the `op` raised to the power e.

Currently assumes that $e \geq 0$.

Note that for any input `op`, `rop` is set to 1 whenever $e = 0$.

41.4 Roots

```
void fq_nmod_pth_root(fq_nmod_t rop, const fq_nmod_t op1,
    const fq_nmod_ctx_t ctx)
```

Sets `rop` to a p^{th} root root of `op1`. Currently, this computes the root by raising `op1` to p^{d-1} where d is the degree of the extension.

41.5 Output

```
int fq_nmod_fprint_pretty(FILE *file, const fq_nmod_t op,
    const fq_nmod_ctx_t ctx)
```

Prints a pretty representation of `op` to `file`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

```
int fq_nmod_print_pretty(const fq_nmod_t op, const
    fq_nmod_ctx_t ctx)
```

Prints a pretty representation of `op` to `stdout`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

```
void fq_nmod_fprint(FILE * file, const fq_nmod_t op, const
    fq_nmod_ctx_t ctx)
```

Prints a representation of `op` to `file`.

For further details on the representation used, see `nmod_poly_fprint()`.

```
void fq_nmod_print(const fq_nmod_t op, const fq_nmod_ctx_t
    ctx)
```

Prints a representation of `op` to `stdout`.

For further details on the representation used, see `nmod_poly_print()`.

```
char * fq_nmod_get_str(const fq_nmod_t op, const
    fq_nmod_ctx_t ctx)
```

Returns the plain FLINT string representation of the element `op`.

```
char * fq_nmod_get_str_pretty(const fq_nmod_t op, const
    fq_nmod_ctx_t ctx)
```

Returns a pretty representation of the element `op` using the null-terminated string `x` as the variable name.

41.6 Randomisation

```
void fq_nmod_randtest(fq_nmod_t rop, flint_rand_t state,
    const fq_nmod_ctx_t ctx)
```

Generates a random element of $\mathbb{F}_q$.

```
void fq_nmod_randtest_not_zero(fq_nmod_t rop, flint_rand_t
    state, const fq_nmod_ctx_t ctx)
```

Generates a random non-zero element of $\mathbb{F}_q$.

```
void fq_nmod_randtest_dense(fq_nmod_t rop, flint_rand_t
    state, const fq_nmod_ctx_t ctx)
```

Generates a random element of $\mathbb{F}_q$ which has an underlying polynomial with dense coefficients.

41.7 Assignments and conversions

```
void fq_nmod_set(fq_nmod_t rop, const fq_nmod_t op, const
    fq_nmod_ctx_t ctx)
```

Sets `rop` to `op`.

```
void fq_nmod_set_ui(fq_nmod_t rop, const ulong x, const
    fq_nmod_ctx_t ctx)
```

Sets `rop` to `x`, considered as an element of $\mathbb{F}_p$.

```
void fq_nmod_set_fmpz(fq_nmod_t rop, const fmpz_t x, const
   fq_nmod_ctx_t ctx)
```

Sets `rop` to `x`, considered as an element of $\mathbb{F}_p$.

```
void fq_nmod_swap(fq_nmod_t op1, fq_nmod_t op2, const
   fq_nmod_ctx_t ctx)
```

Swaps the two elements `op1` and `op2`.

```
void fq_nmod_zero(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
```

Sets `rop` to zero.

```
void fq_nmod_one(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
```

Sets `rop` to one, reduced in the given context.

```
void fq_nmod_gen(fq_nmod_t rop, const fq_nmod_ctx_t ctx)
```

Sets `rop` to a multiplicative generator for the finite field.

41.8 Comparison

```
int fq_nmod_is_zero(const fq_nmod_t op, const fq_nmod_ctx_t
   ctx)
```

Returns whether `op` is equal to zero.

```
int fq_nmod_is_one(const fq_nmod_t op, const fq_nmod_ctx_t
   ctx)
```

Returns whether `op` is equal to one.

```
int fq_nmod_equal(const fq_nmod_t op1, const fq_nmod_t op2,
   const fq_nmod_ctx_t ctx)
```

Returns whether `op1` and `op2` are equal.

```
int fq_nmod_is_invertible(const fq_nmod_t op, const
   fq_nmod_ctx_t ctx)
```

Returns whether `op` is an invertible element.

```
int fq_nmod_is_invertible_f(fq_nmod_t f, const fq_nmod_t
   op, const fq_nmod_ctx_t ctx)
```

Returns whether `op` is an invertible element. If it is not, then `f` is set of a factor of the modulus.

41.9 Special functions

```
void _fq_nmod_trace(fmpz_t rop, mp_srcptr *op, slong len,
   const fq_nmod_ctx_t ctx)
```

Sets `rop` to the trace of the non-zero element `(op, len)` in $\mathbf{F}_q$.

```
void fq_nmod_trace(fq_nmod_t rop, const fq_nmod_t op, const
   fq_nmod_ctx_t ctx)
```

Sets `rop` to the trace of `op`.

For an element $a \in \mathbb{F}_q$, multiplication by a defines a $\mathbb{F}_p$-linear map on $\mathbb{F}_q$. We define the trace of a as the trace of this map. Equivalently, if Σ generates $\mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)$ then the trace of a is equal to $\sum_{i=0}^{d-1} \Sigma^i(a)$, where $d = \log_p q$.

```
void _fq_nmod_norm(fmpz_t rop, mp_srcptr *op, slong len,
   const fq_nmod_ctx_t ctx)
```

Sets `rop` to the norm of the non-zero element `(op, len)` in $\mathbf{F}_q$.

```
void fq_nmod_norm(fq_nmod_t rop, const fq_nmod_t op, const
   fq_nmod_ctx_t ctx)
```

Computes the norm of `op`.

For an element $a \in \mathbb{F}_q$, multiplication by a defines a $\mathbb{F}_p$-linear map on $\mathbb{F}_q$. We define the norm of a as the determinant of this map. Equivalently, if Σ generates $\mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)$ then the trace of a is equal to $\prod_{i=0}^{d-1} \Sigma^i(a)$, where $d = \dim_{\mathbb{F}_p}(\mathbb{F}_q)$.

Algorithm selection is automatic depending on the input.

```
void _fq_nmod_frobenius(mp_ptr *rop, mp_srcptr *op, slong
   len, slong e, const fq_nmod_ctx_t ctx)
```

Sets `(rop, 2d-1)` to the image of `(op, len)` under the Frobenius operator raised to the e-th power, assuming that neither `op` nor `e` are zero.

```
void fq_nmod_frobenius(fq_nmod_t rop, const fq_nmod_t op,
   slong e, const fq_nmod_ctx_t ctx)
```

Evaluates the homomorphism Σ^e at `op`.

Recall that $\mathbb{F}_q/\mathbb{F}_p$ is Galois with Galois group $\langle\sigma\rangle$, which is also isomorphic to $\mathbf{Z}/d\mathbf{Z}$, where $\sigma \in \mathrm{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ is the Frobenius element $\sigma\colon x \mapsto x^p$.

41.10 Bit packing

```
void fq_nmod_bit_pack(fmpz_t f, const fq_nmod_t op,
   mp_bitcnt_t bit_size, const fq_nmod_ctx_t ctx)
```

Packs `op` into bitfields of size `bit_size`, writing the result to `f`.

```
void fq_nmod_bit_unpack(fq_nmod_t rop, const fmpz_t f,
   mp_bitcnt_t bit_size, const fq_nmod_ctx_t ctx)
```

Unpacks into `rop` the element with coefficients packed into fields of size `bit_size` as represented by the integer `f`.

§42. fq_nmod_vec

Vectors over finite fields of word-sized characteristic

42.1 Memory management

```
fq_nmod_struct * _fq_nmod_vec_init(slong len, const
   fq_nmod_ctx_t ctx)
```

Returns an initialised vector of `fq_nmod`'s of given length.

```
void _fq_nmod_vec_clear(fq_nmod * vec, slong len, const
   fq_nmod_ctx_t ctx)
```

Clears the entries of `(vec, len)` and frees the space allocated for `vec`.

42.2 Randomisation

```
void _fq_nmod_vec_randtest(fq_nmod_struct * f, flint_rand_t
   state, slong len, const fq_nmod_ctx_t ctx)
```

Sets the entries of a vector of the given length to elements of the finite field.

42.3 Input and output

```
int _fq_nmod_vec_fprint(FILE * file, const fq_nmod_struct *
   vec, slong len, const fq_nmod_ctx_t ctx)
```

Prints the vector of given length to the stream `file`. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_nmod_vec_print(const fq_nmod_struct * vec, slong
   len, const fq_nmod_ctx_t ctx)
```

Prints the vector of given length to `stdout`.

For further details, see `_fq_nmod_vec_fprint()`.

42.4 Assignment and basic manipulation

```
void _fq_nmod_vec_set(fq_nmod_struct * vec1, const
   fq_nmod_struct * vec2, slong len2, const fq_nmod_ctx_t
   ctx)
```

Makes a copy of `(vec2, len2)` into `vec1`.

```
void _fq_nmod_vec_swap(fq_nmod_struct * vec1,
   fq_nmod_struct * vec2, slong len2, const fq_nmod_ctx_t
   ctx)
```

Swaps the elements in `(vec1, len2)` and `(vec2, len2)`.

```
void _fq_nmod_vec_zero(fq_nmod_struct * vec, slong len,
   const fq_nmod_ctx_t ctx)
```

Zeros the entries of `(vec, len)`.

```
void _fq_nmod_vec_neg(fq_nmod_struct * vec1, const
   fq_nmod_struct * vec2, slong len2, const fq_nmod_ctx_t
   ctx)
```

Negates `(vec2, len2)` and places it into `vec1`.

42.5 Comparison

```
int _fq_nmod_vec_equal(const fq_nmod_struct * vec1, const
   fq_nmod_struct * vec2, slong len, const fq_nmod_ctx_t
   ctx)
```

Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

```
int _fq_nmod_vec_is_zero(const fq_nmod_struct * vec, slong
   len, const ctx_ctx)
```

Returns 1 if `(vec, len)` is zero, and 0 otherwise.

42.6 Addition and subtraction

```
void _fq_nmod_vec_add(fq_nmod_struct * res, const
   fq_nmod_struct * vec1, const fq_nmod_struct * vec2,
   slong len2, const fq_nmod_ctx_t ctx)
```

Sets `(res, len2)` to the sum of `(vec1, len2)` and `(vec2, len2)`.

```
void _fq_nmod_vec_sub(fq_nmod_struct * res, const
   fq_nmod_struct * vec1, const fq_nmod_struct * vec2,
   slong len2, const fq_nmod_ctx_t ctx)
```

Sets `(res, len2)` to `(vec1, len2)` minus `(vec2, len2)`.

42.7 Scalar multiplication and division

```
void _fq_nmod_vec_scalar_addmul_fq_nmod(fq_nmod_struct *
   vec1, const fq_nmod_struct * vec2, slong len2, const
   fq_nmod_t c, const fq_nmod_ctx_t ctx)
```

Adds `(vec2, len2)` times c to `(vec1, len2)`, where c is a `fq_nmod_t`.

```
void _fq_nmod_vec_scalar_submul_fq_nmod(fq_nmod_struct *
   vec1, const fq_nmod_struct * vec2, slong len2, const
   fq_nmod_t c, const fq_nmod_ctx_t ctx)
```

Subtracts `(vec2, len2)` times c from `(vec1, len2)`, where c is a `fq_nmod_t`.

42.8 Dot products

```
void _fq_nmod_vec_dot(fq_nmod_t res, const fq_nmod_struct *
   vec1, const fq_nmod_struct * vec2, slong len2, const
   fq_nmod_ctx_t ctx)
```

Sets `res` to the dot product of (`vec1`, `len`) and (`vec2`, `len`).

§43. fq_nmod_mat

Matrices over finite fields of word-sized characteristic

43.1 Memory management

```
void fq_nmod_mat_init(fq_nmod_mat_t mat, slong rows, slong
    cols, const fq_nmod_ctx_t ctx)
```

Initialises `mat` to a `rows`-by-`cols` matrix with coefficients in $\mathbb{F}_q$ given by `ctx`. All elements are set to zero.

```
void fq_nmod_mat_init_set(fq_nmod_mat_t mat, fq_nmod_mat_t
    src, const fq_nmod_ctx_t ctx)
```

Initialises `mat` and sets its dimensions and elements to those of `src`.

```
void fq_nmod_mat_clear(fq_nmod_mat_t mat, const
    fq_nmod_ctx_t ctx)
```

Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an `fq_nmod_mat_t` object.

```
void fq_nmod_mat_set(fq_nmod_mat_t mat, fq_nmod_mat_t src,
    const fq_nmod_ctx_t ctx)
```

Sets `mat` to a copy of `src`. It is assumed that `mat` and `src` have identical dimensions.

43.2 Basic properties and manipulation

```
fq_nmod_struct * fq_nmod_mat_entry(fq_nmod_mat_t mat, slong
    i, slong j)
```

Directly accesses the entry in `mat` in row i and column j, indexed from zero. No bounds checking is performed.

```
fq_nmod_struct * fq_nmod_mat_entry_set(fq_nmod_mat_t mat,
    slong i, slong j, fq_nmod_t x, const fq_nmod_ctx_t ctx)
```

Sets the entry in `mat` in row i and column j to `x`.

```
slong fq_nmod_mat_nrows(fq_nmod_mat_t mat, const
   fq_nmod_ctx_t ctx)
```

Returns the number of rows in `mat`.

```
slong fq_nmod_mat_ncols(fq_nmod_mat_t mat, const
   fq_nmod_ctx_t ctx)
```

Returns the number of columns in `mat`.

```
void fq_nmod_mat_swap(fq_nmod_mat_t mat1, fq_nmod_mat_t
   mat2, const fq_nmod_ctx_t ctx)
```

Swaps two matrices. The dimensions of `mat1` and `mat2` are allowed to be different.

```
void fq_nmod_mat_zero(fq_nmod_mat_t mat, const
   fq_nmod_ctx_t ctx)
```

Sets all entries of `mat` to 0.

43.3 Printing

```
void fq_nmod_mat_print_pretty(const fq_nmod_mat_t mat,
   const fq_nmod_ctx_t ctx)
```

Pretty-prints `mat` to `stdout`. A header is printed followed by the rows enclosed in brackets.

```
int fq_nmod_mat_fprint_pretty(FILE * file, const
   fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
```

Pretty-prints `mat` to `file`. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
void fq_nmod_mat_print(const fq_nmod_mat_t mat, const
   fq_nmod_ctx_t ctx)
```

Prints `mat` to `stdout`. A header is printed followed by the rows enclosed in brackets.

```
int fq_nmod_mat_fprint(FILE * file, const fq_nmod_mat_t
   mat, const fq_nmod_ctx_t ctx)
```

Prints `mat` to `file`. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

43.4 Window

```
void fq_nmod_mat_window_init(fq_nmod_mat_t window, const
   fq_nmod_mat_t mat, slong r1, slong c1, slong r2, slong
   c2, const fq_nmod_ctx_t ctx)
```

Initializes the matrix `window` to be an `r2 - r1` by `c2 - c1` submatrix of `mat` whose `(0,0)` entry is the `(r1, c1)` entry of `mat`. The memory for the elements of `window` is shared with `mat`.

```
void fq_nmod_mat_window_clear(fq_nmod_mat_t window, const
   fq_nmod_ctx_t ctx)
```

Clears the matrix `window` and releases any memory that it uses. Note that the memory to the underlying matrix that `window` points to is not freed.

43.5 Random matrix generation

```
void fq_nmod_mat_randtest(fq_nmod_mat_t mat, flint_rand_t
   state, const fq_nmod_ctx_t ctx)
```

Sets the elements of `mat` to random elements of $\mathbb{F}_q$, given by `ctx`.

```
int fq_nmod_mat_randpermdiag(fq_nmod_mat_t mat,
   fq_nmod_struct * diag, slong n, flint_rand_t state,
   const fq_nmod_ctx_t ctx)
```

Sets `mat` to a random permutation of the diagonal matrix with n leading entries given by the vector `diag`. It is assumed that the main diagonal of `mat` has room for at least n entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.

```
void fq_nmod_mat_randrank(fq_nmod_mat_t mat, slong rank,
   flint_rand_t state, const fq_nmod_ctx_t ctx)
```

Sets `mat` to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random elements of $\mathbb{F}_q$.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling `fq_nmod_mat_randops()`.

```
void fq_nmod_mat_randops(fq_nmod_mat_t mat, slong count,
   flint_rand_t state, const fq_nmod_ctx_t ctx)
```

Randomises `mat` by performing elementary row or column operations. More precisely, at most `count` random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

```
void fq_nmod_mat_randtril(fq_nmod_mat_t mat, flint_rand_t
   state, int unit, const fq_nmod_ctx_t ctx)
```

Sets `mat` to a random lower triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

```
void fq_nmod_mat_randtriu(fq_nmod_mat_t mat, flint_rand_t
   state, int unit, x      const fq_nmod_ctx_t ctx)
```

Sets `mat` to a random upper triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

43.6 Comparison

```
int fq_nmod_mat_equal(fq_nmod_mat_t mat1, fq_nmod_mat_t
   mat2, const fq_nmod_ctx_t ctx)
```

Returns nonzero if mat1 and mat2 have the same dimensions and elements, and zero otherwise.

```
int fq_nmod_mat_is_zero(const fq_nmod_mat_t mat, const
   fq_nmod_ctx_t ctx)
```

Returns a non-zero value if all entries `mat` are zero, and otherwise returns zero.

```
int fq_nmod_mat_is_empty(const fq_nmod_mat_t mat, const
    fq_nmod_ctx_t ctx)
```

Returns a non-zero value if the number of rows or the number of columns in `mat` is zero, and otherwise returns zero.

```
int fq_nmod_mat_is_square(const fq_nmod_mat_t mat, const
    fq_nmod_ctx_t ctx)
```

Returns a non-zero value if the number of rows is equal to the number of columns in `mat`, and otherwise returns zero.

43.7 Addition and subtraction

```
void fq_nmod_mat_add(fq_nmod_mat_t C, const fq_nmod_mat_t
    A, const fq_nmod_mat_t B,  const fq_nmod_ctx_t ctx)
```

Computes $C = A + B$. Dimensions must be identical.

```
void fq_nmod_mat_sub(fq_nmod_mat_t C, const fq_nmod_mat_t
    A, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)
```

Computes $C = A - B$. Dimensions must be identical.

```
void fq_nmod_mat_neg(fq_nmod_mat_t A, const fq_nmod_mat_t
    B, const fq_nmod_ctx_t ctx)
```

Sets $B = -A$. Dimensions must be identical.

43.8 Matrix multiplication

```
void fq_nmod_mat_mul(fq_nmod_mat_t C, const fq_nmod_mat_t
    A, const fq_nmod_mat_t B,  const fq_nmod_ctx_t ctx)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. This function automatically chooses between classical and KS multiplication.

```
void fq_nmod_mat_mul_classical(fq_nmod_mat_t C, const
    fq_nmod_mat_t A, const fq_nmod_mat_t B, const
    fq_nmod_ctx_t ctx)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Uses classical matrix multiplication.

```
void fq_nmod_mat_mul_KS(fq_nmod_mat_t C, const
    fq_nmod_mat_t A, const fq_nmod_mat_t B, const
    fq_nmod_ctx_t ctx)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Uses Kronecker substitution to perform the multiplication over the integers.

```
void fq_nmod_mat_submul(fq_nmod_mat_t D, const
    fq_nmod_mat_t C, const fq_nmod_mat_t A, const
    fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)
```

Sets $D = C + AB$. C and D may be aliased with each other but not with A or B.

43.9 LU decomposition

```
slong fq_nmod_mat_lu(slong * P, fq_nmod_mat_t A, int
    rank_check, const fq_nmod_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A.

If A is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r, U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and return 0 if A is detected to be rank-deficient.

This function calls `fq_nmod_mat_lu_recursive`.

```
slong fq_nmod_mat_lu_classical(slong * P, fq_nmod_mat_t A,
    int rank_check, const fq_nmod_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A. The behavior of this function is identical to that of `fq_nmod_mat_lu`. Uses Gaussian elimination.

```
slong fq_nmod_mat_lu_recursive(slong * P, fq_nmod_mat_t A,
    int rank_check, const fq_nmod_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A. The behavior of this function is identical to that of `fq_nmod_mat_lu`. Uses recursive block decomposition, switching to classical Gaussian elimination for sufficiently small blocks.

43.10 Reduced row echelon form

```
slong fq_nmod_mat_rref(fq_nmod_mat_t A, const fq_nmod_ctx_t
    ctx)
```

Puts A in reduced row echelon form and returns the rank of A.

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

43.11 Triangular solving

```
void fq_nmod_mat_solve_tril(fq_nmod_mat_t X, const
    fq_nmod_mat_t L, const fq_nmod_mat_t B, int unit, const
    fq_nmod_ctx_t ctx)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

```
void fq_nmod_mat_solve_tril_classical(fq_nmod_mat_t X,
    const fq_nmod_mat_t L, const fq_nmod_mat_t B, int unit,
    const fq_nmod_ctx_t ctx)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

```
void fq_nmod_mat_solve_tril_recursive(fq_nmod_mat_t X,
    const fq_nmod_mat_t L, const fq_nmod_mat_t B, int unit,
    const fq_nmod_ctx_t ctx)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$\begin{pmatrix} A & 0 \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}X \\ D^{-1}(Y - CA^{-1}X) \end{pmatrix}$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

```
void fq_nmod_mat_solve_triu(fq_nmod_mat_t X, const
    fq_nmod_mat_t U, const fq_nmod_mat_t B, int unit, const
    fq_nmod_ctx_t ctx)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

```
void fq_nmod_mat_solve_triu_classical(fq_nmod_mat_t X,
    const fq_nmod_mat_t U, const fq_nmod_mat_t B, int unit,
    const fq_nmod_ctx_t ctx)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

```
void fq_nmod_mat_solve_triu_recursive(fq_nmod_mat_t X,
    const fq_nmod_mat_t U, const fq_nmod_mat_t B, int unit,
    const fq_nmod_ctx_t ctx)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$\begin{pmatrix} A & B \\ 0 & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}(X - BD^{-1}Y) \\ D^{-1}Y \end{pmatrix}$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

§44. fq_nmod_poly

polynomials over finite fields of word-sized characteristic

We represent a polynomial in $\mathbf{F}_q[X]$ as a `struct` which includes an array `coeffs` with the coefficients, as well as the length `length` and the number `alloc` of coefficients for which memory has been allocated.

As a data structure, we call this polynomial *normalised* if the top coefficient is non-zero.

Unless otherwise stated here, all functions that deal with polynomials assume that the $\mathbf{F}_q$ context of said polynomials are compatible, i.e., it assumes that the fields are generated by the same polynomial.

44.1 Memory management

```
void fq_nmod_poly_init(fq_nmod_poly_t poly, const
   fq_nmod_ctx_t ctx)
```

Initialises `poly` for use, with context ctx, and setting its length to zero. A corresponding call to `fq_nmod_poly_clear()` must be made after finishing with the `fq_nmod_poly_t` to free the memory used by the polynomial.

```
void fq_nmod_poly_init2(fq_nmod_poly_t poly, slong alloc,
   const fq_nmod_ctx_t ctx)
```

Initialises `poly` with space for at least `alloc` coefficients and sets the length to zero. The allocated coefficients are all set to zero. A corresponding call to `fq_nmod_poly_clear()` must be made after finishing with the `fq_nmod_poly_t` to free the memory used by the polynomial.

```
void fq_nmod_poly_realloc(fq_nmod_poly_t poly, slong alloc,
   const fq_nmod_ctx_t ctx)
```

Reallocates the given polynomial to have space for `alloc` coefficients. If `alloc` is zero the polynomial is cleared and then reinitialised. If the current length is greater than `alloc` the polynomial is first truncated to length `alloc`.

```
void fq_nmod_poly_fit_length(fq_nmod_poly_t poly, slong
   len, const fq_nmod_ctx_t ctx)
```

If `len` is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least `len` coefficients. No data is lost when calling this function.

The function efficiently deals with the case where `fit_length` is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

```
void _fq_nmod_poly_set_length(fq_nmod_poly_t poly, slong
   newlen, const fq_nmod_ctx_t ctx)
```

Sets the coefficients of `poly` beyond `len` to zero and sets the length of `poly` to `len`.

```
void fq_nmod_poly_clear(fq_nmod_poly_t poly, const
   fq_nmod_ctx_t ctx)
```

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

```
void _fq_nmod_poly_normalise(fq_nmod_poly_t poly, const
   fq_nmod_ctx_t ctx)
```

Sets the length of `poly` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

```
void _fq_nmod_poly_normalise2(fq_nmod_struct *poly, slong
   *length, const fq_nmod_ctx_t ctx)
```

Sets the length `length` of `(poly,length)` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

```
void fq_nmod_poly_truncate(fq_nmod_poly_t poly, slong
   newlen, const fq_nmod_ctx_t ctx)
```

Truncates the polynomial to length at most n.

```
void _fq_nmod_poly_reverse(fq_nmod_struct* output, const
   fq_nmod_struct* input, slong len, slong m, const
   fq_nmod_ctx_t ctx)
```

Sets `output` to the reverse of `input`, which is of length `len`, but thinking of it as a polynomial of length `m`, notionally zero-padded if necessary. The length `m` must be non-negative, but there are no other restrictions. The polynomial `output` must have space for `m` coefficients.

```
void fq_nmod_poly_reverse(fq_nmod_poly_t output, const
   fq_nmod_poly_t input, slong m, const fq_nmod_ctx_t ctx)
```

Sets `output` to the reverse of `input`, thinking of it as a polynomial of length `m`, notionally zero-padded if necessary). The length `m` must be non-negative, but there are no other restrictions. The output polynomial will be set to length `m` and then normalised.

44.2 Polynomial parameters

```
long fq_nmod_poly_degree(fq_nmod_poly_t poly, const
   fq_nmod_ctx_t ctx)
```

Returns the degree of the polynomial `poly`.

```
long fq_nmod_poly_length(fq_nmod_poly_t poly, const
   fq_nmod_ctx_t ctx)
```

Returns the length of the polynomial `poly`.

```
fq_nmod_struct * fq_nmod_poly_lead(const fq_nmod_poly_t
   poly, const fq_nmod_ctx_t ctx)
```

Returns a pointer to the leading coefficient of `poly`, or `NULL` if `poly` is the zero polynomial.

44.3 Randomisation

```
void fq_nmod_poly_randtest(fq_nmod_poly_t f, flint_rand_t
   state, slong len, const fq_nmod_ctx_t ctx)
```

Sets f to a random polynomial of length at most `len` with entries in the field described by `ctx`.

```
void fq_nmod_poly_randtest_not_zero(fq_nmod_poly_t f,
   flint_rand_t state, slong len, const fq_nmod_ctx_t ctx)
```

Same as `fq_nmod_poly_randtest` but guarantees that the polynomial is not zero.

```
void fq_nmod_poly_randtest_monic(fq_nmod_poly_t f,
   flint_rand_t state, slong len, const fq_nmod_ctx_t ctx)
```

Sets f to a random monic polynomial of length `len` with entries in the field described by `ctx`.

```
void fq_nmod_poly_randtest_irreducible(fq_nmod_poly_t f,
   flint_rand_t state, slong len, const fq_nmod_ctx_t ctx)
```

Sets f to a random monic, irreducible polynomial of length `len` with entries in the field described by `ctx`.

44.4 Assignment and basic manipulation

```
void _fq_nmod_poly_set(fq_nmod_struct *rop, const
   fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)
```

Sets `(rop, len)` to `(op, len)`.

```
void fq_nmod_poly_set(fq_nmod_poly_t poly1, const
   fq_nmod_poly_t poly2, const fq_nmod_ctx_t ctx)
```

Sets the polynomial `poly1` to the polynomial `poly2`.

```
void fq_nmod_poly_set_fq_nmod(fq_nmod_poly_t poly, const
   fq_nmod_t c, const fq_nmod_ctx_t ctx)
```

Sets the polynomial `poly` to `c`.

```
void fq_nmod_poly_swap(fq_nmod_poly_t op1, fq_nmod_poly_t
   op2, const fq_nmod_ctx_t ctx)
```

Swaps the two polynomials `op1` and `op2`.

```
void _fq_nmod_poly_zero(fq_nmod_struct *rop, slong len,
   const fq_nmod_ctx_t ctx)
```

Sets `(rop, len)` to the zero polynomial.

```
void fq_nmod_poly_zero(fq_nmod_poly_t poly, const
   fq_nmod_ctx_t ctx)
```

Sets poly to the zero polynomial.

```
void void fq_nmod_poly_one(fq_nmod_poly_t poly, const
   fq_nmod_ctx_t ctx)
```

Sets poly to the constant polynomial 1.

```
void void fq_nmod_poly_gen(fq_nmod_poly_t poly, const
   fq_nmod_ctx_t ctx)
```

Sets poly to the polynomial x.

```
void fq_nmod_poly_make_monic(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Sets rop to op, normed to have leading coefficient 1.

```
void _fq_nmod_poly_make_monic(fq_nmod_struct *rop, const
   fq_nmod_struct *op, slong length, const fq_nmod_ctx_t
   ctx)
```

Sets rop to (op,length), normed to have leading coefficient 1. Assumes that rop has enough space for the polynomial, assumes that op is not zero (and thus has an invertible leading coefficient).

44.5 Getting and setting coefficients

```
void fq_nmod_poly_get_coeff(fq_nmod_t x, const
   fq_nmod_poly_t poly, slong n, const fq_nmod_ctx_t ctx)
```

Sets x to the coefficient of X^n in poly.

```
void fq_nmod_poly_set_coeff(fq_nmod_poly_t poly, slong n,
   const fq_nmod_t x, const fq_nmod_ctx_t ctx)
```

Sets the coefficient of X^n in poly to x.

```
void fq_nmod_poly_set_coeff_fmpz(fq_nmod_poly_t poly, slong
   n, const fmpz_t x, const fq_nmod_ctx_t ctx)
```

Sets the coefficient of X^n in the polynomial to x, assuming $n \geq 0$.

44.6 Comparison

```
int fq_nmod_poly_equal(const fq_nmod_poly_t poly1, const
   fq_nmod_poly_t poly2, const fq_nmod_ctx_t ctx)
```

Returns whether the two polynomials poly1 and poly2 are equal.

```
int fq_nmod_poly_is_zero(const fq_nmod_poly_t poly, const
   fq_nmod_ctx_t ctx)
```

Returns whether the polynomial poly is the zero polynomial.

```
int fq_nmod_poly_is_one(const fq_nmod_poly_t op)
```

Returns whether the polynomial poly is equal to the constant polynomial 1.

```
int fq_nmod_poly_is_gen(const fq_nmod_poly_t op, const
   fq_nmod_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal to the polynomial x.

```
int fq_nmod_poly_is_unit(const fq_nmod_poly_t op, const
    fq_nmod_ctx_t ctx)
```

Returns whether the polynomial `poly` is a unit in the polynomial ring $\mathbf{F}_q[X]$, i.e. if it has degree 0 and is non-zero.

```
int fq_nmod_poly_equal_fq_nmod(const fq_nmod_poly_t poly,
    const fq_nmod_t c, const fq_nmod_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal the (constant) $\mathbf{F}_q$ element `c`

44.7 Addition and subtraction

```
void _fq_nmod_poly_add(fq_nmod_struct *res, const
    fq_nmod_struct *poly1, slong len1, const fq_nmod_struct
    *poly2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets `res` to the sum of `(poly1,len1)` and `(poly2,len2)`.

```
void fq_nmod_poly_add(fq_nmod_poly_t res, const
    fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const
    fq_nmod_ctx_t ctx)
```

Sets `res` to the sum of `poly1` and `poly2`.

```
void _fq_nmod_poly_sub(fq_nmod_struct *res, const
    fq_nmod_struct *poly1, slong len1, const fq_nmod_struct
    *poly2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets `res` to the difference of `(poly1,len1)` and `(poly2,len2)`.

```
void fq_nmod_poly_sub(fq_nmod_poly_t res, const
    fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const
    fq_nmod_ctx_t ctx)
```

Sets `res` to the difference of `poly1` and `poly2`.

```
void _fq_nmod_poly_neg(fq_nmod_struct *rop, const
    fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)
```

Sets `res` to the additive inverse of `(poly,len)`.

```
void fq_nmod_poly_neg(fq_nmod_poly_t res, const
    fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
```

Sets `res` to the additive inverse of `poly`.

44.8 Scalar multiplication and division

```
void _fq_nmod_poly_scalar_mul_fq_nmod(fq_nmod_struct *rop,
    const fq_nmod_struct *op, slong len, const fq_nmod_t x,
    const fq_nmod_ctx_t ctx)
```

Sets `(rop,len)` to the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`.

```
void fq_nmod_poly_scalar_mul_fq_nmod(fq_nmod_poly_t rop,
    const fq_nmod_poly_t op, const fq_nmod_t x, const
    fq_nmod_ctx_t ctx)
```

Sets `(rop,len)` to the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`.

```
void _fq_nmod_poly_scalar_addmul_fq_nmod(fq_nmod_struct
   *rop, const fq_nmod_struct *op, slong len, const
   fq_nmod_t x, const fq_nmod_ctx_t ctx)
```

Adds to `(rop,len)` the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`. In particular, assumes the same length for `op` and `rop`.

```
void fq_nmod_poly_scalar_addmul_fq_nmod(fq_nmod_poly_t rop,
   const fq_nmod_poly_t op, const fq_nmod_t x, const
   fq_nmod_ctx_t ctx)
```

Adds to `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.

```
void _fq_nmod_poly_scalar_submul_fq_nmod(fq_nmod_struct
   *rop, const fq_nmod_struct *op, slong len, const
   fq_nmod_t x, const fq_nmod_ctx_t ctx)
```

Substracts from `(rop,len)` the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`. In particular, assumes the same length for `op` and `rop`.

```
void fq_nmod_poly_scalar_submul_fq_nmod(fq_nmod_poly_t rop,
   const fq_nmod_poly_t op, const fq_nmod_t x, const
   fq_nmod_ctx_t ctx)
```

Substracts from `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.

44.9 Multiplication

```
void _fq_nmod_poly_mul_classical(fq_nmod_struct *rop, const
   fq_nmod_struct *op1, slong len1, const fq_nmod_struct
   *op2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, assuming that `len1` is at least `len2` and neither is zero.

Permits zero padding. Does not support aliasing of `rop` with either `op1` or `op2`.

```
void fq_nmod_poly_mul_classical(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const
   fq_nmod_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2` using classical polynomial multiplication.

```
void _fq_nmod_poly_mul_reorder(fq_nmod_struct *rop, const
   fq_nmod_struct *op1, slong len1, const fq_nmod_struct
   *op2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, assuming that `len1` and `len2` are non-zero.

Permits zero padding. Supports aliasing.

```
void fq_nmod_poly_mul_reorder(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const
   fq_nmod_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`, reordering the two indeterminates X and Y when viewing the polynomials as elements of $\mathbf{F}_p[X, Y]$.

Suppose $\mathbf{F}_q = \mathbf{F}_p[X]/(f(X))$ and recall that elements of $\mathbf{F}_q$ are internally represented by elements of type `fmpz_poly`. For small degree extensions but polynomials in $\mathbf{F}_q[Y]$ of large degree n, we change the representation to

$$\begin{aligned} g(Y) &= \sum_{i=0}^{n} a_i(X) Y^i \\ &= \sum_{j=0}^{d} \sum_{i=0}^{n} \text{Coeff}(a_i(X), j) Y^i. \end{aligned}$$

This allows us to use a poor algorithm (such as classical multiplication) in the X-direction and leverage the existing fast integer multiplication routines in the Y-direction where the polynomial degree n is large.

```
void _fq_nmod_poly_mul_KS(fq_nmod_struct *rop, const
   fq_nmod_struct *op1, slong len1, const fq_nmod_struct
   *op2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`.

Permits zero padding and places no assumptions on the lengths `len1` and `len2`. Supports aliasing.

```
void fq_nmod_poly_mul_KS(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const
   fq_nmod_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2` using Kronecker substitution, that is, by encoding each coefficient in $\mathbf{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

```
void _fq_nmod_poly_mul(fq_nmod_struct *rop, const
   fq_nmod_struct *op1, slong len1, const fq_nmod_struct
   *op2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

```
void fq_nmod_poly_mul(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const
   fq_nmod_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`, choosing an appropriate algorithm.

```
void _fq_nmod_poly_mullow_classical(fq_nmod_struct *rop,
   const fq_nmod_struct *op1, slong len1, const
   fq_nmod_struct *op2, slong len2, slong n, const
   fq_nmod_ctx_t ctx)
```

Sets `(res, n)` to the first n coefficients of `(poly1, len1)` multiplied by `(poly2, len2)`.

Assumes `0 < n <= len1 + len2 - 1`. Assumes neither `len1` nor `len2` is zero.

```
void fq_nmod_poly_mullow_classical(fq_nmod_poly_t rop,
   const fq_nmod_poly_t op1, const fq_nmod_poly_t op2,
   slong n, const fq_nmod_ctx_t ctx)
```

Sets `res` to the product of `poly1` and `poly2`, computed using the classical or schoolbook method.

```
void _fq_nmod_poly_mullow_KS(fq_nmod_struct *rop, const
   fq_nmod_struct *op1, slong len1, const fq_nmod_struct
   *op2, slong len2, slong n, const fq_nmod_ctx_t ctx)
```

Sets `(res, n)` to the lowest n coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `len1` and `len2` are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between `res`, `poly1` and `poly2`.

```
void fq_nmod_poly_mullow_KS(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op1, const fq_nmod_poly_t op2, slong n,
   const fq_nmod_ctx_t ctx)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _fq_nmod_poly_mullow(fq_nmod_struct *rop, const
   fq_nmod_struct *op1, slong len1, const fq_nmod_struct
   *op2, slong len2, slong n, const fq_nmod_ctx_t ctx)
```

Sets `(res, n)` to the lowest n coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes `0 < n <= len1 + len2 - 1`. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

```
void fq_nmod_poly_mullow(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op1, const fq_nmod_poly_t op2, slong n,
   const fq_nmod_ctx_t ctx)
```

Sets `res` to the lowest n coefficients of the product of `poly1` and `poly2`.

```
void _fq_nmod_poly_mulmod(fq_nmod_struct* res, const
   fq_nmod_struct* poly1, slong len1, const fq_nmod_struct*
   poly2, slong len2, const fq_nmod_struct* f, slong lenf,
   const fq_nmod_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_fq_nmod_poly_mul` instead.

Aliasing of `f` and `res` is not permitted.

```
void fq_nmod_poly_mulmod(fq_nmod_poly_t res,const
   fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const
   fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

```
void _fq_nmod_poly_mulmod_preinv(fq_nmod_struct* res, const
   fq_nmod_struct* poly1, slong len1, const fq_nmod_struct*
   poly2, slong len2, const fq_nmod_struct* f, slong lenf,
   const fq_nmod_struct* finv, slong lenfinv, const
   fq_nmod_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `finv` is the inverse of the reverse of `f` mod `x^lenf`. It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_fq_nmod_poly_mul` instead.

Aliasing of `f` or `finv` and `res` is not permitted.

```
void fq_nmod_poly_mulmod_preinv(fq_nmod_poly_t res, const
   fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const
   fq_nmod_poly_t f, const fq_nmod_poly_t finv, const
   fq_nmod_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`. `finv` is the inverse of the reverse of `f`.

44.10 Squaring

```
void _fq_nmod_poly_sqr_classical(fq_nmod_struct *rop, const
   fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `(op,len)` is not zero and using classical polynomial multiplication.

Permits zero padding. Does not support aliasing of `rop` with either `op1` or `op2`.

```
void fq_nmod_poly_sqr_classical(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Sets `rop` to the square of `op` using classical polynomial multiplication.

```
void _fq_nmod_poly_sqr_KS(fq_nmod_struct *rop, const
   fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`.

Permits zero padding and places no assumptions on the lengths `len1` and `len2`. Supports aliasing.

```
void fq_nmod_poly_sqr_KS(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Sets `rop` to the square `op` using Kronecker substitution, that is, by encoding each coefficient in $\mathbf{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

```
void _fq_nmod_poly_sqr(fq_nmod_struct *rop, const
   fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)
```

Sets `(rop, 2* len - 1)` to the square of `(op, len)`, choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

```
void fq_nmod_poly_sqr(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Sets rop to the square of op, choosing an appropriate algorithm.

44.11 Powering

```
void _fq_nmod_poly_pow(fq_nmod_struct *rop, const
   fq_nmod_struct *op, slong len, ulong e, const
   fq_nmod_ctx_t ctx)
```

Sets res = poly^e, assuming that e, len > 0 and that res has space for e*(len - 1)+ 1 coefficients. Does not support aliasing.

```
void fq_nmod_poly_pow(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op, ulong e, const fq_nmod_ctx_t ctx)
```

Computes res = poly^e. If e is zero, returns one, so that in particular 0^0 = 1.

```
void _fq_nmod_poly_powmod_ui_binexp(fq_nmod_struct* res,
   const fq_nmod_struct* poly, ulong e, const
   fq_nmod_struct* f, slong lenf, const fq_nmod_ctx_t ctx)
```

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0.

We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

```
void fq_nmod_poly_powmod_ui_binexp(fq_nmod_poly_t res,
   const fq_nmod_poly_t poly, ulong e, const fq_nmod_poly_t
   f, const fq_nmod_ctx_t ctx)
```

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0.

```
void _fq_nmod_poly_powmod_ui_binexp_preinv(fq_nmod_struct*
   res, const fq_nmod_struct* poly, ulong e, const
   fq_nmod_struct* f, slong lenf, const fq_nmod_struct*
   finv, slong lenfinv, const fq_nmod_ctx_t ctx)
```

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0. We require finv to be the inverse of the reverse of f.

We require lenf > 1. It is assumed that poly is already reduced modulo f and zero-padded as necessary to have length exactly lenf - 1. The output res must have room for lenf - 1 coefficients.

```
void fq_nmod_poly_powmod_ui_binexp_preinv(fq_nmod_poly_t
   res, const fq_nmod_poly_t poly, ulong e, const
   fq_nmod_poly_t f, const fq_nmod_poly_t finv, const
   fq_nmod_ctx_t ctx)
```

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e >= 0. We require finv to be the inverse of the reverse of f.

```
void _fq_nmod_poly_powmod_fmpz_binexp(fq_nmod_struct* res,
   const fq_nmod_struct* poly, fmpz_t e, const
   fq_nmod_struct* f, slong lenf, const fq_nmod_ctx_t ctx)
```

Sets res to poly raised to the power e modulo f, using binary exponentiation. We require e > 0.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_nmod_poly_powmod_fmpz_binexp(fq_nmod_poly_t res,
    const fq_nmod_poly_t poly, fmpz_t e, const
    fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void
    _fq_nmod_poly_powmod_fmpz_binexp_preinv(fq_nmod_struct*
    res, const fq_nmod_struct* poly, fmpz_t e, const
    fq_nmod_struct* f, slong lenf, const fq_nmod_struct*
    finv, slong lenfinv, const fq_nmod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_nmod_poly_powmod_fmpz_binexp_preinv(fq_nmod_poly_t
    res, const fq_nmod_poly_t poly, fmpz_t e, const
    fq_nmod_poly_t f, const fq_nmod_poly_t finv, const
    fq_nmod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void
    _fq_nmod_poly_powmod_fmpz_sliding_preinv(fq_nmod_struct*
    res, const fq_nmod_struct* poly, fmpz_t e, ulong k,
    const fq_nmod_struct* f, slong lenf, const
    fq_nmod_struct* finv, slong lenfinv, const fq_nmod_ctx_t
    ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an "optimum" size will be selected automatically base on `e`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_nmod_poly_powmod_fmpz_sliding_preinv(fq_nmod_poly_t
    res, const fq_nmod_poly_t poly, fmpz_t e, ulong k, const
    fq_nmod_poly_t f, const fq_nmod_poly_t finv, const
    fq_nmod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an "optimum" size will be selected automatically base on `e`.

```
void _fq_nmod_poly_powmod_x_fmpz_preinv(fq_nmod_struct *
    res, const fmpz_t e, const fq_nmod_struct * f, slong
    lenf, const fq_nmod_struct * finv, slong lenfinv, const
    fq_nmod_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 2`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_nmod_poly_powmod_x_fmpz_preinv(fq_nmod_poly_t res,
   const fmpz_t e, const fq_nmod_poly_t f, const
   fq_nmod_poly_t finv, const fq_nmod_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

44.12 Shifting

```
void _fq_nmod_poly_shift_left(fq_nmod_struct *rop, const
   fq_nmod_struct *op, slong len, slong n, const
   fq_nmod_ctx_t ctx)
```

Sets `(res, len + n)` to `(poly, len)` shifted left by n coefficients.

Inserts zero coefficients at the lower end. Assumes that `len` and n are positive, and that `res` fits `len + n` elements. Supports aliasing between `res` and `poly`.

```
void fq_nmod_poly_shift_left(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op, slong n, const fq_nmod_ctx_t ctx)
```

Sets `res` to `poly` shifted left by n coeffs. Zero coefficients are inserted.

```
void _fq_nmod_poly_shift_right(fq_nmod_struct *rop, const
   fq_nmod_struct *op, slong len, slong n, const
   fq_nmod_ctx_t ctx)
```

Sets `(res, len - n)` to `(poly, len)` shifted right by n coefficients.

Assumes that `len` and n are positive, that `len > n`, and that `res` fits `len - n` elements. Supports aliasing between `res` and `poly`, although in this case the top coefficients of `poly` are not set to zero.

```
void fq_nmod_poly_shift_right(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op, slong n, const fq_nmod_ctx_t ctx)
```

Sets `res` to `poly` shifted right by n coefficients. If n is equal to or greater than the current length of `poly`, `res` is set to the zero polynomial.

44.13 Norms

```
long _fq_nmod_poly_hamming_weight(const fq_nmod_poly *op,
   slong len, const fq_nmod_ctx_t ctx)
```

Returns the number of non-zero entries in `(op, len)`.

```
long fq_nmod_poly_hamming_weight(const fq_nmod_poly_t op,
   const fq_nmod_ctx_t ctx)
```

Returns the number of non-zero entries in the polynomial `op`.

44.14 Euclidean division

```
void _fq_nmod_poly_divrem_basecase(fq_nmod_struct *Q,
   fq_nmod_struct *R, const fq_nmod_struct *A, slong lenA,
   const fq_nmod_struct *B, slong lenB, const fq_nmod_t
   invB, const fq_nmod_ctx_t ctx)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ with $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is its inverse.

Assumes that $\operatorname{len}(A), \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fq_nmod_poly_divrem_basecase(fq_nmod_poly_t Q,
   fq_nmod_poly_t R, const fq_nmod_poly_t A, const
   fq_nmod_poly_t B, const fq_nmod_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ with $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible. This can be taken for granted the context is for a finite field, that is, when p is prime and $f(X)$ is irreducible.

```
void _fq_nmod_poly_divrem(fq_nmod_struct *Q, fq_nmod_struct
   *R, const fq_nmod_struct *A, slong lenA, const
   fq_nmod_struct *B, slong lenB, const fq_nmod_t invB,
   const fq_nmod_ctx_t ctx)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ with $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is its inverse.

Assumes that $\operatorname{len}(A), \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fq_nmod_poly_divrem(fq_nmod_poly_t Q, fq_nmod_poly_t
   R, const fq_nmod_poly_t A, const fq_nmod_poly_t B, const
   fq_nmod_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ with $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible. This can be taken for granted the context is for a finite field, that is, when p is prime and $f(X)$ is irreducible.

```
void _fq_nmod_poly_rem(fq_nmod_struct *R, const
   fq_nmod_struct *A, slong lenA, const fq_nmod_struct *B,
   slong lenB, const fq_nmod_t invB, const fq_nmod_ctx_t
   ctx)
```

Sets `R` to the remainder of the division of `(A,lenA)` by `(B,lenB)`. Assumes that the leading coefficient of `(B,lenB)` is invertible and that `invB` is its inverse.

```
void fq_nmod_poly_rem(fq_nmod_poly_t R, const
   fq_nmod_poly_t A, const fq_nmod_poly_t B, const
   fq_nmod_ctx_t ctx)
```

Sets `R` to the remainder of the division of `A` by `B` in the context described by `ctx`.

```
void _fq_nmod_poly_div_basecase(fq_nmod_struct *Q,
   fq_nmod_struct *R, const fq_nmod_struct *A, slong lenA,
   const fq_nmod_struct *B, slong lenB, const fq_nmod_t
   invB, const fq_nmod_ctx_t ctx)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets `(Q, lenA - lenB + 1)`.

Requires temporary space `(R, lenA)`. If `R` is `NULL`, then the temporary space will be allocated. Allows aliasing only between A and R. Allows zero-padding in A but not in B. Assumes that the leading coefficient of B is a unit.

```
void fq_nmod_poly_div_basecase(fq_nmod_poly_t Q, const
   fq_nmod_poly_t A, const fq_nmod_poly_t B, const
   fq_nmod_ctx_t ctx)
```

Notionally finds polynomials Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only `Q`. If $\text{len}(B) = 0$ an exception is raised.

```
void
   _fq_nmod_poly_divrem_divconquer_recursive(fq_nmod_struct
   * Q, fq_nmod_struct * BQ, fq_nmod_struct * W, const
   fq_nmod_struct * A, const fq_nmod_struct * B, slong
   lenB, const fq_nmod_t invB, const fq_nmod_ctx_t ctx)
```

Computes `(Q, lenB)`, `(BQ, 2 lenB - 1)` such that $BQ = B \times Q$ and $A = BQ + R$ where $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is the inverse.

Assumes $\text{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. Requires a temporary array `(W, 2 lenB - 1)`. No aliasing of input and output operands is allowed.

This function does not read the bottom $\text{len}(B) - 1$ coefficients from A, which means that they might not even need to exist in allocated memory.

```
void _fq_nmod_poly_divrem_divconquer(fq_nmod_struct * Q,
   fq_nmod_struct * R, const fq_nmod_struct * A, slong
   lenA, const fq_nmod_struct * B, slong lenB, const
   fq_nmod_t invB, const fq_nmod_ctx_t ctx)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is the inverse.

Assumes $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. No aliasing of input and output operands is allowed.

```
void fq_nmod_poly_divrem_divconquer(fq_nmod_poly_t Q,
   fq_nmod_poly_t R, const fq_nmod_poly_t A, const
   fq_nmod_poly_t B, const fq_nmod_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that B is non-zero and that the leading coefficient of B is invertible.

```
void _fq_nmod_poly_div_newton_n_preinv(fq_nmod_struct* Q,
   const fq_nmod_struct* A, slong lenA, const
   fq_nmod_struct* B, slong lenB, const fq_nmod_struct*
   Binv, slong lenBinv, const fq_nmod_struct ctx_t)
```

Notionally computes polynomials Q and R such that $A = BQ+R$ with $\text{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`, but return only Q.

We require that Q have space for `lenA - lenB + 1` coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void fq_nmod_poly_div_newton_n_preinv(fq_nmod_poly_t Q,
    const fq_nmod_poly_t A, const fq_nmod_poly_t B, const
    fq_nmod_poly_t Binv, const fq_nmod_ctx_t ctx)
```

Notionally computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$, but returns only Q.

We assume that the leading coefficient of B is a unit and that $Binv$ is the inverse of the reverse of B mod $x^{\operatorname{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void _fq_nmod_poly_divrem_newton_n_preinv(fq_nmod_struct*
    Q, fq_nmod_struct* R, const fq_nmod_struct* A, slong
    lenA, const fq_nmod_struct* B, slong lenB, const
    fq_nmod_struct* Binv, slong lenBinv, const fq_nmod_ctx_t
    ctx)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R)$ less than `lenB`, where A is of length `lenA` and B is of length `lenB`. We require that Q have space for `lenA - lenB + 1` coefficients. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\operatorname{len}(B)}$. The algorithm used is to call `div_newton_preinv()` and then multiply out and compute the remainder.

```
void fq_nmod_poly_divrem_newton_n_preinv(fq_nmod_poly_t Q,
    fq_nmod_poly_t R, const fq_nmod_poly_t A, const
    fq_nmod_poly_t B, const fq_nmod_poly_t Binv, const
    fq_nmod_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$. We assume $Binv$ is the inverse of the reverse of B mod $x^{\operatorname{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to call `div_newton()` and then multiply out and compute the remainder.

```
void _fq_nmod_poly_inv_series_newton(fq_nmod_struct* Qinv,
    const fq_nmod_struct* Q, slong n, const fq_nmod_ctx_t
    ctx)
```

Given `Q` of length `n` whose constant coefficient is invertible modulo the given modulus, find a polynomial `Qinv` of length `n` such that `Q * Qinv` is `1` modulo x^n. Requires `n > 0`. This function can be viewed as inverting a power series via Newton iteration.

```
void fq_nmod_poly_inv_series_newton(fq_nmod_poly_t Qinv,
    const fq_nmod_poly_t Q, slong n, const fq_nmod_ctx_t ctx)
```

Given `Q` find `Qinv` such that `Q * Qinv` is `1` modulo x^n. The constant coefficient of `Q` must be invertible modulo the modulus of `Q`. An exception is raised if this is not the case or if `n = 0`. This function can be viewed as inverting a power series via Newton iteration.

44.15 Greatest common divisor

```
void fq_nmod_poly_gcd(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const
   fq_nmod_ctx_t ctx)
```

Sets `rop` to the greatest common divisor of `op1` and `op2`, using the either the Euclidean or HGCD algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

```
long _fq_nmod_poly_gcd(fq_nmod_struct* G,const
   fq_nmod_struct* A, slong lenA, const fq_nmod_struct* B,
   slong lenB, const fq_nmod_ctx_t ctx)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= lenB > 0` and sets G to it. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for `lenB` coefficients.

```
void fq_nmod_poly_gcd_euclidean(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const
   fq_nmod_ctx_t ctx)
```

Sets `rop` to the greatest common divisor of `op1` and `op2`, using the Euclidean algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

```
long _fq_nmod_poly_gcd_euclidean(fq_nmod_struct* G, const
   fq_nmod_struct* A, slong lenA, const fq_nmod_struct* B,
   slong lenB, const fq_nmod_ctx_t ctx)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= lenB > 0` and sets G to it. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for `lenB` coefficients.

44.16 Divisibility testing

```
int _fq_nmod_poly_divides(fq_nmod_struct *Q, const
   fq_nmod_struct *A, slong lenA, const fq_nmod_struct *B,
   slong lenB, const fq_nmod_t invB, const fq_nmod_ctx_t
   ctx)
```

Returns 1 if `(B, lenB)` divides `(A, lenA)` exactly and sets Q to the quotient, otherwise returns 0.

It is assumed that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$ and that Q has space for $\operatorname{len}(A) - \operatorname{len}(B) + 1$ coefficients.

Aliasing of Q with either of the inputs is not permitted.

This function is currently unoptimised and provided for convenience only.

```
int fq_nmod_poly_divides(fq_nmod_poly_t Q, const
   fq_nmod_poly_t A, const fq_nmod_poly_t B, const
   fq_nmod_ctx_t ctx)
```

Returns 1 if B divides A exactly and sets Q to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.

44.17 Derivative

```
void _fq_nmod_poly_derivative(fq_nmod_struct *rop, const
   fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)
```

Sets `(rpoly, len - 1)` to the derivative of `(poly, len)`. Also handles the cases where `len` is 0 or 1 correctly. Supports aliasing of `rpoly` and `poly`.

```
void fq_nmod_poly_derivative(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Sets `res` to the derivative of `poly`.

44.18 Evaluation

```
void _fq_nmod_poly_evaluate_fq_nmod(fq_nmod_t rop, const
   fq_nmod_struct *op, slong len, const fq_nmod_t a, const
   fq_nmod_ctx_t ctx)
```

Sets `rop` to `(op, len)` evaluated at a.

Supports zero padding. There are no restrictions on `len`, that is, `len` is allowed to be zero, too.

```
void fq_nmod_poly_evaluate_fq_nmod(fq_nmod_t rop, const
   fq_nmod_poly_t f, const fq_nmod_t a, const fq_nmod_ctx_t
   ctx)
```

Sets `rop` to the value of $f(a)$.

As the coefficient ring $\mathbf{F}_q$ is finite, Horner's method is sufficient.

44.19 Composition

```
void _fq_nmod_poly_compose_divconquer(fq_nmod_struct *rop,
   const fq_nmod_struct *op1, slong len1, const
   fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)
```

Computes the composition of `(op1, len1)` and `(op2, len2)` using a divide and conquer approach and places the result into `rop`, assuming `rop` can hold the output of length `(len1 - 1)* (len2 - 1)+ 1`.

Assumes `len1, len2 > 0`. Does not support aliasing between `rop` and any of `(op1, len1)` and `(op2, len2)`.

```
void fq_nmod_poly_compose_divconquer(fq_nmod_poly_t rop,
   const fq_nmod_poly_t op1, const fq_nmod_poly_t op2,
   const fq_nmod_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be precise about the order of composition, denoting `rop`, `op1`, and `op2` by f, g, and h, respectively, sets $f(t) = g(h(t))$.

```
void _fq_nmod_poly_compose_horner(fq_nmod_struct *rop,
   const fq_nmod_struct *op1, slong len1, const
   fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets `rop` to the composition of `(op1, len1)` and `(op2, len2)`.

Assumes that `rop` has space for `(len1-1)*(len2-1)+ 1` coefficients. Assumes that `op1` and `op2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fq_nmod_poly_compose_horner(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const
   fq_nmod_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be more precise, denoting `rop`, `op1`, and `op2` by f, g, and h, sets $f(t) = g(h(t))$.

This implementation uses Horner's method.

```
void _fq_nmod_poly_compose(fq_nmod_struct *rop, const
   fq_nmod_struct *op1, slong len1, const fq_nmod_struct
   *op2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets `rop` to the composition of `(op1, len1)` and `(op2, len2)`.

Assumes that `rop` has space for `(len1-1)*(len2-1)+ 1` coefficients. Assumes that `op1` and `op2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fq_nmod_poly_compose(fq_nmod_poly_t rop, const
   fq_nmod_poly_t op1, const fq_nmod_poly_t op2, const
   fq_nmod_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be precise about the order of composition, denoting `rop`, `op1`, and `op2` by f, g, and h, respectively, sets $f(t) = g(h(t))$.

```
void _fq_nmod_poly_compose_mod_horner(fq_nmod_struct * res,
   const fq_nmod_struct * f, slong lenf, const
   fq_nmod_struct * g, const fq_nmod_struct * h, slong
   lenh, const fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_nmod_poly_compose_mod_horner(fq_nmod_poly_t res,
   const fq_nmod_poly_t f, const fq_nmod_poly_t g, const
   fq_nmod_poly_t h, const fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero. The algorithm used is Horner's rule.

```
void _fq_nmod_poly_compose_mod_horner_preinv(fq_nmod_struct
   * res, const fq_nmod_struct * f, slong lenf, const
   fq_nmod_struct * g, const fq_nmod_struct * h, slong
   lenh, const fq_nmod_struct * hinv, slong lenhiv, const
   fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_nmod_poly_compose_mod_horner_preinv(fq_nmod_poly_t
   res, const fq_nmod_poly_t f, const fq_nmod_poly_t g,
   const fq_nmod_poly_t h, const fq_nmod_poly_t hinv, const
   fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The algorithm used is Horner's rule.

```
void _fq_nmod_poly_compose_mod_brent_kung(fq_nmod_struct *
   res, const fq_nmod_struct * f, slong lenf, const
   fq_nmod_struct * g, const fq_nmod_struct * h, slong
   lenh, const fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_nmod_poly_compose_mod_brent_kung(fq_nmod_poly_t
   res, const fq_nmod_poly_t f, const fq_nmod_poly_t g,
   const fq_nmod_poly_t h, const fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. The algorithm used is the Brent-Kung matrix algorithm.

```
void
   _fq_nmod_poly_compose_mod_brent_kung_preinv(fq_nmod_struct
   * res, const fq_nmod_struct * f, slong lenf, const
   fq_nmod_struct * g, const fq_nmod_struct * h, slong
   lenh, const fq_nmod_struct * hinv, slong lenhiv, const
   fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void
   fq_nmod_poly_compose_mod_brent_kung_preinv(fq_nmod_poly_t
   res, const fq_nmod_poly_t f, const fq_nmod_poly_t g,
   const fq_nmod_poly_t h, const fq_nmod_poly_t hinv, const
   fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The algorithm used is the Brent-Kung matrix algorithm.

```
void _fq_nmod_poly_compose_mod(fq_nmod_struct * res, const
   fq_nmod_struct * f, slong lenf, const fq_nmod_struct *
   g, const fq_nmod_struct * h, slong lenh, const
   fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```
void fq_nmod_poly_compose_mod(fq_nmod_poly_t res, const
   fq_nmod_poly_t f, const fq_nmod_poly_t g, const
   fq_nmod_poly_t h, const fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero.

```
void _fq_nmod_poly_compose_mod_preinv(fq_nmod_struct * res,
   const fq_nmod_struct * f, slong lenf, const
   fq_nmod_struct * g, const fq_nmod_struct * h, slong
   lenh, const fq_nmod_struct * hinv, slong lenhiv, const
   fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

```
void fq_nmod_poly_compose_mod_preinv(fq_nmod_poly_t res,
   const fq_nmod_poly_t f, const fq_nmod_poly_t g, const
   fq_nmod_poly_t h, const fq_nmod_poly_t hinv, const
   fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`.

```
void _fq_nmod_poly_reduce_matrix_mod_poly (fq_nmod_mat_t A,
   const fq_nmod_mat_t B, const fq_nmod_poly_t f, const
   fq_nmod_ctx_t ctx)
```

Sets the ith row of `A` to the reduction of the ith row of B modulo f for $i = 1, \ldots, \sqrt{\deg(f)}$. We require B to be at least a $\sqrt{\deg(f)} \times \deg(f)$ matrix and f to be nonzero.

```
void _fq_nmod_poly_precompute_matrix (fq_nmod_mat_t A,
   const fq_nmod_struct* f, const fq_nmod_struct* g, slong
   leng, const fq_nmod_struct* ginv, slong lenginv, const
   fq_nmod_ctx_t ctx)
```

Sets the ith row of `A` to f^i modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of `g` and g to be nonzero.

```
void fq_nmod_poly_precompute_matrix (fq_nmod_mat_t A, const
   fq_nmod_poly_t f, const fq_nmod_poly_t g, const
   fq_nmod_poly_t ginv, const fq_nmod_ctx_t ctx)
```

Sets the ith row of `A` to f^i modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of `g`.

```
void
   _fq_nmod_poly_compose_mod_brent_kung_precomp_preinv(fq_nmod_struct*
   res, const fq_nmod_struct* f, slong lenf, const
   fq_nmod_mat_t A, const fq_nmod_struct* h, slong h, const
   fq_nmod_struct* hinv, slong lenhinv, const fq_nmod_ctx_t
   ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero. We require that the ith row of A contains g^i for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void
    fq_nmod_poly_compose_mod_brent_kung_precomp_preinv(fq_nmod_poly_t
    res, const fq_nmod_poly_t f, const fq_nmod_mat_t A,
    const fq_nmod_poly_t h, const fq_nmod_poly_t hinv, const
    fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that the ith row of A contains g^i for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo h for fixed g and h.

44.20 Output

```
int _fq_nmod_poly_fprint_pretty(FILE *file, const
    fq_nmod_struct *poly, slong len, const char *x, const
    fq_nmod_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_nmod_poly_fprint_pretty(FILE * file, const
    fq_nmod_poly_t poly, const char *x, const fq_nmod_ctx_t
    ctx)
```

Prints the pretty representation of `poly` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_nmod_poly_print_pretty(const fq_nmod_struct *poly,
    slong len, const char *x, const fq_nmod_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_nmod_poly_print_pretty(const fq_nmod_poly_t poly,
    const char *x, const fq_nmod_ctx_t ctx)
```

Prints the pretty representation of `poly` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_nmod_poly_fprint(FILE *file, const fq_nmod_struct
    *poly, slong len, const fq_nmod_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_nmod_poly_fprint(FILE * file, const fq_nmod_poly_t
    poly, const fq_nmod_ctx_t ctx)
```

Prints the pretty representation of `poly` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_nmod_poly_print(const fq_nmod_struct *poly, slong
    len, const fq_nmod_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_nmod_poly_print(const fq_nmod_poly_t poly, const
    fq_nmod_ctx_t ctx)
```

Prints the representation of `poly` to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
char * _fq_nmod_poly_get_str(const fq_nmod_struct * poly,
    slong len, const fq_nmod_ctx_t ctx)
```

Returns the plain FLINT string representation of the polynomial `(poly, len)`.

```
char * fq_nmod_poly_get_str(const fq_nmod_poly_t poly,
    const fq_nmod_ctx_t ctx)
```

Returns the plain FLINT string representation of the polynomial `poly`.

```
char * _fq_nmod_poly_get_str_pretty(const fq_nmod_struct *
    poly, slong len, const char * x, const fq_nmod_ctx_t ctx)
```

Returns a pretty representation of the polynomial `(poly, len)` using the null-terminated string `x` as the variable name.

```
char * fq_nmod_poly_get_str_pretty(const fq_nmod_poly_t
    poly, const char * x, const fq_nmod_ctx_t ctx)
```

Returns a pretty representation of the polynomial `poly` using the null-terminated string `x` as the variable name

44.21 Inflation and deflation

```
void fq_nmod_poly_inflate(fq_nmod_poly_t result, const
    fq_nmod_poly_t input, ulong inflation, const
    fq_nmod_ctx_t ctx)
```

Sets `result` to the inflated polynomial $p(x^n)$ where p is given by `input` and n is given by `inflation`.

```
void fq_nmod_poly_deflate(fq_nmod_poly_t result, const
    fq_nmod_poly_t input, ulong deflation, const
    fq_nmod_ctx_t ctx)
```

Sets `result` to the deflated polynomial $p(x^{1/n})$ where p is given by `input` and n is given by `deflation`. Requires $n > 0$.

```
ulong fq_nmod_poly_deflation(const fq_nmod_poly_t input,
    const fq_nmod_ctx_t ctx)
```

Returns the largest integer by which `input` can be deflated. As special cases, returns 0 if `input` is the zero polynomial and 1 of `input` is a constant polynomial.

§45. fq_nmod_poly_factor

Factorisation of polynomials over finite fields of word-sized characteristic

---

The `fq_nmod__poly_factor` module is included automatically when one includes `fq_nmod_poly.h`. One should not try to include `fq_nmod_poly_factor.h` directly.

45.1 Memory Management

```
void fq_nmod_poly_factor_init(fq_nmod_poly_factor_t fac,
   const fq_nmod_ctx_t ctx)
```

Initialises `fac` for use. An `fq_nmod_poly_factor_t` represents a polynomial in factorised form as a product of polynomials with associated exponents.

```
void fq_nmod_poly_factor_clear(fq_nmod_poly_factor_t fac,
   const fq_nmod_ctx_t ctx)
```

Frees all memory associated with `fac`.

```
void fq_nmod_poly_factor_realloc(fq_nmod_poly_factor_t fac,
   slong alloc, const fq_nmod_ctx_t ctx)
```

Reallocates the factor structure to provide space for precisely `alloc` factors.

```
void fq_nmod_poly_factor_fit_length(fq_nmod_poly_factor_t
   fac, slong len, const fq_nmod_ctx_t ctx)
```

Ensures that the factor structure has space for at least `len` factors. This functions takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

45.2 Basic Operations

```
void fq_nmod_poly_factor_set(fq_nmod_poly_factor_t res,
   const fq_nmod_poly_factor_t fac, const fq_nmod_ctx_t ctx)
```

Sets `res` to the same factorisation as `fac`.

```
void fq_nmod_poly_factor_print_pretty(const
   fq_nmod_poly_factor_t fac, const fq_nmod_ctx_t ctx)
```

Pretty-prints the entries of `fac` to standard output.

```
void fq_nmod_poly_factor_print(const fq_nmod_poly_factor_t
   fac, const fq_nmod_ctx_t ctx)
```

Prints the entries of `fac` to standard output.

```
void fq_nmod_poly_factor_insert(fq_nmod_poly_factor_t fac,
   const fq_nmod_poly_t poly, slong exp, const
   fq_nmod_ctx_t ctx)
```

Inserts the factor `poly` with multiplicity `exp` into the factorisation `fac`.

If `fac` already contains `poly`, then `exp` simply gets added to the exponent of the existing entry.

```
void fq_nmod_poly_factor_concat(fq_nmod_poly_factor_t res,
   const fq_nmod_poly_factor_t fac, const fq_nmod_ctx_t ctx)
```

Concatenates two factorisations.

This is equivalent to calling `fq_nmod_poly_factor_insert()` repeatedly with the individual factors of `fac`.

Does not support aliasing between `res` and `fac`.

```
void fq_nmod_poly_factor_pow(fq_nmod_poly_factor_t fac,
   slong exp, const fq_nmod_ctx_t ctx)
```

Raises `fac` to the power `exp`.

```
ulong fq_nmod_poly_remove(fq_nmod_poly_t f, const
   fq_nmod_poly_t p, const fq_nmod_ctx_t ctx)
```

Removes the highest possible power of `p` from `f` and returns the exponent.

45.3 Irreducibility Testing

```
int fq_nmod_poly_is_irreducible(const fq_nmod_poly_t f,
   const fq_nmod_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0.

```
int fq_nmod_poly_is_irreducible_ddf(const fq_nmod_poly_t f,
   const fq_nmod_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

```
int fq_nmod_poly_is_irreducible_ben_or(const fq_nmod_poly_t
   f, const fq_nmod_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses Ben-Or's irreducibility test.

```
int _fq_nmod_poly_is_squarefree(const fq_nmod_struct * f,
   slong len, const fq_nmod_ctx_t ctx)
```

Returns 1 if `(f, len)` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

```
int fq_nmod_poly_is_squarefree(const fq_nmod_poly_t f,
    const fq_nmod_ctx_t ctx)
```

Returns 1 if `f` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

45.4 Factorisation

```
int fq_nmod_poly_factor_equal_deg_prob(fq_nmod_poly_t
    factor, flint_rand_t state, const fq_nmod_poly_t pol,
    slong d, const fq_nmod_ctx_t ctx)
```

Probabilistic equal degree factorisation of `pol` into irreducible factors of degree `d`. If it passes, a factor is placed in factor and 1 is returned, otherwise 0 is returned and the value of factor is undetermined.

Requires that `pol` be monic, non-constant and squarefree.

```
void fq_nmod_poly_factor_equal_deg(fq_nmod_poly_factor_t
    factors, const fq_nmod_poly_t pol, slong d, const
    fq_nmod_ctx_t ctx)
```

Assuming `pol` is a product of irreducible factors all of degree `d`, finds all those factors and places them in factors. Requires that `pol` be monic, non-constant and squarefree.

```
void fq_nmod_poly_factor_distinct_deg(fq_nmod_poly_factor_t
    res, const fq_nmod_poly_t poly, slong * const *degs,
    const fq_nmod_ctx_t ctx)
```

Factorises a monic non-constant squarefree polymnomial `poly` of degree n into factors $f[d]$ such that for $1 \leq d \leq n$ $f[d]$ is the product of the monic irreducible factors of `poly` of degree d. Factors are stored in `res`, assotiated powers of irreducible polynomials are stored in `degs` in the same order as factors.

Requires that `degs` have enough space for irreducible polynomials' powers (maximum space required is $n * sizeof(slong)$).

```
void fq_nmod_poly_factor_squarefree(fq_nmod_poly_factor_t
    res, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)
```

Sets `res` to a squarefree factorization of `f`.

```
void fq_nmod_poly_factor(fq_nmod_poly_factor_t res, const
    fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors choosing the best algorithm for given modulo and degree. Choise is based on heuristic measurments.

```
void
    fq_nmod_poly_factor_cantor_zassenhaus(fq_nmod_poly_factor_t
    res, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the Cantor-Zassenhaus algorithm.

```
void
    fq_nmod_poly_factor_kaltofen_shoup(fq_nmod_poly_factor_t
    res, const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a âĂIJbaby step/giant stepâĂİ strategy for the distinct-degree factorization step.

```
void fq_nmod_poly_factor_berlekamp(fq_nmod_poly_factor_t
   factors, const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the Berlekamp algorithm.

```
void
   fq_nmod_poly_factor_with_berlekamp(fq_nmod_poly_factor_t
   res, fq_nmod_t leading_coeff, const fq_nmod_poly_t f,
   const fq_nmod_ctx_t)
```

Factorises a general polynomial `f` into monic irreducible factors and sets `leading_coeff` to the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Berlekamp on all the individual square-free factors.

```
void
   fq_nmod_poly_factor_with_cantor_zassenhaus(fq_nmod_poly_factor_t
   res, fq_nmod_t leading_coeff const fq_nmod_poly_t f,
   const fq_nmod_ctx_t ctx)
```

Factorises a general polynomial `f` into monic irreducible factors and sets `leading_coeff` to the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

```
void
   fq_nmod_poly_factor_with_kaltofen_shoup(fq_nmod_poly_factor_t
   res, fq_nmod_t leading_coeff, const fq_nmod_poly_t f,
   const fq_nmod_ctx_t ctx)
```

Factorises a general polynomial `f` into monic irreducible factors and sets `leading_coeff` to the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

```
void fq_nmod_poly_iterated_frobenius_preinv(fq_nmod_poly_t
   *rop, slong n, const fq_nmod_poly_t v, const
   fq_nmod_poly_t vinv, const fq_nmod_ctx_t ctx)
```

Sets `rop[i]` to be $x^{q^i} mod v$ for $0 <= i < n$.

It is required that `vinv` is the inverse of the reverse of `v` mod `x^lenv`.

§46. fq_zech

Finite fields in Zech logarithm representation

We represent an element of the finite field as a power of a generator for the multiplicative group of the finite field. In particular, we use a root of $f(x)$), where $f(X) \in \mathbb{F}_p[X]$ is a monic, irreducible polynomial of degree n, as a polynomial in $\mathbb{F}_p[X]$ of degree less than n. The underlying data structure is just an `mp_limb_t`.

The default choice for $f(X)$ is the Conway polynomial for the pair (p, n). Frank Luebeck's data base of Conway polynomials is made available in the file `qadic/CPimport.txt`. If a Conway polynomial is not available, then a random irreducible polynomial will be chosen for $f(X)$. Additionally, the user is able to supply their own $f(X)$.

We required that the order of the field fits inside of an `mp_limb_t`; however, it is recommended that $p^n < 2^{20}$ due to the time and memory needed to compute the Zech logarithm table.

46.1 Context Management

```
void fq_zech_ctx_init(fq_zech_ctx_t ctx, const fmpz_t p,
   slong d, const char *var)
```

Initialises the context for prime p and extension degree d, with name `var` for the generator. By default, it will try use a Conway polynomial; if one is not available, a random irreducible polynomial will be used.

Assumes that p is a prime and $p^d < 2^{FLINT_BITS}$.

Assumes that the string `var` is a null-terminated string of length at least one.

```
int _fq_zech_ctx_init_conway(fq_zech_ctx_t ctx, const
   fmpz_t p, slong d, const char *var)
```

Attempts to initialise the context for prime p and extension degree d, with name `var` for the generator using a Conway polynomial for the modulus.

Returns 1 if the Conway polynomial is in the database for the given size and the initialization is successful; otherwise, returns 0.

Assumes that p is a prime and $p^d < 2^{FLINT_BITS}$.

Assumes that the string `var` is a null-terminated string of length at least one.

```
void fq_zech_ctx_init_conway(fq_zech_ctx_t ctx, const
   fmpz_t p, slong d, const char *var)
```

Initialises the context for prime p and extension degree d, with name `var` for the generator using a Conway polynomial for the modulus.

Assumes that p is a prime and $p^d < 2^{FLINT_BITS}$.

Assumes that the string `var` is a null-terminated string of length at least one.

```
void fq_zech_ctx_init_modulus(fq_zech_ctx_t ctx nmod_poly_t
   modulus, const char *var)
```

Initialises the context for given `modulus` with name `var` for the generator.

Assumes that `modulus` is and irreducible polynomial over $\mathbf{F}_p$.

Assumes that the string `var` is a null-terminated string of length at least one.

```
void fq_zech_ctx_init_fq_nmod_ctx(fq_zech_ctx_t ctx,
   fq_nmod_ctx_t ctxn);
```

Initializes the context `ctx` to be the Zech representation for the finite field given by `ctxn`.

```
void fq_zech_ctx_clear(fq_zech_ctx_t ctx)
```

Clears all memory that has been allocated as part of the context.

```
long fq_zech_ctx_degree(const fq_zech_ctx_t ctx)
```

Returns the degree of the field extension $[\mathbf{F}_q : \mathbf{F}_p]$, which is equal to $\log_p q$.

```
fmpz * fq_zech_ctx_prime(const fq_zech_ctx_t ctx)
```

Returns a pointer to the prime p in the context.

```
void fq_zech_ctx_order(fmpz_t f, const fq_zech_ctx_t ctx)
```

Sets f to be the size of the finite field.

```
mp_limb_t fq_zech_ctx_order_ui(const fq_zech_ctx_t ctx)
```

Returns the size of the finite field.

```
int fq_zech_ctx_fprint(FILE * file, const fq_zech_ctx_t ctx)
```

Prints the context information to `file`. Returns 1 for a success and a negative number for an error.

```
void fq_zech_ctx_print(const fq_zech_ctx_t ctx)
```

Prints the context information to `stdout`.

```
void fq_zech_ctx_randtest(fq_zech_ctx_t ctx)
```

Initializes `ctx` to a random finite field. Assumes that `fq_zech_ctx_init` as not been called on `ctx` already.

```
void fq_zech_ctx_randtest_reducible(fq_zech_ctx_t ctx)
```

Since the Zech logarithm representation does not work with a non-irreducible modulus, does the same as `fq_zech_ctx_randtest`.

46.2 Memory management

```
void fq_zech_init(fq_zech_t rop, const fq_zech_ctx_t ctx)
```

Initialises the element `rop`, setting its value to 0.

```
void fq_zech_init2(fq_zech_t rop, const fq_zech_ctx_t ctx)
```

Initialises `poly` with at least enough space for it to be an element of `ctx` and sets it to 0.

```
void fq_zech_clear(fq_zech_t rop, const fq_zech_ctx_t ctx)
```

Clears the element `rop`.

```
void _fq_zech_sparse_reduce(mp_ptr R, slong lenR, const
   fq_zech_ctx_t ctx)
```

Reduces `(R, lenR)` modulo the polynomial f given by the modulus of `ctx`.

```
void _fq_zech_dense_reduce(mp_ptr R, slong lenR, const
   fq_zech_ctx_t ctx)
```

Reduces `(R, lenR)` modulo the polynomial f given by the modulus of `ctx` using Newton division.

```
void _fq_zech_reduce(mp_ptr r, slong lenR, const
   fq_zech_ctx_t ctx)
```

Reduces `(R, lenR)` modulo the polynomial f given by the modulus of `ctx`. Does either sparse or dense reduction based on `ctx->sparse_modulus`.

```
void fq_zech_reduce(fq_zech_t rop, const fq_zech_ctx_t ctx)
```

Reduces the polynomial `rop` as an element of $\mathbf{F}_p[X]/(f(X))$.

46.3 Basic arithmetic

```
void fq_zech_add(fq_zech_t rop, const fq_zech_t op1, const
   fq_zech_t op2, const fq_zech_ctx_t ctx)
```

Sets `rop` to the sum of `op1` and `op2`.

```
void fq_zech_sub(fq_zech_t rop, const fq_zech_t op1, const
   fq_zech_t op2, const fq_zech_ctx_t ctx)
```

Sets `rop` to the difference of `op1` and `op2`.

```
void fq_zech_sub_one(fq_zech_t rop, const fq_zech_t op1,
   const fq_zech_ctx_t ctx)
```

Sets `rop` to the difference of `op1` and 1.

```
void fq_zech_neg(fq_zech_t rop, const fq_zech_t op, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to the negative of `op`.

```
void fq_zech_mul(fq_zech_t rop, const fq_zech_t op1, const
   fq_zech_t op2, const fq_zech_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`, reducing the output in the given context.

```
void fq_zech_mul_fmpz(fq_zech_t rop, const fq_zech_t op,
   const fmpz_t x, const fq_zech_ctx_t ctx)
```

Sets `rop` to the product of `op` and x, reducing the output in the given context.

```
void fq_zech_mul_si(fq_zech_t rop, const fq_zech_t op,
   slong x, const fq_zech_ctx_t ctx)
```

Sets `rop` to the product of `op` and x, reducing the output in the given context.

```
void fq_zech_mul_ui(fq_zech_t rop, const fq_zech_t op,
   ulong x, const fq_zech_ctx_t ctx)
```

Sets `rop` to the product of `op` and x, reducing the output in the given context.

```
void fq_zech_sqr(fq_zech_t rop, const fq_zech_t op, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to the square of `op`, reducing the output in the given context.

```
void _fq_zech_inv(mp_ptr *rop, mp_srcptr *op, slong len,
   const fq_zech_ctx_t ctx)
```

Sets `(rop, d)` to the inverse of the non-zero element `(op, len)`.

```
void fq_zech_inv(fq_zech_t rop, const fq_zech_t op, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to the inverse of the non-zero element `op`.

```
void _fq_zech_pow(mp_ptr *rop, mp_srcptr *op, slong len,
   const fmpz_t e, const fq_zech_ctx_t ctx)
```

Sets `(rop, 2*d-1)` to `(op,len)` raised to the power e, reduced modulo $f(X)$, the modulus of `ctx`.

Assumes that $e \geq 0$ and that `len` is positive and at most d.

Although we require that `rop` provides space for $2d - 1$ coefficients, the output will be reduced modulo $f(X)$, which is a polynomial of degree d.

Does not support aliasing.

```
void fq_zech_pow(fq_zech_t rop, const fq_zech_t op, const
   fmpz_t e, const fq_zech_ctx_t ctx)
```

Sets `rop` the `op` raised to the power e.

Currently assumes that $e \geq 0$.

Note that for any input `op`, `rop` is set to 1 whenever $e = 0$.

```
void fq_zech_pow_ui(fq_zech_t rop, const fq_zech_t op,
   const ulong e, const fq_zech_ctx_t ctx)
```

Sets `rop` the `op` raised to the power e.

Currently assumes that $e \geq 0$.

Note that for any input `op`, `rop` is set to 1 whenever $e = 0$.

46.4 Roots

```
void fq_zech_pth_root(fq_zech_t rop, const fq_zech_t op1,
   const fq_zech_ctx_t ctx)
```

Sets `rop` to a p^{th} root root of `op1`. Currently, this computes the root by raising `op1` to p^{d-1} where d is the degree of the extension.

46.5 Output

```
int fq_zech_fprint_pretty(FILE *file, const fq_zech_t op,
    const fq_zech_ctx_t ctx)
```

Prints a pretty representation of `op` to `file`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

```
int fq_zech_print_pretty(const fq_zech_t op, const
    fq_zech_ctx_t ctx)
```

Prints a pretty representation of `op` to `stdout`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

```
void fq_zech_fprint(FILE * file, const fq_zech_t op, const
    fq_zech_ctx_t ctx)
```

Prints a representation of `op` to `file`.

```
void fq_zech_print(const fq_zech_t op, const fq_zech_ctx_t
    ctx)
```

Prints a representation of `op` to `stdout`.

```
char * fq_zech_get_str(const fq_zech_t op, const
    fq_zech_ctx_t ctx)
```

Returns the plain FLINT string representation of the element `op`.

```
char * fq_zech_get_str_pretty(const fq_zech_t op, const
    fq_zech_ctx_t ctx)
```

Returns a pretty representation of the element `op` using the null-terminated string `x` as the variable name.

46.6 Randomisation

```
void fq_zech_randtest(fq_zech_t rop, flint_rand_t state,
    const fq_zech_ctx_t ctx)
```

Generates a random element of $\mathbb{F}_q$.

```
void fq_zech_randtest_not_zero(fq_zech_t rop, flint_rand_t
    state, const fq_zech_ctx_t ctx)
```

Generates a random non-zero element of $\mathbb{F}_q$.

```
void fq_zech_randtest_dense(fq_zech_t rop, flint_rand_t
    state, const fq_zech_ctx_t ctx)
```

Generates a random element of $\mathbb{F}_q$ which has an underlying polynomial with dense coefficients.

46.7 Assignments and conversions

```
void fq_zech_set(fq_zech_t rop, const fq_zech_t op, const
    fq_zech_ctx_t ctx)
```

Sets `rop` to `op`.

```
void fq_zech_set_ui(fq_zech_t rop, const ulong x, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to `x`, considered as an element of $\mathbb{F}_p$.

```
void fq_zech_set_fmpz(fq_zech_t rop, const fmpz_t x, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to `x`, considered as an element of $\mathbb{F}_p$.

```
void fq_zech_swap(fq_zech_t op1, fq_zech_t op2, const
   fq_zech_ctx_t ctx)
```

Swaps the two elements `op1` and `op2`.

```
void fq_zech_zero(fq_zech_t rop, const fq_zech_ctx_t ctx)
```

Sets `rop` to zero.

```
void fq_zech_one(fq_zech_t rop, const fq_zech_ctx_t ctx)
```

Sets `rop` to one, reduced in the given context.

```
void fq_zech_gen(fq_zech_t rop, const fq_zech_ctx_t ctx)
```

Sets `rop` to a multiplicative generator for the finite field.

```
void fq_zech_get_fq_nmod(fq_nmod_t rop, const fq_zech_t op,
   const fq_zech_ctx_t ctx)
```

Sets `rop` to the `fq_nmod_t` element corresponding to `op`.

```
void fq_zech_set_fq_nmod(fq_zech_t rop, const fq_nmod_t op,
   const fq_zech_ctx_t ctx)
```

Sets `rop` to the `fq_zech_t` element corresponding to `op`.

46.8 Comparison

```
int fq_zech_is_zero(const fq_zech_t op, const fq_zech_ctx_t
   ctx)
```

Returns whether `op` is equal to zero.

```
int fq_zech_is_one(const fq_zech_t op, const fq_zech_ctx_t
   ctx)
```

Returns whether `op` is equal to one.

```
int fq_zech_equal(const fq_zech_t op1, const fq_zech_t op2,
   const fq_zech_ctx_t ctx)
```

Returns whether `op1` and `op2` are equal.

```
int fq_zech_is_invertible(const fq_zech_t op, const
   fq_zech_ctx_t ctx)
```

Returns whether `op` is an invertible element.

```
int fq_zech_is_invertible_f(fq_zech_t f, const fq_zech_t
    op, const fq_zech_ctx_t ctx)
```

Returns whether `op` is an invertible element. If it is not, then `f` is set of a factor of the modulus. Since the modulus for an `fq_zech_ctx_t` is always irreducible, then any non-zero `op` will be invertible.

46.9 Special functions

```
void fq_zech_trace(fq_zech_t rop, const fq_zech_t op, const
    fq_zech_ctx_t ctx)
```

Sets `rop` to the trace of `op`.

For an element $a \in \mathbb{F}_q$, multiplication by a defines a $\mathbb{F}_p$-linear map on $\mathbb{F}_q$. We define the trace of a as the trace of this map. Equivalently, if Σ generates $\mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)$ then the trace of a is equal to $\sum_{i=0}^{d-1} \Sigma^i(a)$, where $d = \log_p q$.

```
void fq_zech_norm(fq_zech_t rop, const fq_zech_t op, const
    fq_zech_ctx_t ctx)
```

Computes the norm of `op`.

For an element $a \in \mathbb{F}_q$, multiplication by a defines a $\mathbb{F}_p$-linear map on $\mathbb{F}_q$. We define the norm of a as the determinant of this map. Equivalently, if Σ generates $\mathrm{Gal}(\mathbb{F}_q/\mathbb{F}_p)$ then the trace of a is equal to $\prod_{i=0}^{d-1} \Sigma^i(a)$, where $d = \dim_{\mathbb{F}_p}(\mathbb{F}_q)$.

Algorithm selection is automatic depending on the input.

```
void fq_zech_frobenius(fq_zech_t rop, const fq_zech_t op,
    slong e, const fq_zech_ctx_t ctx)
```

Evaluates the homomorphism Σ^e at `op`.

Recall that $\mathbb{F}_q/\mathbb{F}_p$ is Galois with Galois group $\langle\sigma\rangle$, which is also isomorphic to $\mathbf{Z}/d\mathbf{Z}$, where $\sigma \in \mathrm{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ is the Frobenius element $\sigma\colon x \mapsto x^p$.

46.10 Bit packing

```
void fq_zech_bit_pack(fmpz_t f, const fq_zech_t op,
    mp_bitcnt_t bit_size, const fq_zech_ctx_t ctx)
```

Packs `op` into bitfields of size `bit_size`, writing the result to `f`.

```
void fq_zech_bit_unpack(fq_zech_t rop, const fmpz_t f,
    mp_bitcnt_t bit_size, const fq_zech_ctx_t ctx)
```

Unpacks into `rop` the element with coefficients packed into fields of size `bit_size` as represented by the integer `f`.

§47. fq_zech_vec

Vectors over finite fields in Zech logarithm representation

47.1 Memory management

```
fq_zech_struct * _fq_zech_vec_init(slong len, const
   fq_zech_ctx_t ctx)
```

Returns an initialised vector of `fq_zech`'s of given length.

```
void _fq_zech_vec_clear(fq_zech * vec, slong len, const
   fq_zech_ctx_t ctx)
```

Clears the entries of `(vec, len)` and frees the space allocated for `vec`.

47.2 Randomisation

```
void _fq_zech_vec_randtest(fq_zech_struct * f, flint_rand_t
   state, slong len, const fq_zech_ctx_t ctx)
```

Sets the entries of a vector of the given length to elements of the finite field.

47.3 Input and output

```
int _fq_zech_vec_fprint(FILE * file, const fq_zech_struct *
   vec, slong len, const fq_zech_ctx_t ctx)
```

Prints the vector of given length to the stream `file`. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_zech_vec_print(const fq_zech_struct * vec, slong
   len, const fq_zech_ctx_t ctx)
```

Prints the vector of given length to `stdout`.

For further details, see `_fq_zech_vec_fprint()`.

47.4 Assignment and basic manipulation

```
void _fq_zech_vec_set(fq_zech_struct * vec1, const
   fq_zech_struct * vec2, slong len2, const fq_zech_ctx_t
   ctx)
```

Makes a copy of `(vec2, len2)` into `vec1`.

```
void _fq_zech_vec_swap(fq_zech_struct * vec1,
   fq_zech_struct * vec2, slong len2, const fq_zech_ctx_t
   ctx)
```

Swaps the elements in `(vec1, len2)` and `(vec2, len2)`.

```
void _fq_zech_vec_zero(fq_zech_struct * vec, slong len,
   const fq_zech_ctx_t ctx)
```

Zeros the entries of `(vec, len)`.

```
void _fq_zech_vec_neg(fq_zech_struct * vec1, const
   fq_zech_struct * vec2, slong len2, const fq_zech_ctx_t
   ctx)
```

Negates `(vec2, len2)` and places it into `vec1`.

47.5 Comparison

```
int _fq_zech_vec_equal(const fq_zech_struct * vec1, const
   fq_zech_struct * vec2, slong len, const fq_zech_ctx_t
   ctx)
```

Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

```
int _fq_zech_vec_is_zero(const fq_zech_struct * vec, slong
   len, const ctx_ctx)
```

Returns 1 if `(vec, len)` is zero, and 0 otherwise.

47.6 Addition and subtraction

```
void _fq_zech_vec_add(fq_zech_struct * res, const
   fq_zech_struct * vec1, const fq_zech_struct * vec2,
   slong len2, const fq_zech_ctx_t ctx)
```

Sets `(res, len2)` to the sum of `(vec1, len2)` and `(vec2, len2)`.

```
void _fq_zech_vec_sub(fq_zech_struct * res, const
   fq_zech_struct * vec1, const fq_zech_struct * vec2,
   slong len2, const fq_zech_ctx_t ctx)
```

Sets `(res, len2)` to `(vec1, len2)` minus `(vec2, len2)`.

47.7 Scalar multiplication and division

```
void _fq_zech_vec_scalar_addmul_fq_zech(fq_zech_struct *
   vec1, const fq_zech_struct * vec2, slong len2, const
   fq_zech_t c, const fq_zech_ctx_t ctx)
```

Adds `(vec2, len2)` times c to `(vec1, len2)`, where c is a `fq_zech_t`.

```
void _fq_zech_vec_scalar_submul_fq_zech(fq_zech_struct *
   vec1, const fq_zech_struct * vec2, slong len2, const
   fq_zech_t c, const fq_zech_ctx_t ctx)
```

Subtracts `(vec2, len2)` times c from `(vec1, len2)`, where c is a `fq_zech_t`.

47.8 Dot products

```
void _fq_zech_vec_dot(fq_zech_t res, const fq_zech_struct *
   vec1, const fq_zech_struct * vec2, slong len2, const
   fq_zech_ctx_t ctx)
```

Sets `res` to the dot product of (`vec1`, `len`) and (`vec2`, `len`).

§48. fq_zech_mat

Matrices over finite fields in Zech logarithm representation

48.1 Memory management

```
void fq_zech_mat_init(fq_zech_mat_t mat, slong rows, slong
    cols, const fq_zech_ctx_t ctx)
```

Initialises `mat` to a `rows`-by-`cols` matrix with coefficients in $\mathbb{F}_q$ given by `ctx`. All elements are set to zero.

```
void fq_zech_mat_init_set(fq_zech_mat_t mat, fq_zech_mat_t
    src, const fq_zech_ctx_t ctx)
```

Initialises `mat` and sets its dimensions and elements to those of `src`.

```
void fq_zech_mat_clear(fq_zech_mat_t mat, const
    fq_zech_ctx_t ctx)
```

Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an `fq_zech_mat_t` object.

```
void fq_zech_mat_set(fq_zech_mat_t mat, fq_zech_mat_t src,
    const fq_zech_ctx_t ctx)
```

Sets `mat` to a copy of `src`. It is assumed that `mat` and `src` have identical dimensions.

48.2 Basic properties and manipulation

```
fq_zech_struct * fq_zech_mat_entry(fq_zech_mat_t mat, slong
    i, slong j)
```

Directly accesses the entry in `mat` in row i and column j, indexed from zero. No bounds checking is performed.

```
fq_zech_struct * fq_zech_mat_entry_set(fq_zech_mat_t mat,
    slong i, slong j, fq_zech_t x, const fq_zech_ctx_t ctx)
```

Sets the entry in `mat` in row i and column j to `x`.

```
slong fq_zech_mat_nrows(fq_zech_mat_t mat, const
   fq_zech_ctx_t ctx)
```

Returns the number of rows in `mat`.

```
slong fq_zech_mat_ncols(fq_zech_mat_t mat, const
   fq_zech_ctx_t ctx)
```

Returns the number of columns in `mat`.

```
void fq_zech_mat_swap(fq_zech_mat_t mat1, fq_zech_mat_t
   mat2, const fq_zech_ctx_t ctx)
```

Swaps two matrices. The dimensions of `mat1` and `mat2` are allowed to be different.

```
void fq_zech_mat_zero(fq_zech_mat_t mat, const
   fq_zech_ctx_t ctx)
```

Sets all entries of `mat` to 0.

48.3 Printing

```
void fq_zech_mat_print_pretty(const fq_zech_mat_t mat,
   const fq_zech_ctx_t ctx)
```

Pretty-prints `mat` to `stdout`. A header is printed followed by the rows enclosed in brackets.

```
int fq_zech_mat_fprint_pretty(FILE * file, const
   fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
```

Pretty-prints `mat` to `file`. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
void fq_zech_mat_print(const fq_zech_mat_t mat, const
   fq_zech_ctx_t ctx)
```

Prints `mat` to `stdout`. A header is printed followed by the rows enclosed in brackets.

```
int fq_zech_mat_fprint(FILE * file, const fq_zech_mat_t
   mat, const fq_zech_ctx_t ctx)
```

Prints `mat` to `file`. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

48.4 Window

```
void fq_zech_mat_window_init(fq_zech_mat_t window, const
   fq_zech_mat_t mat, slong r1, slong c1, slong r2, slong
   c2, const fq_zech_ctx_t ctx)
```

Initializes the matrix `window` to be an `r2 - r1` by `c2 - c1` submatrix of `mat` whose `(0,0)` entry is the `(r1, c1)` entry of `mat`. The memory for the elements of `window` is shared with `mat`.

```
void fq_zech_mat_window_clear(fq_zech_mat_t window, const
   fq_zech_ctx_t ctx)
```

Clears the matrix `window` and releases any memory that it uses. Note that the memory to the underlying matrix that `window` points to is not freed.

48.5 Random matrix generation

```
void fq_zech_mat_randtest(fq_zech_mat_t mat, flint_rand_t
   state, const fq_zech_ctx_t ctx)
```

Sets the elements of `mat` to random elements of $\mathbb{F}_q$, given by `ctx`.

```
int fq_zech_mat_randpermdiag(fq_zech_mat_t mat,
   fq_zech_struct * diag, slong n, flint_rand_t state,
   const fq_zech_ctx_t ctx)
```

Sets `mat` to a random permutation of the diagonal matrix with n leading entries given by the vector `diag`. It is assumed that the main diagonal of `mat` has room for at least n entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.

```
void fq_zech_mat_randrank(fq_zech_mat_t mat, slong rank,
   flint_rand_t state, const fq_zech_ctx_t ctx)
```

Sets `mat` to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random elements of $\mathbb{F}_q$.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling `fq_zech_mat_randops()`.

```
void fq_zech_mat_randops(fq_zech_mat_t mat, slong count,
   flint_rand_t state, const fq_zech_ctx_t ctx)
```

Randomises `mat` by performing elementary row or column operations. More precisely, at most `count` random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

```
void fq_zech_mat_randtril(fq_zech_mat_t mat, flint_rand_t
   state, int unit, const fq_zech_ctx_t ctx)
```

Sets `mat` to a random lower triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

```
void fq_zech_mat_randtriu(fq_zech_mat_t mat, flint_rand_t
   state, int unit, x      const fq_zech_ctx_t ctx)
```

Sets `mat` to a random upper triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

48.6 Comparison

```
int fq_zech_mat_equal(fq_zech_mat_t mat1, fq_zech_mat_t
   mat2, const fq_zech_ctx_t ctx)
```

Returns nonzero if mat1 and mat2 have the same dimensions and elements, and zero otherwise.

```
int fq_zech_mat_is_zero(const fq_zech_mat_t mat, const
   fq_zech_ctx_t ctx)
```

Returns a non-zero value if all entries `mat` are zero, and otherwise returns zero.

```
int fq_zech_mat_is_empty(const fq_zech_mat_t mat, const
    fq_zech_ctx_t ctx)
```

Returns a non-zero value if the number of rows or the number of columns in `mat` is zero, and otherwise returns zero.

```
int fq_zech_mat_is_square(const fq_zech_mat_t mat, const
    fq_zech_ctx_t ctx)
```

Returns a non-zero value if the number of rows is equal to the number of columns in `mat`, and otherwise returns zero.

48.7 Addition and subtraction

```
void fq_zech_mat_add(fq_zech_mat_t C, const fq_zech_mat_t
    A, const fq_zech_mat_t B,  const fq_zech_ctx_t ctx)
```

Computes $C = A + B$. Dimensions must be identical.

```
void fq_zech_mat_sub(fq_zech_mat_t C, const fq_zech_mat_t
    A, const fq_zech_mat_t B, const fq_zech_ctx_t ctx)
```

Computes $C = A - B$. Dimensions must be identical.

```
void fq_zech_mat_neg(fq_zech_mat_t A, const fq_zech_mat_t
    B, const fq_zech_ctx_t ctx)
```

Sets $B = -A$. Dimensions must be identical.

48.8 Matrix multiplication

```
void fq_zech_mat_mul(fq_zech_mat_t C, const fq_zech_mat_t
    A, const fq_zech_mat_t B,  const fq_zech_ctx_t ctx)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. This function automatically chooses between classical and KS multiplication.

```
void fq_zech_mat_mul_classical(fq_zech_mat_t C, const
    fq_zech_mat_t A, const fq_zech_mat_t B, const
    fq_zech_ctx_t ctx)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Uses classical matrix multiplication.

```
void fq_zech_mat_mul_KS(fq_zech_mat_t C, const
    fq_zech_mat_t A, const fq_zech_mat_t B, const
    fq_zech_ctx_t ctx)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Uses Kronecker substitution to perform the multiplication over the integers.

```
void fq_zech_mat_submul(fq_zech_mat_t D, const
    fq_zech_mat_t C, const fq_zech_mat_t A, const
    fq_zech_mat_t B, const fq_zech_ctx_t ctx)
```

Sets $D = C + AB$. C and D may be aliased with each other but not with A or B.

48.9 LU decomposition

```
slong fq_zech_mat_lu(slong * P, fq_zech_mat_t A, int
    rank_check, const fq_zech_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A.

If A is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r, U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and return 0 if A is detected to be rank-deficient.

This function calls `fq_zech_mat_lu_recursive`.

```
slong fq_zech_mat_lu_classical(slong * P, fq_zech_mat_t A,
    int rank_check, const fq_zech_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A. The behavior of this function is identical to that of `fq_zech_mat_lu`. Uses Gaussian elimination.

```
slong fq_zech_mat_lu_recursive(slong * P, fq_zech_mat_t A,
    int rank_check, const fq_zech_ctx_t ctx)
```

Computes a generalised LU decomposition $LU = PA$ of a given matrix A, returning the rank of A. The behavior of this function is identical to that of `fq_zech_mat_lu`. Uses recursive block decomposition, switching to classical Gaussian elimination for sufficiently small blocks.

48.10 Reduced row echelon form

```
slong fq_zech_mat_rref(fq_zech_mat_t A, const fq_zech_ctx_t
    ctx)
```

Puts A in reduced row echelon form and returns the rank of A.

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

48.11 Triangular solving

```
void fq_zech_mat_solve_tril(fq_zech_mat_t X, const
    fq_zech_mat_t L, const fq_zech_mat_t B, int unit, const
    fq_zech_ctx_t ctx)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

```
void fq_zech_mat_solve_tril_classical(fq_zech_mat_t X,
    const fq_zech_mat_t L, const fq_zech_mat_t B, int unit,
    const fq_zech_ctx_t ctx)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

```
void fq_zech_mat_solve_tril_recursive(fq_zech_mat_t X,
    const fq_zech_mat_t L, const fq_zech_mat_t B, int unit,
    const fq_zech_ctx_t ctx)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` $= 1$, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$\begin{pmatrix} A & 0 \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}X \\ D^{-1}(Y - CA^{-1}X) \end{pmatrix}$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

```
void fq_zech_mat_solve_triu(fq_zech_mat_t X, const
    fq_zech_mat_t U, const fq_zech_mat_t B, int unit, const
    fq_zech_ctx_t ctx)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

```
void fq_zech_mat_solve_triu_classical(fq_zech_mat_t X,
    const fq_zech_mat_t U, const fq_zech_mat_t B, int unit,
    const fq_zech_ctx_t ctx)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

```
void fq_zech_mat_solve_triu_recursive(fq_zech_mat_t X,
    const fq_zech_mat_t U, const fq_zech_mat_t B, int unit,
    const fq_zech_ctx_t ctx)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` $= 1$, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$\begin{pmatrix} A & B \\ 0 & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}(X - BD^{-1}Y) \\ D^{-1}Y \end{pmatrix}$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

§49. fq_zech_poly

Polynomials over finite fields in Zech logarithm representation

We represent a polynomial in $\mathbf{F}_q[X]$ as a `struct` which includes an array `coeffs` with the coefficients, as well as the length `length` and the number `alloc` of coefficients for which memory has been allocated.

As a data structure, we call this polynomial *normalised* if the top coefficient is non-zero.

Unless otherwise stated here, all functions that deal with polynomials assume that the $\mathbf{F}_q$ context of said polynomials are compatible, i.e., it assumes that the fields are generated by the same polynomial.

49.1 Memory management

```
void fq_zech_poly_init(fq_zech_poly_t poly, const
   fq_zech_ctx_t ctx)
```

Initialises `poly` for use, with context ctx, and setting its length to zero. A corresponding call to `fq_zech_poly_clear()` must be made after finishing with the `fq_zech_poly_t` to free the memory used by the polynomial.

```
void fq_zech_poly_init2(fq_zech_poly_t poly, slong alloc,
   const fq_zech_ctx_t ctx)
```

Initialises `poly` with space for at least `alloc` coefficients and sets the length to zero. The allocated coefficients are all set to zero. A corresponding call to `fq_zech_poly_clear()` must be made after finishing with the `fq_zech_poly_t` to free the memory used by the polynomial.

```
void fq_zech_poly_realloc(fq_zech_poly_t poly, slong alloc,
   const fq_zech_ctx_t ctx)
```

Reallocates the given polynomial to have space for `alloc` coefficients. If `alloc` is zero the polynomial is cleared and then reinitialised. If the current length is greater than `alloc` the polynomial is first truncated to length `alloc`.

```
void fq_zech_poly_fit_length(fq_zech_poly_t poly, slong
   len, const fq_zech_ctx_t ctx)
```

If `len` is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least `len` coefficients. No data is lost when calling this function.

The function efficiently deals with the case where `fit_length` is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

```
void _fq_zech_poly_set_length(fq_zech_poly_t poly, slong
   newlen, const fq_zech_ctx_t ctx)
```

Sets the coefficients of `poly` beyond `len` to zero and sets the length of `poly` to `len`.

```
void fq_zech_poly_clear(fq_zech_poly_t poly, const
   fq_zech_ctx_t ctx)
```

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

```
void _fq_zech_poly_normalise(fq_zech_poly_t poly, const
   fq_zech_ctx_t ctx)
```

Sets the length of `poly` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

```
void _fq_zech_poly_normalise2(fq_zech_struct *poly, slong
   *length, const fq_zech_ctx_t ctx)
```

Sets the length `length` of `(poly,length)` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

```
void fq_zech_poly_truncate(fq_zech_poly_t poly, slong
   newlen, const fq_zech_ctx_t ctx)
```

Truncates the polynomial to length at most n.

```
void _fq_zech_poly_reverse(fq_zech_struct* output, const
   fq_zech_struct* input, slong len, slong m, const
   fq_zech_ctx_t ctx)
```

Sets `output` to the reverse of `input`, which is of length `len`, but thinking of it as a polynomial of length `m`, notionally zero-padded if necessary. The length `m` must be non-negative, but there are no other restrictions. The polynomial `output` must have space for `m` coefficients.

```
void fq_zech_poly_reverse(fq_zech_poly_t output, const
   fq_zech_poly_t input, slong m, const fq_zech_ctx_t ctx)
```

Sets `output` to the reverse of `input`, thinking of it as a polynomial of length `m`, notionally zero-padded if necessary). The length `m` must be non-negative, but there are no other restrictions. The output polynomial will be set to length `m` and then normalised.

49.2 Polynomial parameters

```
long fq_zech_poly_degree(fq_zech_poly_t poly, const
   fq_zech_ctx_t ctx)
```

Returns the degree of the polynomial `poly`.

```
long fq_zech_poly_length(fq_zech_poly_t poly, const
   fq_zech_ctx_t ctx)
```

Returns the length of the polynomial `poly`.

```
fq_zech_struct * fq_zech_poly_lead(const fq_zech_poly_t
   poly, const fq_zech_ctx_t ctx)
```

Returns a pointer to the leading coefficient of `poly`, or `NULL` if `poly` is the zero polynomial.

49.3 Randomisation

```
void fq_zech_poly_randtest(fq_zech_poly_t f, flint_rand_t
   state, slong len, const fq_zech_ctx_t ctx)
```

Sets f to a random polynomial of length at most `len` with entries in the field described by `ctx`.

```
void fq_zech_poly_randtest_not_zero(fq_zech_poly_t f,
   flint_rand_t state, slong len, const fq_zech_ctx_t ctx)
```

Same as `fq_zech_poly_randtest` but guarantees that the polynomial is not zero.

```
void fq_zech_poly_randtest_monic(fq_zech_poly_t f,
   flint_rand_t state, slong len, const fq_zech_ctx_t ctx)
```

Sets f to a random monic polynomial of length `len` with entries in the field described by `ctx`.

```
void fq_zech_poly_randtest_irreducible(fq_zech_poly_t f,
   flint_rand_t state, slong len, const fq_zech_ctx_t ctx)
```

Sets f to a random monic, irreducible polynomial of length `len` with entries in the field described by `ctx`.

49.4 Assignment and basic manipulation

```
void _fq_zech_poly_set(fq_zech_struct *rop, const
   fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)
```

Sets `(rop, len)` to `(op, len)`.

```
void fq_zech_poly_set(fq_zech_poly_t poly1, const
   fq_zech_poly_t poly2, const fq_zech_ctx_t ctx)
```

Sets the polynomial `poly1` to the polynomial `poly2`.

```
void fq_zech_poly_set_fq_zech(fq_zech_poly_t poly, const
   fq_zech_t c, const fq_zech_ctx_t ctx)
```

Sets the polynomial `poly` to `c`.

```
void fq_zech_poly_swap(fq_zech_poly_t op1, fq_zech_poly_t
   op2, const fq_zech_ctx_t ctx)
```

Swaps the two polynomials `op1` and `op2`.

```
void _fq_zech_poly_zero(fq_zech_struct *rop, slong len,
   const fq_zech_ctx_t ctx)
```

Sets `(rop, len)` to the zero polynomial.

```
void fq_zech_poly_zero(fq_zech_poly_t poly, const
   fq_zech_ctx_t ctx)
```

Sets `poly` to the zero polynomial.

```
void void fq_zech_poly_one(fq_zech_poly_t poly, const
   fq_zech_ctx_t ctx)
```

Sets `poly` to the constant polynomial 1.

```
void void fq_zech_poly_gen(fq_zech_poly_t poly, const
   fq_zech_ctx_t ctx)
```

Sets `poly` to the polynomial x.

```
void fq_zech_poly_make_monic(fq_zech_poly_t rop, const
   fq_zech_poly_t op, const fq_zech_ctx_t ctx)
```

Sets `rop` to `op`, normed to have leading coefficient 1.

```
void _fq_zech_poly_make_monic(fq_zech_struct *rop, const
   fq_zech_struct *op, slong length, const fq_zech_ctx_t
   ctx)
```

Sets `rop` to `(op,length)`, normed to have leading coefficient 1. Assumes that `rop` has enough space for the polynomial, assumes that `op` is not zero (and thus has an invertible leading coefficient).

49.5 Getting and setting coefficients

```
void fq_zech_poly_get_coeff(fq_zech_t x, const
   fq_zech_poly_t poly, slong n, const fq_zech_ctx_t ctx)
```

Sets x to the coefficient of X^n in `poly`.

```
void fq_zech_poly_set_coeff(fq_zech_poly_t poly, slong n,
   const fq_zech_t x, const fq_zech_ctx_t ctx)
```

Sets the coefficient of X^n in `poly` to x.

```
void fq_zech_poly_set_coeff_fmpz(fq_zech_poly_t poly, slong
   n, const fmpz_t x, const fq_zech_ctx_t ctx)
```

Sets the coefficient of X^n in the polynomial to x, assuming $n \geq 0$.

49.6 Comparison

```
int fq_zech_poly_equal(const fq_zech_poly_t poly1, const
   fq_zech_poly_t poly2, const fq_zech_ctx_t ctx)
```

Returns whether the two polynomials `poly1` and `poly2` are equal.

```
int fq_zech_poly_is_zero(const fq_zech_poly_t poly, const
   fq_zech_ctx_t ctx)
```

Returns whether the polynomial `poly` is the zero polynomial.

```
int fq_zech_poly_is_one(const fq_zech_poly_t op)
```

Returns whether the polynomial `poly` is equal to the constant polynomial 1.

```
int fq_zech_poly_is_gen(const fq_zech_poly_t op, const
   fq_zech_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal to the polynomial x.

```
int fq_zech_poly_is_unit(const fq_zech_poly_t op, const
    fq_zech_ctx_t ctx)
```

Returns whether the polynomial `poly` is a unit in the polynomial ring $\mathbf{F}_q[X]$, i.e. if it has degree 0 and is non-zero.

```
int fq_zech_poly_equal_fq_zech(const fq_zech_poly_t poly,
    const fq_zech_t c, const fq_zech_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal the (constant) $\mathbf{F}_q$ element `c`

49.7 Addition and subtraction

```
void _fq_zech_poly_add(fq_zech_struct *res, const
    fq_zech_struct *poly1, slong len1, const fq_zech_struct
    *poly2, slong len2, const fq_zech_ctx_t ctx)
```

Sets `res` to the sum of `(poly1,len1)` and `(poly2,len2)`.

```
void fq_zech_poly_add(fq_zech_poly_t res, const
    fq_zech_poly_t poly1, const fq_zech_poly_t poly2, const
    fq_zech_ctx_t ctx)
```

Sets `res` to the sum of `poly1` and `poly2`.

```
void _fq_zech_poly_sub(fq_zech_struct *res, const
    fq_zech_struct *poly1, slong len1, const fq_zech_struct
    *poly2, slong len2, const fq_zech_ctx_t ctx)
```

Sets `res` to the difference of `(poly1,len1)` and `(poly2,len2)`.

```
void fq_zech_poly_sub(fq_zech_poly_t res, const
    fq_zech_poly_t poly1, const fq_zech_poly_t poly2, const
    fq_zech_ctx_t ctx)
```

Sets `res` to the difference of `poly1` and `poly2`.

```
void _fq_zech_poly_neg(fq_zech_struct *rop, const
    fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)
```

Sets `res` to the additive inverse of `(poly,len)`.

```
void fq_zech_poly_neg(fq_zech_poly_t res, const
    fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
```

Sets `res` to the additive inverse of `poly`.

49.8 Scalar multiplication and division

```
void _fq_zech_poly_scalar_mul_fq_zech(fq_zech_struct *rop,
    const fq_zech_struct *op, slong len, const fq_zech_t x,
    const fq_zech_ctx_t ctx)
```

Sets `(rop,len)` to the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`.

```
void fq_zech_poly_scalar_mul_fq_zech(fq_zech_poly_t rop,
    const fq_zech_poly_t op, const fq_zech_t x, const
    fq_zech_ctx_t ctx)
```

Sets `(rop,len)` to the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`.

```
void _fq_zech_poly_scalar_addmul_fq_zech(fq_zech_struct
   *rop, const fq_zech_struct *op, slong len, const
   fq_zech_t x, const fq_zech_ctx_t ctx)
```

Adds to `(rop,len)` the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`. In particular, assumes the same length for `op` and `rop`.

```
void fq_zech_poly_scalar_addmul_fq_zech(fq_zech_poly_t rop,
   const fq_zech_poly_t op, const fq_zech_t x, const
   fq_zech_ctx_t ctx)
```

Adds to `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.

```
void _fq_zech_poly_scalar_submul_fq_zech(fq_zech_struct
   *rop, const fq_zech_struct *op, slong len, const
   fq_zech_t x, const fq_zech_ctx_t ctx)
```

Substracts from `(rop,len)` the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`. In particular, assumes the same length for `op` and `rop`.

```
void fq_zech_poly_scalar_submul_fq_zech(fq_zech_poly_t rop,
   const fq_zech_poly_t op, const fq_zech_t x, const
   fq_zech_ctx_t ctx)
```

Substracts from `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.

49.9 Multiplication

```
void _fq_zech_poly_mul_classical(fq_zech_struct *rop, const
   fq_zech_struct *op1, slong len1, const fq_zech_struct
   *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, assuming that `len1` is at least `len2` and neither is zero.

Permits zero padding. Does not support aliasing of `rop` with either `op1` or `op2`.

```
void fq_zech_poly_mul_classical(fq_zech_poly_t rop, const
   fq_zech_poly_t op1, const fq_zech_poly_t op2, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2` using classical polynomial multiplication.

```
void _fq_zech_poly_mul_reorder(fq_zech_struct *rop, const
   fq_zech_struct *op1, slong len1, const fq_zech_struct
   *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, assuming that `len1` and `len2` are non-zero.

Permits zero padding. Supports aliasing.

```
void fq_zech_poly_mul_reorder(fq_zech_poly_t rop, const
   fq_zech_poly_t op1, const fq_zech_poly_t op2, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`, reordering the two indeterminates X and Y when viewing the polynomials as elements of $\mathbf{F}_p[X, Y]$.

Suppose $\mathbf{F}_q = \mathbf{F}_p[X]/(f(X))$ and recall that elements of $\mathbf{F}_q$ are internally represented by elements of type `fmpz_poly`. For small degree extensions but polynomials in $\mathbf{F}_q[Y]$ of large degree n, we change the representation to

$$\begin{aligned} g(Y) &= \sum_{i=0}^{n} a_i(X) Y^i \\ &= \sum_{j=0}^{d} \sum_{i=0}^{n} \text{Coeff}(a_i(X), j) Y^i. \end{aligned}$$

This allows us to use a poor algorithm (such as classical multiplication) in the X-direction and leverage the existing fast integer multiplication routines in the Y-direction where the polynomial degree n is large.

```
void _fq_zech_poly_mul_KS(fq_zech_struct *rop, const
   fq_zech_struct *op1, slong len1, const fq_zech_struct
   *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`.

Permits zero padding and places no assumptions on the lengths `len1` and `len2`. Supports aliasing.

```
void fq_zech_poly_mul_KS(fq_zech_poly_t rop, const
   fq_zech_poly_t op1, const fq_zech_poly_t op2, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2` using Kronecker substitution, that is, by encoding each coefficient in $\mathbf{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

```
void _fq_zech_poly_mul(fq_zech_struct *rop, const
   fq_zech_struct *op1, slong len1, const fq_zech_struct
   *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

```
void fq_zech_poly_mul(fq_zech_poly_t rop, const
   fq_zech_poly_t op1, const fq_zech_poly_t op2, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`, choosing an appropriate algorithm.

```
void _fq_zech_poly_mullow_classical(fq_zech_struct *rop,
   const fq_zech_struct *op1, slong len1, const
   fq_zech_struct *op2, slong len2, slong n, const
   fq_zech_ctx_t ctx)
```

Sets `(res, n)` to the first n coefficients of `(poly1, len1)` multiplied by `(poly2, len2)`.

Assumes `0 < n <= len1 + len2 - 1`. Assumes neither `len1` nor `len2` is zero.

```
void fq_zech_poly_mullow_classical(fq_zech_poly_t rop,
   const fq_zech_poly_t op1, const fq_zech_poly_t op2,
   slong n, const fq_zech_ctx_t ctx)
```

Sets `res` to the product of `poly1` and `poly2`, computed using the classical or schoolbook method.

```
void _fq_zech_poly_mullow_KS(fq_zech_struct *rop, const
   fq_zech_struct *op1, slong len1, const fq_zech_struct
   *op2, slong len2, slong n, const fq_zech_ctx_t ctx)
```

Sets `(res, n)` to the lowest n coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `len1` and `len2` are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between `res`, `poly1` and `poly2`.

```
void fq_zech_poly_mullow_KS(fq_zech_poly_t rop, const
   fq_zech_poly_t op1, const fq_zech_poly_t op2, slong n,
   const fq_zech_ctx_t ctx)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _fq_zech_poly_mullow(fq_zech_struct *rop, const
   fq_zech_struct *op1, slong len1, const fq_zech_struct
   *op2, slong len2, slong n, const fq_zech_ctx_t ctx)
```

Sets `(res, n)` to the lowest n coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes `0 < n <= len1 + len2 - 1`. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

```
void fq_zech_poly_mullow(fq_zech_poly_t rop, const
   fq_zech_poly_t op1, const fq_zech_poly_t op2, slong n,
   const fq_zech_ctx_t ctx)
```

Sets `res` to the lowest n coefficients of the product of `poly1` and `poly2`.

```
void _fq_zech_poly_mulmod(fq_zech_struct* res, const
   fq_zech_struct* poly1, slong len1, const fq_zech_struct*
   poly2, slong len2, const fq_zech_struct* f, slong lenf,
   const fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_fq_zech_poly_mul` instead.

Aliasing of `f` and `res` is not permitted.

```
void fq_zech_poly_mulmod(fq_zech_poly_t res,const
   fq_zech_poly_t poly1, const fq_zech_poly_t poly2, const
   fq_zech_poly_t f, const fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

```
void _fq_zech_poly_mulmod_preinv(fq_zech_struct* res, const
   fq_zech_struct* poly1, slong len1, const fq_zech_struct*
   poly2, slong len2, const fq_zech_struct* f, slong lenf,
   const fq_zech_struct* finv, slong lenfinv, const
   fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `finv` is the inverse of the reverse of `f` mod `x^lenf`. It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_fq_zech_poly_mul` instead.

Aliasing of `f` or `finv` and `res` is not permitted.

```
void fq_zech_poly_mulmod_preinv(fq_zech_poly_t res, const
   fq_zech_poly_t poly1, const fq_zech_poly_t poly2, const
   fq_zech_poly_t f, const fq_zech_poly_t finv, const
   fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`. `finv` is the inverse of the reverse of `f`.

49.10 Squaring

```
void _fq_zech_poly_sqr_classical(fq_zech_struct *rop, const
   fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `(op,len)` is not zero and using classical polynomial multiplication.

Permits zero padding. Does not support aliasing of `rop` with either `op1` or `op2`.

```
void fq_zech_poly_sqr_classical(fq_zech_poly_t rop, const
   fq_zech_poly_t op, const fq_zech_ctx_t ctx)
```

Sets `rop` to the square of `op` using classical polynomial multiplication.

```
void _fq_zech_poly_sqr_KS(fq_zech_struct *rop, const
   fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`.

Permits zero padding and places no assumptions on the lengths `len1` and `len2`. Supports aliasing.

```
void fq_zech_poly_sqr_KS(fq_zech_poly_t rop, const
   fq_zech_poly_t op, const fq_zech_ctx_t ctx)
```

Sets `rop` to the square `op` using Kronecker substitution, that is, by encoding each coefficient in $\mathbf{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

```
void _fq_zech_poly_sqr(fq_zech_struct *rop, const
   fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)
```

Sets `(rop, 2* len - 1)` to the square of `(op, len)`, choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

```
void fq_zech_poly_sqr(fq_zech_poly_t rop, const
   fq_zech_poly_t op, const fq_zech_ctx_t ctx)
```

Sets `rop` to the square of `op`, choosing an appropriate algorithm.

49.11 Powering

```
void _fq_zech_poly_pow(fq_zech_struct *rop, const
   fq_zech_struct *op, slong len, ulong e, const
   fq_zech_ctx_t ctx)
```

Sets `res = poly^e`, assuming that `e, len > 0` and that `res` has space for `e*(len - 1)+ 1` coefficients. Does not support aliasing.

```
void fq_zech_poly_pow(fq_zech_poly_t rop, const
   fq_zech_poly_t op, ulong e, const fq_zech_ctx_t ctx)
```

Computes `res = poly^e`. If e is zero, returns one, so that in particular `0^0 = 1`.

```
void _fq_zech_poly_powmod_ui_binexp(fq_zech_struct* res,
   const fq_zech_struct* poly, ulong e, const
   fq_zech_struct* f, slong lenf, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_zech_poly_powmod_ui_binexp(fq_zech_poly_t res,
   const fq_zech_poly_t poly, ulong e, const fq_zech_poly_t
   f, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void _fq_zech_poly_powmod_ui_binexp_preinv(fq_zech_struct*
   res, const fq_zech_struct* poly, ulong e, const
   fq_zech_struct* f, slong lenf, const fq_zech_struct*
   finv, slong lenfinv, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_zech_poly_powmod_ui_binexp_preinv(fq_zech_poly_t
   res, const fq_zech_poly_t poly, ulong e, const
   fq_zech_poly_t f, const fq_zech_poly_t finv, const
   fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fq_zech_poly_powmod_fmpz_binexp(fq_zech_struct* res,
   const fq_zech_struct* poly, fmpz_t e, const
   fq_zech_struct* f, slong lenf, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_zech_poly_powmod_fmpz_binexp(fq_zech_poly_t res,
    const fq_zech_poly_t poly, fmpz_t e, const
    fq_zech_poly_t f, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void
    _fq_zech_poly_powmod_fmpz_binexp_preinv(fq_zech_struct*
    res, const fq_zech_struct* poly, fmpz_t e, const
    fq_zech_struct* f, slong lenf, const fq_zech_struct*
    finv, slong lenfinv, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_zech_poly_powmod_fmpz_binexp_preinv(fq_zech_poly_t
    res, const fq_zech_poly_t poly, fmpz_t e, const
    fq_zech_poly_t f, const fq_zech_poly_t finv, const
    fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void
    _fq_zech_poly_powmod_fmpz_sliding_preinv(fq_zech_struct*
    res, const fq_zech_struct* poly, fmpz_t e, ulong k,
    const fq_zech_struct* f, slong lenf, const
    fq_zech_struct* finv, slong lenfinv, const fq_zech_ctx_t
    ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an "optimum" size will be selected automatically base on `e`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_zech_poly_powmod_fmpz_sliding_preinv(fq_zech_poly_t
    res, const fq_zech_poly_t poly, fmpz_t e, ulong k, const
    fq_zech_poly_t f, const fq_zech_poly_t finv, const
    fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an "optimum" size will be selected automatically base on `e`.

```
void _fq_zech_poly_powmod_x_fmpz_preinv(fq_zech_struct *
    res, const fmpz_t e, const fq_zech_struct * f, slong
    lenf, const fq_zech_struct * finv, slong lenfinv, const
    fq_zech_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 2`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_zech_poly_powmod_x_fmpz_preinv(fq_zech_poly_t res,
   const fmpz_t e, const fq_zech_poly_t f, const
   fq_zech_poly_t finv, const fq_zech_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

49.12 Shifting

```
void _fq_zech_poly_shift_left(fq_zech_struct *rop, const
   fq_zech_struct *op, slong len, slong n, const
   fq_zech_ctx_t ctx)
```

Sets `(res, len + n)` to `(poly, len)` shifted left by n coefficients.

Inserts zero coefficients at the lower end. Assumes that `len` and n are positive, and that `res` fits `len + n` elements. Supports aliasing between `res` and `poly`.

```
void fq_zech_poly_shift_left(fq_zech_poly_t rop, const
   fq_zech_poly_t op, slong n, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` shifted left by n coeffs. Zero coefficients are inserted.

```
void _fq_zech_poly_shift_right(fq_zech_struct *rop, const
   fq_zech_struct *op, slong len, slong n, const
   fq_zech_ctx_t ctx)
```

Sets `(res, len - n)` to `(poly, len)` shifted right by n coefficients.

Assumes that `len` and n are positive, that `len > n`, and that `res` fits `len - n` elements. Supports aliasing between `res` and `poly`, although in this case the top coefficients of `poly` are not set to zero.

```
void fq_zech_poly_shift_right(fq_zech_poly_t rop, const
   fq_zech_poly_t op, slong n, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` shifted right by n coefficients. If n is equal to or greater than the current length of `poly`, `res` is set to the zero polynomial.

49.13 Norms

```
long _fq_zech_poly_hamming_weight(const fq_zech_poly *op,
   slong len, const fq_zech_ctx_t ctx)
```

Returns the number of non-zero entries in `(op, len)`.

```
long fq_zech_poly_hamming_weight(const fq_zech_poly_t op,
   const fq_zech_ctx_t ctx)
```

Returns the number of non-zero entries in the polynomial `op`.

49.14 Euclidean division

```
void _fq_zech_poly_divrem_basecase(fq_zech_struct *Q,
   fq_zech_struct *R, const fq_zech_struct *A, slong lenA,
   const fq_zech_struct *B, slong lenB, const fq_zech_t
   invB, const fq_zech_ctx_t ctx)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ with $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is its inverse.

Assumes that $\operatorname{len}(A), \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fq_zech_poly_divrem_basecase(fq_zech_poly_t Q,
   fq_zech_poly_t R, const fq_zech_poly_t A, const
   fq_zech_poly_t B, const fq_zech_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ with $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible. This can be taken for granted the context is for a finite field, that is, when p is prime and $f(X)$ is irreducible.

```
void _fq_zech_poly_divrem(fq_zech_struct *Q, fq_zech_struct
   *R, const fq_zech_struct *A, slong lenA, const
   fq_zech_struct *B, slong lenB, const fq_zech_t invB,
   const fq_zech_ctx_t ctx)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ with $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is its inverse.

Assumes that $\operatorname{len}(A), \operatorname{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fq_zech_poly_divrem(fq_zech_poly_t Q, fq_zech_poly_t
   R, const fq_zech_poly_t A, const fq_zech_poly_t B, const
   fq_zech_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ with $0 \leq \operatorname{len}(R) < \operatorname{len}(B)$.

Assumes that the leading coefficient of B is invertible. This can be taken for granted the context is for a finite field, that is, when p is prime and $f(X)$ is irreducible.

```
void _fq_zech_poly_rem(fq_zech_struct *R, const
   fq_zech_struct *A, slong lenA, const fq_zech_struct *B,
   slong lenB, const fq_zech_t invB, const fq_zech_ctx_t
   ctx)
```

Sets `R` to the remainder of the division of `(A,lenA)` by `(B,lenB)`. Assumes that the leading coefficient of `(B,lenB)` is invertible and that `invB` is its inverse.

```
void fq_zech_poly_rem(fq_zech_poly_t R, const
   fq_zech_poly_t A, const fq_zech_poly_t B, const
   fq_zech_ctx_t ctx)
```

Sets `R` to the remainder of the division of `A` by `B` in the context described by `ctx`.

```
void _fq_zech_poly_div_basecase(fq_zech_struct *Q,
   fq_zech_struct *R, const fq_zech_struct *A, slong lenA,
   const fq_zech_struct *B, slong lenB, const fq_zech_t
   invB, const fq_zech_ctx_t ctx)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets `(Q, lenA - lenB + 1)`.

Requires temporary space `(R, lenA)`. If `R` is `NULL`, then the temporary space will be allocated. Allows aliasing only between A and R. Allows zero-padding in A but not in B. Assumes that the leading coefficient of B is a unit.

```
void fq_zech_poly_div_basecase(fq_zech_poly_t Q, const
   fq_zech_poly_t A, const fq_zech_poly_t B, const
   fq_zech_ctx_t ctx)
```

Notionally finds polynomials Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only `Q`. If $\text{len}(B) = 0$ an exception is raised.

```
void
   _fq_zech_poly_divrem_divconquer_recursive(fq_zech_struct
   * Q, fq_zech_struct * BQ, fq_zech_struct * W, const
   fq_zech_struct * A, const fq_zech_struct * B, slong
   lenB, const fq_zech_t invB, const fq_zech_ctx_t ctx)
```

Computes `(Q, lenB)`, `(BQ, 2 lenB - 1)` such that $BQ = B \times Q$ and $A = BQ + R$ where $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is the inverse.

Assumes $\text{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. Requires a temporary array `(W, 2 lenB - 1)`. No aliasing of input and output operands is allowed.

This function does not read the bottom $\text{len}(B) - 1$ coefficients from A, which means that they might not even need to exist in allocated memory.

```
void _fq_zech_poly_divrem_divconquer(fq_zech_struct * Q,
   fq_zech_struct * R, const fq_zech_struct * A, slong
   lenA, const fq_zech_struct * B, slong lenB, const
   fq_zech_t invB, const fq_zech_ctx_t ctx)
```

Computes `(Q, lenA - lenB + 1)`, `(R, lenA)` such that $A = BQ+R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is the inverse.

Assumes $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in `(A, lenA)`. No aliasing of input and output operands is allowed.

```
void fq_zech_poly_divrem_divconquer(fq_zech_poly_t Q,
   fq_zech_poly_t R, const fq_zech_poly_t A, const
   fq_zech_poly_t B, const fq_zech_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that B is non-zero and that the leading coefficient of B is invertible.

```
void _fq_zech_poly_div_newton_n_preinv(fq_zech_struct* Q,
   const fq_zech_struct* A, slong lenA, const
   fq_zech_struct* B, slong lenB, const fq_zech_struct*
   Binv, slong lenBinv, const fq_zech_struct ctx_t)
```

Notionally computes polynomials Q and R such that $A = BQ+R$ with $\text{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`, but return only Q.

We require that Q have space for `lenA - lenB + 1` coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void fq_zech_poly_div_newton_n_preinv(fq_zech_poly_t Q,
   const fq_zech_poly_t A, const fq_zech_poly_t B, const
   fq_zech_poly_t Binv, const fq_zech_ctx_t ctx)
```

Notionally computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$, but returns only Q.

We assume that the leading coefficient of B is a unit and that $Binv$ is the inverse of the reverse of B mod $x^{\operatorname{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void _fq_zech_poly_divrem_newton_n_preinv(fq_zech_struct*
   Q, fq_zech_struct* R, const fq_zech_struct* A, slong
   lenA, const fq_zech_struct* B, slong lenB, const
   fq_zech_struct* Binv, slong lenBinv, const fq_zech_ctx_t
   ctx)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R)$ less than `lenB`, where A is of length `lenA` and B is of length `lenB`. We require that Q have space for `lenA - lenB + 1` coefficients. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\operatorname{len}(B)}$. The algorithm used is to call `div_newton_preinv()` and then multiply out and compute the remainder.

```
void fq_zech_poly_divrem_newton_n_preinv(fq_zech_poly_t Q,
   fq_zech_poly_t R, const fq_zech_poly_t A, const
   fq_zech_poly_t B, const fq_zech_poly_t Binv, const
   fq_zech_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\operatorname{len}(R) < \operatorname{len}(B)$. We assume $Binv$ is the inverse of the reverse of B mod $x^{\operatorname{len}(B)}$.

It is required that the length of A is less than or equal to 2*the length of B - 2.

The algorithm used is to call `div_newton()` and then multiply out and compute the remainder.

```
void _fq_zech_poly_inv_series_newton(fq_zech_struct* Qinv,
   const fq_zech_struct* Q, slong n, const fq_zech_ctx_t
   ctx)
```

Given `Q` of length `n` whose constant coefficient is invertible modulo the given modulus, find a polynomial `Qinv` of length `n` such that `Q * Qinv` is `1` modulo x^n. Requires `n > 0`. This function can be viewed as inverting a power series via Newton iteration.

```
void fq_zech_poly_inv_series_newton(fq_zech_poly_t Qinv,
   const fq_zech_poly_t Q, slong n, const fq_zech_ctx_t ctx)
```

Given `Q` find `Qinv` such that `Q * Qinv` is `1` modulo x^n. The constant coefficient of `Q` must be invertible modulo the modulus of `Q`. An exception is raised if this is not the case or if `n = 0`. This function can be viewed as inverting a power series via Newton iteration.

49.15 Greatest common divisor

```
void fq_zech_poly_gcd(fq_zech_poly_t rop, const
   fq_zech_poly_t op1, const fq_zech_poly_t op2, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to the greatest common divisor of `op1` and `op2`, using the either the Euclidean or HGCD algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

```
long _fq_zech_poly_gcd(fq_zech_struct* G,const
   fq_zech_struct* A, slong lenA, const fq_zech_struct* B,
   slong lenB, const fq_zech_ctx_t ctx)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= lenB > 0` and sets G to it. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for `lenB` coefficients.

```
void fq_zech_poly_gcd_euclidean(fq_zech_poly_t rop, const
   fq_zech_poly_t op1, const fq_zech_poly_t op2, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to the greatest common divisor of `op1` and `op2`, using the Euclidean algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P. Except in the case where the GCD is zero, the GCD G is made monic.

```
long _fq_zech_poly_gcd_euclidean(fq_zech_struct* G, const
   fq_zech_struct* A, slong lenA, const fq_zech_struct* B,
   slong lenB, const fq_zech_ctx_t ctx)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA >= lenB > 0` and sets G to it. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for `lenB` coefficients.

49.16 Divisibility testing

```
int _fq_zech_poly_divides(fq_zech_struct *Q, const
   fq_zech_struct *A, slong lenA, const fq_zech_struct *B,
   slong lenB, const fq_zech_t invB, const fq_zech_ctx_t
   ctx)
```

Returns 1 if `(B, lenB)` divides `(A, lenA)` exactly and sets Q to the quotient, otherwise returns 0.

It is assumed that $\operatorname{len}(A) \geq \operatorname{len}(B) > 0$ and that Q has space for $\operatorname{len}(A) - \operatorname{len}(B) + 1$ coefficients.

Aliasing of Q with either of the inputs is not permitted.

This function is currently unoptimised and provided for convenience only.

```
int fq_zech_poly_divides(fq_zech_poly_t Q, const
   fq_zech_poly_t A, const fq_zech_poly_t B, const
   fq_zech_ctx_t ctx)
```

Returns 1 if B divides A exactly and sets Q to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.

49.17 Derivative

```
void _fq_zech_poly_derivative(fq_zech_struct *rop, const
   fq_zech_struct *op, slong len, const fq_zech_ctx_t ctx)
```

Sets `(rpoly, len - 1)` to the derivative of `(poly, len)`. Also handles the cases where `len` is 0 or 1 correctly. Supports aliasing of `rpoly` and `poly`.

```
void fq_zech_poly_derivative(fq_zech_poly_t rop, const
   fq_zech_poly_t op, const fq_zech_ctx_t ctx)
```

Sets `res` to the derivative of `poly`.

49.18 Evaluation

```
void _fq_zech_poly_evaluate_fq_zech(fq_zech_t rop, const
   fq_zech_struct *op, slong len, const fq_zech_t a, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to `(op, len)` evaluated at a.

Supports zero padding. There are no restrictions on `len`, that is, `len` is allowed to be zero, too.

```
void fq_zech_poly_evaluate_fq_zech(fq_zech_t rop, const
   fq_zech_poly_t f, const fq_zech_t a, const fq_zech_ctx_t
   ctx)
```

Sets `rop` to the value of $f(a)$.

As the coefficient ring $\mathbf{F}_q$ is finite, Horner's method is sufficient.

49.19 Composition

```
void _fq_zech_poly_compose_divconquer(fq_zech_struct *rop,
   const fq_zech_struct *op1, slong len1, const
   fq_zech_struct *op2, slong len2, const fq_zech_ctx_t ctx)
```

Computes the composition of `(op1, len1)` and `(op2, len2)` using a divide and conquer approach and places the result into `rop`, assuming `rop` can hold the output of length `(len1 - 1)* (len2 - 1)+ 1`.

Assumes `len1, len2 > 0`. Does not support aliasing between `rop` and any of `(op1, len1)` and `(op2, len2)`.

```
void fq_zech_poly_compose_divconquer(fq_zech_poly_t rop,
   const fq_zech_poly_t op1, const fq_zech_poly_t op2,
   const fq_zech_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be precise about the order of composition, denoting `rop`, `op1`, and `op2` by f, g, and h, respectively, sets $f(t) = g(h(t))$.

```
void _fq_zech_poly_compose_horner(fq_zech_struct *rop,
   const fq_zech_struct *op1, slong len1, const
   fq_zech_struct *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets `rop` to the composition of `(op1, len1)` and `(op2, len2)`.

Assumes that `rop` has space for `(len1-1)*(len2-1)+ 1` coefficients. Assumes that `op1` and `op2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fq_zech_poly_compose_horner(fq_zech_poly_t rop, const
   fq_zech_poly_t op1, const fq_zech_poly_t op2, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be more precise, denoting `rop`, `op1`, and `op2` by f, g, and h, sets $f(t) = g(h(t))$.

This implementation uses Horner's method.

```
void _fq_zech_poly_compose(fq_zech_struct *rop, const
   fq_zech_struct *op1, slong len1, const fq_zech_struct
   *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets `rop` to the composition of `(op1, len1)` and `(op2, len2)`.

Assumes that `rop` has space for `(len1-1)*(len2-1)+ 1` coefficients. Assumes that `op1` and `op2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fq_zech_poly_compose(fq_zech_poly_t rop, const
   fq_zech_poly_t op1, const fq_zech_poly_t op2, const
   fq_zech_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be precise about the order of composition, denoting `rop`, `op1`, and `op2` by f, g, and h, respectively, sets $f(t) = g(h(t))$.

```
void _fq_zech_poly_compose_mod_horner(fq_zech_struct * res,
   const fq_zech_struct * f, slong lenf, const
   fq_zech_struct * g, const fq_zech_struct * h, slong
   lenh, const fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_zech_poly_compose_mod_horner(fq_zech_poly_t res,
   const fq_zech_poly_t f, const fq_zech_poly_t g, const
   fq_zech_poly_t h, const fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero. The algorithm used is Horner's rule.

```
void _fq_zech_poly_compose_mod_horner_preinv(fq_zech_struct
   * res, const fq_zech_struct * f, slong lenf, const
   fq_zech_struct * g, const fq_zech_struct * h, slong
   lenh, const fq_zech_struct * hinv, slong lenhiv, const
   fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_zech_poly_compose_mod_horner_preinv(fq_zech_poly_t
   res, const fq_zech_poly_t f, const fq_zech_poly_t g,
   const fq_zech_poly_t h, const fq_zech_poly_t hinv, const
   fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The algorithm used is Horner's rule.

```
void _fq_zech_poly_compose_mod_brent_kung(fq_zech_struct *
   res, const fq_zech_struct * f, slong lenf, const
   fq_zech_struct * g, const fq_zech_struct * h, slong
   lenh, const fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_zech_poly_compose_mod_brent_kung(fq_zech_poly_t
   res, const fq_zech_poly_t f, const fq_zech_poly_t g,
   const fq_zech_poly_t h, const fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. The algorithm used is the Brent-Kung matrix algorithm.

```
void
   _fq_zech_poly_compose_mod_brent_kung_preinv(fq_zech_struct
   * res, const fq_zech_struct * f, slong lenf, const
   fq_zech_struct * g, const fq_zech_struct * h, slong
   lenh, const fq_zech_struct * hinv, slong lenhiv, const
   fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void
   fq_zech_poly_compose_mod_brent_kung_preinv(fq_zech_poly_t
   res, const fq_zech_poly_t f, const fq_zech_poly_t g,
   const fq_zech_poly_t h, const fq_zech_poly_t hinv, const
   fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The algorithm used is the Brent-Kung matrix algorithm.

```
void _fq_zech_poly_compose_mod(fq_zech_struct * res, const
   fq_zech_struct * f, slong lenf, const fq_zech_struct *
   g, const fq_zech_struct * h, slong lenh, const
   fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```
void fq_zech_poly_compose_mod(fq_zech_poly_t res, const
   fq_zech_poly_t f, const fq_zech_poly_t g, const
   fq_zech_poly_t h, const fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero.

```
void _fq_zech_poly_compose_mod_preinv(fq_zech_struct * res,
   const fq_zech_struct * f, slong lenf, const
   fq_zech_struct * g, const fq_zech_struct * h, slong
   lenh, const fq_zech_struct * hinv, slong lenhiv, const
   fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

```
void fq_zech_poly_compose_mod_preinv(fq_zech_poly_t res,
   const fq_zech_poly_t f, const fq_zech_poly_t g, const
   fq_zech_poly_t h, const fq_zech_poly_t hinv, const
   fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`.

```
void _fq_zech_poly_reduce_matrix_mod_poly (fq_zech_mat_t A,
   const fq_zech_mat_t B, const fq_zech_poly_t f, const
   fq_zech_ctx_t ctx)
```

Sets the ith row of `A` to the reduction of the ith row of B modulo f for $i = 1, \ldots, \sqrt{\deg(f)}$. We require B to be at least a $\sqrt{\deg(f)} \times \deg(f)$ matrix and f to be nonzero.

```
void _fq_zech_poly_precompute_matrix (fq_zech_mat_t A,
   const fq_zech_struct* f, const fq_zech_struct* g, slong
   leng, const fq_zech_struct* ginv, slong lenginv, const
   fq_zech_ctx_t ctx)
```

Sets the ith row of `A` to f^i modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of `g` and g to be nonzero.

```
void fq_zech_poly_precompute_matrix (fq_zech_mat_t A, const
   fq_zech_poly_t f, const fq_zech_poly_t g, const
   fq_zech_poly_t ginv, const fq_zech_ctx_t ctx)
```

Sets the ith row of `A` to f^i modulo g for $i = 1, \ldots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of `g`.

```
void
   _fq_zech_poly_compose_mod_brent_kung_precomp_preinv(fq_zech_struct*
   res, const fq_zech_struct* f, slong lenf, const
   fq_zech_mat_t A, const fq_zech_struct* h, slong h, const
   fq_zech_struct* hinv, slong lenhinv, const fq_zech_ctx_t
   ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that h is nonzero. We require that the ith row of A contains g^i for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of f is less than the length of h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void
    fq_zech_poly_compose_mod_brent_kung_precomp_preinv(fq_zech_poly_t
    res, const fq_zech_poly_t f, const fq_zech_mat_t A,
    const fq_zech_poly_t h, const fq_zech_poly_t hinv, const
    fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h. We require that the ith row of A contains g^i for $i = 1, \ldots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that f has smaller degree than h. Furthermore, we require `hinv` to be the inverse of the reverse of `h`. This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo h for fixed g and h.

49.20 Output

```
int _fq_zech_poly_fprint_pretty(FILE *file, const
    fq_zech_struct *poly, slong len, const char *x, const
    fq_zech_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_zech_poly_fprint_pretty(FILE * file, const
    fq_zech_poly_t poly, const char *x, const fq_zech_ctx_t
    ctx)
```

Prints the pretty representation of `poly` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_zech_poly_print_pretty(const fq_zech_struct *poly,
    slong len, const char *x, const fq_zech_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_zech_poly_print_pretty(const fq_zech_poly_t poly,
    const char *x, const fq_zech_ctx_t ctx)
```

Prints the pretty representation of `poly` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_zech_poly_fprint(FILE *file, const fq_zech_struct
    *poly, slong len, const fq_zech_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_zech_poly_fprint(FILE * file, const fq_zech_poly_t
    poly, const fq_zech_ctx_t ctx)
```

Prints the pretty representation of `poly` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_zech_poly_print(const fq_zech_struct *poly, slong
    len, const fq_zech_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_zech_poly_print(const fq_zech_poly_t poly, const
    fq_zech_ctx_t ctx)
```

Prints the representation of `poly` to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
char * _fq_zech_poly_get_str(const fq_zech_struct * poly,
    slong len, const fq_zech_ctx_t ctx)
```

Returns the plain FLINT string representation of the polynomial `(poly, len)`.

```
char * fq_zech_poly_get_str(const fq_zech_poly_t poly,
    const fq_zech_ctx_t ctx)
```

Returns the plain FLINT string representation of the polynomial `poly`.

```
char * _fq_zech_poly_get_str_pretty(const fq_zech_struct *
    poly, slong len, const char * x, const fq_zech_ctx_t ctx)
```

Returns a pretty representation of the polynomial `(poly, len)` using the null-terminated string `x` as the variable name.

```
char * fq_zech_poly_get_str_pretty(const fq_zech_poly_t
    poly, const char * x, const fq_zech_ctx_t ctx)
```

Returns a pretty representation of the polynomial `poly` using the null-terminated string `x` as the variable name

49.21 Inflation and deflation

```
void fq_zech_poly_inflate(fq_zech_poly_t result, const
    fq_zech_poly_t input, ulong inflation, const
    fq_zech_ctx_t ctx)
```

Sets `result` to the inflated polynomial $p(x^n)$ where p is given by `input` and n is given by `inflation`.

```
void fq_zech_poly_deflate(fq_zech_poly_t result, const
    fq_zech_poly_t input, ulong deflation, const
    fq_zech_ctx_t ctx)
```

Sets `result` to the deflated polynomial $p(x^{1/n})$ where p is given by `input` and n is given by `deflation`. Requires $n > 0$.

```
ulong fq_zech_poly_deflation(const fq_zech_poly_t input,
    const fq_zech_ctx_t ctx)
```

Returns the largest integer by which `input` can be deflated. As special cases, returns 0 if `input` is the zero polynomial and 1 of `input` is a constant polynomial.

§50. fq_zech_poly_factor

Factorisation of polynomials over finite fields in Zech logarithm representation

---

The `fq_zech__poly_factor` module is included automatically when one includes `fq_zech_poly.h`. One should not try to include `fq_zech_poly_factor.h` directly.

50.1 Memory Management

```
void fq_zech_poly_factor_init(fq_zech_poly_factor_t fac,
   const fq_zech_ctx_t ctx)
```

Initialises `fac` for use. An `fq_zech_poly_factor_t` represents a polynomial in factorised form as a product of polynomials with associated exponents.

```
void fq_zech_poly_factor_clear(fq_zech_poly_factor_t fac,
   const fq_zech_ctx_t ctx)
```

Frees all memory associated with `fac`.

```
void fq_zech_poly_factor_realloc(fq_zech_poly_factor_t fac,
   slong alloc, const fq_zech_ctx_t ctx)
```

Reallocates the factor structure to provide space for precisely `alloc` factors.

```
void fq_zech_poly_factor_fit_length(fq_zech_poly_factor_t
   fac, slong len, const fq_zech_ctx_t ctx)
```

Ensures that the factor structure has space for at least `len` factors. This functions takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

50.2 Basic Operations

```
void fq_zech_poly_factor_set(fq_zech_poly_factor_t res,
   const fq_zech_poly_factor_t fac, const fq_zech_ctx_t ctx)
```

Sets `res` to the same factorisation as `fac`.

```
void fq_zech_poly_factor_print_pretty(const
   fq_zech_poly_factor_t fac, const fq_zech_ctx_t ctx)
```

Pretty-prints the entries of `fac` to standard output.

```
void fq_zech_poly_factor_print(const fq_zech_poly_factor_t
   fac, const fq_zech_ctx_t ctx)
```

Prints the entries of `fac` to standard output.

```
void fq_zech_poly_factor_insert(fq_zech_poly_factor_t fac,
   const fq_zech_poly_t poly, slong exp, const
   fq_zech_ctx_t ctx)
```

Inserts the factor `poly` with multiplicity `exp` into the factorisation `fac`.

If `fac` already contains `poly`, then `exp` simply gets added to the exponent of the existing entry.

```
void fq_zech_poly_factor_concat(fq_zech_poly_factor_t res,
   const fq_zech_poly_factor_t fac, const fq_zech_ctx_t ctx)
```

Concatenates two factorisations.

This is equivalent to calling `fq_zech_poly_factor_insert()` repeatedly with the individual factors of `fac`.

Does not support aliasing between `res` and `fac`.

```
void fq_zech_poly_factor_pow(fq_zech_poly_factor_t fac,
   slong exp, const fq_zech_ctx_t ctx)
```

Raises `fac` to the power `exp`.

```
ulong fq_zech_poly_remove(fq_zech_poly_t f, const
   fq_zech_poly_t p, const fq_zech_ctx_t ctx)
```

Removes the highest possible power of `p` from `f` and returns the exponent.

50.3 Irreducibility Testing

```
int fq_zech_poly_is_irreducible(const fq_zech_poly_t f,
   const fq_zech_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0.

```
int fq_zech_poly_is_irreducible_ddf(const fq_zech_poly_t f,
   const fq_zech_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

```
int fq_zech_poly_is_irreducible_ben_or(const fq_zech_poly_t
   f, const fq_zech_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses Ben-Or's irreducibility test.

```
int _fq_zech_poly_is_squarefree(const fq_zech_struct * f,
   slong len, const fq_zech_ctx_t ctx)
```

Returns 1 if `(f, len)` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

```
int fq_zech_poly_is_squarefree(const fq_zech_poly_t f,
    const fq_zech_ctx_t ctx)
```

Returns 1 if `f` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

50.4 Factorisation

```
int fq_zech_poly_factor_equal_deg_prob(fq_zech_poly_t
    factor, flint_rand_t state, const fq_zech_poly_t pol,
    slong d, const fq_zech_ctx_t ctx)
```

Probabilistic equal degree factorisation of `pol` into irreducible factors of degree `d`. If it passes, a factor is placed in factor and 1 is returned, otherwise 0 is returned and the value of factor is undetermined.

Requires that `pol` be monic, non-constant and squarefree.

```
void fq_zech_poly_factor_equal_deg(fq_zech_poly_factor_t
    factors, const fq_zech_poly_t pol, slong d, const
    fq_zech_ctx_t ctx)
```

Assuming `pol` is a product of irreducible factors all of degree `d`, finds all those factors and places them in factors. Requires that `pol` be monic, non-constant and squarefree.

```
void fq_zech_poly_factor_distinct_deg(fq_zech_poly_factor_t
    res, const fq_zech_poly_t poly, slong * const *degs,
    const fq_zech_ctx_t ctx)
```

Factorises a monic non-constant squarefree polymnomial `poly` of degree n into factors $f[d]$ such that for $1 \leq d \leq n$ $f[d]$ is the product of the monic irreducible factors of `poly` of degree d. Factors are stored in `res`, assotiated powers of irreducible polynomials are stored in `degs` in the same order as factors.

Requires that `degs` have enough space for irreducible polynomials' powers (maximum space required is $n * sizeof(slong)$).

```
void fq_zech_poly_factor_squarefree(fq_zech_poly_factor_t
    res, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
```

Sets `res` to a squarefree factorization of `f`.

```
void fq_zech_poly_factor(fq_zech_poly_factor_t res, const
    fq_zech_poly_t f, const fq_zech_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors choosing the best algorithm for given modulo and degree. Choise is based on heuristic measurments.

```
void
    fq_zech_poly_factor_cantor_zassenhaus(fq_zech_poly_factor_t
    res, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the Cantor-Zassenhaus algorithm.

```
void
    fq_zech_poly_factor_kaltofen_shoup(fq_zech_poly_factor_t
    res, const fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a âĂIJbaby step/giant stepâĂİ strategy for the distinct-degree factorization step.

```
void fq_zech_poly_factor_berlekamp(fq_zech_poly_factor_t
   factors, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors using the Berlekamp algorithm.

```
void
   fq_zech_poly_factor_with_berlekamp(fq_zech_poly_factor_t
   res, fq_zech_t leading_coeff, const fq_zech_poly_t f,
   const fq_zech_ctx_t)
```

Factorises a general polynomial `f` into monic irreducible factors and sets `leading_coeff` to the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Berlekamp on all the individual square-free factors.

```
void
   fq_zech_poly_factor_with_cantor_zassenhaus(fq_zech_poly_factor_t
   res, fq_zech_t leading_coeff const fq_zech_poly_t f,
   const fq_zech_ctx_t ctx)
```

Factorises a general polynomial `f` into monic irreducible factors and sets `leading_coeff` to the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

```
void
   fq_zech_poly_factor_with_kaltofen_shoup(fq_zech_poly_factor_t
   res, fq_zech_t leading_coeff, const fq_zech_poly_t f,
   const fq_zech_ctx_t ctx)
```

Factorises a general polynomial `f` into monic irreducible factors and sets `leading_coeff` to the leading coefficient of `f`, or 0 if `f` is the zero polynomial.

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

```
void fq_zech_poly_iterated_frobenius_preinv(fq_zech_poly_t
   *rop, slong n, const fq_zech_poly_t v, const
   fq_zech_poly_t vinv, const fq_zech_ctx_t ctx)
```

Sets `rop[i]` to be $x^{q^i} mod v$ for $0 <= i < n$.

It is required that `vinv` is the inverse of the reverse of `v` mod `x^lenv`.

§51. padic

p-Adic numbers in $\mathbf{Q}_p$

51.1 Introduction

The `padic_t` data type represents elements of $\mathbf{Q}_p$ to precision N, stored in the form $x = p^v u$ with $u, v \in \mathbf{Z}$. Arithmetic operations can be carried out with respect to a context containing the prime number p and various pieces of pre-computed data.

Independent of the context, we consider a p-adic number $x = up^v$ to be in *canonical form* whenever either $p \nmid u$ or $u = v = 0$, and we say it is *reduced* if, in addition, for non-zero u, $u \in (0, p^{N-v})$.

We briefly describe the interface:

The functions in this module expect arguments of type `padic_t`, and each variable carries its own precision. The functions have an interface that is similar to the MPFR functions. In particular, they have the same semantics, specified as follows: Compute the requested operation exactly and then reduce the result to the precision of the output variable.

51.2 Data structures

A p-adic number of type `padic_t` comprises a unit u, a valuation v, and a precision N.

We provide the following macros to access these fields, so that code can be developed somewhat independently from the underlying data layout.

```
fmpz * padic_unit(const padic_t op)
```

Returns the unit part of the p-adic number as a FLINT integer, which can be used as an operand for the `fmpz` functions.

Note that this function is implemented as a macro, but it can only be used as an *rvalue*.

```
slong padic_val(const padic_t op)
```

Returns the valuation part of the p-adic number.

Note that this function is implemented as a macro and that the expression `padic_val(op)` can be used as both an *lvalue* and an *rvalue*.

```
slong padic_prec(const padic_t op)
```

Returns the precision of the p-adic number.

Note that this function is implemented as a macro and that the expression `padic_prec(op)` can be used as both an *lvalue* and an *rvalue*.

51.3 Context

A context object for p-adic arithmetic contains data pertinent to p-adic computations, but which we choose not to store with each element individually.

Currently, this includes the prime number p, its `double` inverse in case of word-sized primes, precomputed powers of p in the range given by `min` and `max`, and the printing mode.

```
void padic_ctx_init(padic_ctx_t ctx, const fmpz_t p, slong
   min, slong max, enum padic_print_mode mode)
```

Initialises the context `ctx` with the given data.

Assumes that p is a prime. This is not verified but the subsequent behaviour is undefined if p is a composite number.

Assumes that `min` and `max` are non-negative and that `min` is at most `max`, raising and `abort` signal otherwise.

Assumes that the printing mode is one of `PADIC_TERSE`, `PADIC_SERIES`, or `PADIC_VAL_UNIT`. Using the example $x = 7^{-1}12$ in $\mathbf{Q}_7$, these behave as follows:

- In `PADIC_TERSE` mode, a p-adic number is printed in the same way as a rational number, e.g. `12/7`.
- In `PADIC_SERIES` mode, a p-adic number is printed digit by digit, e.g. `5*7^-1 + 1`.
- In `PADIC_VAL_UNIT` mode, a p-adic number is printed showing the valuation and unit parts separately, e.g. `12*7^-1`.

```
void padic_ctx_clear(padic_ctx_t ctx);
```

Clears all memory that has been allocated as part of the context.

```
int _padic_ctx_pow_ui(fmpz_t rop, ulong e, const
   padic_ctx_t ctx)
```

Sets `rop` to p^e as efficiently as possible, where `rop` is expected to be an uninitialised `fmpz_t`.

If the return value is non-zero, it is the responsibility of the caller to clear the returned integer.

51.4 Memory management

```
void padic_init(padic_t rop)
```

Initialises the p-adic number with the precision set to `PADIC_DEFAULT_PREC`, which is defined as 20.

```
void padic_init2(padic_t rop, slong N)
```

Initialises the p-adic number `rop` with precision N.

```
void padic_clear(padic_t rop)
```

Clears all memory used by the p-adic number `rop`.

```
void _padic_canonicalise(padic_t rop, const padic_ctx_t ctx)
```

Brings the p-adic number `rop` into canonical form.

That is to say, ensures that either $u = v = 0$ or $p \nmid u$. There is no reduction modulo a power of p.

```
void _padic_reduce(padic_t rop, const padic_ctx_t ctx)
```

Given a p-adic number `rop` in canonical form, reduces it modulo p^N.

```
void padic_reduce(padic_t rop, const padic_ctx_t ctx)
```

Ensures that the p-adic number `rop` is reduced.

51.5 Randomisation

```
void padic_randtest(padic_t rop, flint_rand_t state, const
   padic_ctx_t ctx)
```

Sets `rop` to a random p-adic number modulo p^N with valuation in the range $[-\lceil N/10 \rceil, N)$, $[N - \lceil -N/10 \rceil, N)$, or $[-10, 0)$ as N is positive, negative or zero, whenever `rop` is non-zero.

```
void padic_randtest_not_zero(padic_t rop, flint_rand_t
   state, const padic_ctx_t ctx)
```

Sets `rop` to a random non-zero p-adic number modulo p^N, where the range of the valuation is as for the function `padic_randtest()`.

```
void padic_randtest_int(padic_t rop, flint_rand_t state,
   const padic_ctx_t ctx)
```

Sets `rop` to a random p-adic integer modulo p^N.

Note that whenever $N \leq 0$, `rop` is set to zero.

51.6 Assignments and conversions

All assignment functions set the value of `rop` from `op`, reduced to the precision of `rop`.

```
void padic_set(padic_t rop, const padic_t op, const
   padic_ctx_t ctx)
```

Sets `rop` to the p-adic number `op`.

```
void padic_set_si(padic_t rop, slong op, const padic_ctx_t
   ctx)
```

Sets the p-adic number `rop` to the `slong` integer `op`.

```
void padic_set_ui(padic_t rop, ulong op, const padic_ctx_t
   ctx)
```

Sets the p-adic number `rop` to the `ulong` integer `op`.

```
void padic_set_fmpz(padic_t rop, const fmpz_t op, const
   padic_ctx_t ctx)
```

Sets the p-adic number `rop` to the integer `op`.

```
void padic_set_fmpq(padic_t rop, const fmpq_t op, const
   padic_ctx_t ctx)
```

Sets `rop` to the rational `op`.

```
void padic_set_mpz(padic_t rop, const mpz_t op, const
   padic_ctx_t ctx)
```

Sets the p-adic number `rop` to the MPIR integer `op`.

```
void padic_set_mpq(padic_t rop, const mpq_t op, const
   padic_ctx_t ctx)
```

Sets `rop` to the MPIR rational `op`.

```
void padic_get_fmpz(fmpz_t rop, const padic_t op, const
   padic_ctx_t ctx)
```

Sets the integer `rop` to the exact p-adic integer `op`.

If `op` is not a p-adic integer, raises an `abort` signal.

```
void padic_get_fmpq(fmpq_t rop, const padic_t op, const
   padic_ctx_t ctx)
```

Sets the rational `rop` to the p-adic number `op`.

```
void padic_get_mpz(mpz_t rop, const padic_t op, const
   padic_ctx_t ctx)
```

Sets the MPIR integer `rop` to the p-adic integer `op`.

If `op` is not a p-adic integer, raises an `abort` signal.

```
void padic_get_mpq(mpq_t rop, const padic_t op, const
   padic_ctx_t ctx)
```

Sets the MPIR rational `rop` to the value of `op`.

```
void padic_swap(padic_t op1, padic_t op2)
```

Swaps the two p-adic numbers `op1` and `op2`.

Note that this includes swapping the precisions. In particular, this operation is not equivalent to swapping `op1` and `op2` using `padic_set()` and an auxiliary variable whenever the precisions of the two elements are different.

```
void padic_zero(padic_t rop)
```

Sets the p-adic number `rop` to zero.

```
void padic_one(padic_t rop)
```

Sets the p-adic number `rop` to one, reduced modulo the precision of `rop`.

51.7 Comparison

```
int padic_is_zero(const padic_t op)
```

Returns whether `op` is equal to zero.

```
int padic_is_one(const padic_t op)
```

Returns whether `op` is equal to one, that is, whether $u = 1$ and $v = 0$.

```
int padic_equal(const padic_t op1, const padic_t op2)
```

Returns whether `op1` and `op2` are equal, that is, whether $u_1 = u_2$ and $v_1 = v_2$.

51.8 Arithmetic operations

```
slong * _padic_lifts_exps(slong *n, slong N)
```

Given a positive integer N define the sequence $a_0 = N, a_1 = \lceil a_0/2 \rceil, \ldots, a_{n-1} = \lceil a_{n-2}/2 \rceil = 1$. Then $n = \lceil \log_2 N \rceil + 1$.

This function sets n and allocates and returns the array a.

```
void _padic_lifts_pows(fmpz *pow, const slong *a, slong n,
   const fmpz_t p)
```

Given an array a as computed above, this function computes the corresponding powers of p, that is, `pow[i]` is equal to p^{a_i}.

```
void padic_add(padic_t rop, const padic_t op1, const
   padic_t op2, const padic_ctx_t ctx)
```

Sets `rop` to the sum of `op1` and `op2`.

```
void padic_sub(padic_t rop, const padic_t op1, const
   padic_t op2, const padic_ctx_t ctx)
```

Sets `rop` to the difference of `op1` and `op2`.

```
void padic_neg(padic_t rop, const padic_t op, const
   padic_ctx_t ctx)
```

Sets `rop` to the additive inverse of `op`.

```
void padic_mul(padic_t rop, const padic_t op1, const
   padic_t op2, const padic_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`.

```
void padic_shift(padic_t rop, const padic_t op, slong v,
   const padic_ctx_t ctx)
```

Sets `rop` to the product of `op` and p^v.

```
void padic_div(padic_t rop, const padic_t op1, const
   padic_t op2, const padic_ctx_t ctx)
```

Sets `rop` to the quotient of `op1` and `op2`.

```
void _padic_inv_precompute(padic_inv_t S, const fmpz_t p,
   slong N)
```

Pre-computes some data and allocates temporary space for p-adic inversion using Hensel lifting.

```
void _padic_inv_clear(padic_inv_t S)
```

Frees the memory used by S.

```
void _padic_inv_precomp(fmpz_t rop, const fmpz_t op, const
   padic_inv_t S)
```

Sets `rop` to the inverse of `op` modulo p^N, assuming that `op` is a unit and $N \geq 1$.

In the current implementation, allows aliasing, but this might change in future versions.

Uses some data S precomputed by calling the function `_padic_inv_precompute()`. Note that this object is not declared `const` and in fact it carries a field providing temporary work space. This allows repeated calls of this function to avoid repeated memory allocations, as used e.g. by the function `padic_log()`.

```
void _padic_inv(fmpz_t rop, const fmpz_t op, const fmpz_t
   p, slong N)
```

Sets `rop` to the inverse of `op` modulo p^N, assuming that `op` is a unit and $N \geq 1$.

In the current implementation, allows aliasing, but this might change in future versions.

```
void padic_inv(padic_t rop, const padic_t op, const
   padic_ctx_t ctx)
```

Computes the inverse of `op` modulo p^N.

Suppose that `op` is given as $x = up^v$. Raises an `abort` signal if $v < -N$. Otherwise, computes the inverse of u modulo p^{N+v}.

This function employs Hensel lifting of an inverse modulo p.

```
int padic_sqrt(padic_rop, const padic_t op, const
   padic_ctx_t ctx)
```

Returns whether `op` is a p-adic square. If this is the case, sets `rop` to one of the square roots; otherwise, the value of `rop` is undefined.

We have the following theorem:

Let $u \in \mathbf{Z}^\times$. Then u is a square if and only if $u \bmod p$ is a square in $\mathbf{Z}/p\mathbf{Z}$, for $p > 2$, or if $u \bmod 8$ is a square in $\mathbf{Z}/8\mathbf{Z}$, for $p = 2$.

```
void padic_pow_si(padic_t rop, const padic_t op, slong e,
   const padic_ctx_t ctx)
```

Sets `rop` to `op` raised to the power e, which is defined as one whenever $e = 0$.

Assumes that some computations involving e and the valuation of `op` do not overflow in the `slong` range.

Note that if the input $x = p^v u$ is defined modulo p^N then $x^e = p^{ev}u^e$ is defined modulo $p^{N+(e-1)v}$, which is a precision loss in case $v < 0$.

51.9 Exponential

```
slong _padic_exp_bound(slong v, slong N, const fmpz_t p)
```

Returns an integer i such that for all $j \geq i$ we have $\operatorname{ord}_p(x^j/j!) \geq N$, where $\operatorname{ord}_p(x) = v$.

When p is a word-sized prime, returns $\left\lceil \frac{(p-1)N-1}{(p-1)v-1} \right\rceil$. Otherwise, returns $\lceil N/v \rceil$.

Assumes that $v < N$. Moreover, v has to be at least 2 or 1, depending on whether p is 2 or odd.

```
void _padic_exp_rectangular(fmpz_t rop, const fmpz_t u,
    slong v, const fmpz_t p, slong N)

void _padic_exp_balanced(fmpz_t rop, const fmpz_t u, slong
    v, const fmpz_t p, slong N)

void _padic_exp(fmpz_t rop, const fmpz_t u, slong v, const
    fmpz_t p, slong N)
```

Sets `rop` to the p-exponential function evaluated at $x = p^v u$, reduced modulo p^N.

Assumes that $x \neq 0$, that $\operatorname{ord}_p(x) < N$ and that $\exp(x)$ converges, that is, that $\operatorname{ord}_p(x)$ is at least 2 or 1 depending on whether the prime p is 2 or odd.

Supports aliasing between `rop` and u.

```
int padic_exp(padic_t y, const padic_t x, const padic_ctx_t
    ctx)
```

Returns whether the p-adic exponential function converges at the p-adic number x, and if so sets y to its value.

The p-adic exponential function is defined by the usual series

$$\exp_p(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

but this only converges only when $\operatorname{ord}_p(x) > 1/(p-1)$. For elements $x \in \mathbf{Q}_p$, this means that $\operatorname{ord}_p(x) \geq 1$ when $p \geq 3$ and $\operatorname{ord}_2(x) \geq 2$ when $p = 2$.

```
int padic_exp_rectangular(padic_t y, const padic_t x, const
    padic_ctx_t ctx)
```

Returns whether the p-adic exponential function converges at the p-adic number x, and if so sets y to its value.

Uses a rectangular splitting algorithm to evaluate the series expression of $\exp(x) \bmod p^N$.

```
int padic_exp_balanced(padic_t y, const padic_t x, const
    padic_ctx_t ctx)
```

Returns whether the p-adic exponential function converges at the p-adic number x, and if so sets y to its value.

Uses a balanced approach, balancing the size of chunks of x with the valuation and hence the rate of convergence, which results in a quasi-linear algorithm in N, for fixed p.

51.10 Logarithm

```
slong _padic_log_bound(slong v, slong N, const fmpz_t p)
```

Returns b such that for all $i \geq b$ we have

$$iv - \operatorname{ord}_p(i) \geq N$$

where $v \geq 1$.

Assumes that $1 \leq v < N$ or $2 \leq v < N$ when p is odd or $p = 2$, respectively, and also that $N < 2^{f-2}$ where f is `FLINT_BITS`.

```
void _padic_log(fmpz_t z, const fmpz_t y, slong v, const
   fmpz_t p, slong N)

void _padic_log_rectangular(fmpz_t z, const fmpz_t y, slong
   v, const fmpz_t p, slong N)

void _padic_log_satoh(fmpz_t z, const fmpz_t y, slong v,
   const fmpz_t p, slong N)

void _padic_log_balanced(fmpz_t z, const fmpz_t y, slong v,
   const fmpz_t p, slong N)
```

Computes

$$z = -\sum_{i=1}^{\infty} \frac{y^i}{i} \pmod{p^N},$$

reduced modulo p^N.

Note that this can be used to compute the p-adic logarithm via the equation

$$\begin{aligned}\log(x) &= \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i} \\ &= -\sum_{i=1}^{\infty} \frac{(1-x)^i}{i}.\end{aligned}$$

Assumes that $y = 1 - x$ is non-zero and that $v = \operatorname{ord}_p(y)$ is at least 1 when p is odd and at least 2 when $p = 2$ so that the series converges.

Assumes that $v < N$, and hence in particular $N \geq 2$.

Does not support aliasing between y and z.

```
int padic_log(padic_t rop, const padic_t op, const
   padic_ctx_t ctx)
```

Returns whether the p-adic logarithm function converges at the p-adic number `op`, and if so sets `rop` to its value.

The p-adic logarithm function is defined by the usual series

$$\log_p(x) = \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i}$$

but this only converges when $\operatorname{ord}_p(x)$ is at least 2 or 1 when $p = 2$ or $p > 2$, respectively.

```
int padic_log_rectangular(padic_t rop, const padic_t op,
   const padic_ctx_t ctx)
```

Returns whether the p-adic logarithm function converges at the p-adic number `op`, and if so sets `rop` to its value.

Uses a rectangular splitting algorithm to evaluate the series expression of $\log(x) \bmod p^N$.

```
int padic_log_satoh(padic_t rop, const padic_t op, const
   padic_ctx_t ctx)
```

Returns whether the p-adic logarithm function converges at the p-adic number `op`, and if so sets `rop` to its value.

Uses an algorithm based on a result of Satoh, Skjernaa and Taguchi that $\operatorname{ord}_p\left(a^{p^k}-1\right) > k$, which implies that

$$\log(a) \equiv p^{-k}\Big(\log\left(a^{p^k}\right) \pmod{p^{N+k}}\Big) \pmod{p^N}.$$

```
int padic_log_balanced(padic_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Returns whether the p-adic logarithm function converges at the p-adic number `op`, and if so sets `rop` to its value.

51.11 Special functions

```
void _padic_teichmuller(fmpz_t rop, const fmpz_t op, const
    fmpz_t p, slong N)
```

Computes the Teichmuller lift of the p-adic unit `op`, assuming that $N \geq 1$.

Supports aliasing between `rop` and `op`.

```
void padic_teichmuller(padic_t rop, const padic_t op, const
    padic_ctx_t ctx)
```

Computes the Teichmuller lift of the p-adic unit `op`.

If `op` is a p-adic integer divisible by p, sets `rop` to zero, which satisfies $t^p - t = 0$, although it is clearly not a $(p-1)$-st root of unity.

If `op` has negative valuation, raises an `abort` signal.

```
ulong padic_val_fac_ui_2(ulong n)
```

Computes the 2-adic valuation of $n!$.

Note that since n fits into an `ulong`, so does $\operatorname{ord}_2(n!)$ since $\operatorname{ord}_2(n!) \leq (n-1)/(p-1) = n-1$.

```
ulong padic_val_fac_ui(ulong n, const fmpz_t p)
```

Computes the p-adic valuation of $n!$.

Note that since n fits into an `ulong`, so does $\operatorname{ord}_p(n!)$ since $\operatorname{ord}_p(n!) \leq (n-1)/(p-1)$.

```
void padic_val_fac(fmpz_t rop, const fmpz_t op, const
    fmpz_t p)
```

Sets `rop` to the p-adic valuation of the factorial of `op`, assuming that `op` is non-negative.

51.12 Input and output

```
char * padic_get_str(char * str, const padic_t op, const
    padic_ctx_t ctx)
```

Returns the string representation of the p-adic number `op` according to the printing mode set in the context.

If `str` is `NULL` then a new block of memory is allocated and a pointer to this is returned. Otherwise, it is assumed that the string `str` is large enough to hold the representation and it is also the return value.

```
int _padic_fprint(FILE * file, const fmpz_t u, slong v,
    const padic_ctx_t ctx)

int padic_fprint(FILE * file, const padic_t op, const
    padic_ctx_t ctx)
```

Prints the string representation of the p-adic number `op` to the stream `file`.

In the current implementation, always returns 1.

```
int _padic_print(const fmpz_t u, slong v, const padic_ctx_t
    ctx)

int padic_print(const padic_t op, const padic_ctx_t ctx)
```

Prints the string representation of the p-adic number `op` to the stream `stdout`.

In the current implementation, always returns 1.

```
void padic_debug(const padic_t op)
```

Prints debug information about `op` to the stream `stdout`, in the format `"(u v N)"`.

§52. padic_mat

52.1 Module documentation

We represent a matrix over $\mathbf{Q}_p$ as a product $p^v M$, where p is a prime number, $v \in \mathbf{Z}$ and M a matrix over $\mathbf{Z}$.

We say this matrix is in *canonical form* if either M is zero, in which case we choose $v = 0$, too, or if M contains at least one p-adic unit.

We say this matrix is *reduced* modulo p^N if it is canonical form and if all coefficients of M lie in the range $[0, p^{N-v})$.

52.2 Macros

`fmpz_mat_struct * padic_mat(const padic_mat_t A)`

Returns a pointer to the unit part of the matrix, which is a matrix over $\mathbf{Z}$.

The return value can be used as an argument to the functions in the `fmpz_mat` module.

`fmpz * padic_mat_entry(const padic_mat_t A, slong i, slong j)`

Returns a pointer to unit part of the entry in position (i, j). Note that this is not necessarily a unit.

The return value can be used as an argument to the functions in the `fmpz` module.

`slong padic_mat_val(const padic_mat_t A)`

Returns the valuation of the matrix.

This is implemented as a macro and can be used as an *lvalue* as well as an *rvalue*.

`slong padic_mat_nrows(const padic_mat_t A)`

Returns the number of rows of the matrix A.

`slong padic_mat_ncols(const padic_mat_t A)`

Returns the number of columns of the matrix A.

52.3 Memory management

`void padic_mat_init(padic_mat_t A, slong r, slong c)`

Initialises the matrix A as a zero matrix with the specified numbers of rows and columns and precision `PADIC_DEFAULT_PREC`.

```
void padic_mat_init2(padic_mat_t A, slong r, slong c, slong
   prec)
```

Initialises the matrix A as a zero matrix with the specified numbers of rows and columns and the given precision.

```
void padic_mat_clear(padic_mat_t A)
```

Clears the matrix A.

```
void _padic_mat_canonicalise(padic_mat_t A, const
   padic_ctx_t ctx)
```

Ensures that the matrix A is in canonical form.

```
void _padic_mat_reduce(padic_mat_t A, const padic_ctx_t ctx)
```

Ensures that the matrix A is reduced modulo p^N, assuming that it is in canonical form already.

```
void padic_mat_reduce(padic_mat_t A, const padic_ctx_t ctx)
```

Ensures that the matrix A is reduced modulo p^N, without assuming that it is necessarily in canonical form.

```
int padic_mat_is_empty(const padic_mat_t A)
```

Returns whether the matrix A is empty, that is, whether it has zero rows or zero columns.

```
int padic_mat_is_square(const padic_mat_t A)
```

Returns whether the matrix A is square.

```
int padic_mat_is_canonical(const padic_mat_t A, const
   fmpz_t p)
```

Returns whether the matrix A is in canonical form.

52.4 Basic assignment

```
void padic_mat_set(padic_mat_t B, const padic_mat_t A)
```

Sets B to a copy of A, respecting the precision of B.

```
void padic_mat_swap(padic_mat_t A, padic_mat_t B)
```

Swaps the two matrices A and B. This is done efficiently by swapping pointers.

```
void padic_mat_zero(padic_mat_t A)
```

Sets the matrix A to zero.

```
void padic_mat_one(padic_mat_t A)
```

Sets the matrix A to the identity matrix. If the precision is negative then the matrix will be the zero matrix.

52.5 Conversions

```
void padic_mat_set_fmpq_mat(padic_mat_t B, const fmpq_mat_t
   A, const padic_ctx_t ctx)
```

Sets the p-adic matrix B to the rational matrix A, reduced according to the given context.

```
void padic_mat_get_fmpq_mat(fmpq_mat_t B, const padic_mat_t
   A, const padic_ctx_t ctx)
```

Sets the rational matrix B to the p-adic matrices A; no reduction takes place.

52.6 Entries

Because of the choice of the data structure, representing the matrix as $p^v M$, setting an entry of the matrix might lead to changes in all entries in the matrix M. Also, a specific entry is not readily available as a p-adic number. Thus, there are separate functions available for getting and setting entries.

```
void padic_mat_get_entry_padic(padic_t rop, const
   padic_mat_t op, slong i, slong j, const padic_ctx_t ctx)
```

Sets `rop` to the entry in position (i, j) in the matrix `op`.

```
void padic_mat_set_entry_padic(padic_mat_t rop, slong i,
   slong j, const padic_t op, const padic_ctx_t ctx)
```

Sets the entry in position (i, j) in the matrix to `rop`.

52.7 Comparison

```
int padic_mat_equal(const padic_mat_t A, const padic_mat_t
   B)
```

Returns whether the two matrices A and B are equal.

```
int padic_mat_is_zero(const padic_mat_t A)
```

Returns whether the matrix A is zero.

52.8 Input and output

```
int padic_mat_fprint(FILE * file, const padic_mat_t A,
   const padic_ctx_t ctx)
```

Prints a simple representation of the matrix A to the output stream `file`. The format is the number of rows, a space, the number of columns, two spaces, followed by a list of all the entries, one row after the other.

In the current implementation, always returns 1.

```
int padic_mat_fprint_pretty(FILE * file, const padic_mat_t
   A, const padic_ctx_t ctx)
```

Prints a *pretty* representation of the matrix A to the output stream `file`.

In the current implementation, always returns 1.

```
int padic_mat_print(const padic_mat_t A, const padic_ctx_t
   ctx)
```

```
int padic_mat_print_pretty(const padic_mat_t A, const
   padic_ctx_t ctx)
```

52.9 Random matrix generation

```
void padic_mat_randtest(padic_mat_t A, flint_rand_t state,
   const padic_ctx_t ctx)
```

Sets A to a random matrix.

The valuation will be in the range $[-\lceil N/10 \rceil, N)$, $[N - \lceil -N/10 \rceil, N)$, or $[-10, 0)$ as N is positive, negative or zero.

52.10 Transpose

```
void padic_mat_transpose(padic_mat_t B, const padic_mat_t A)
```

Sets B to A^t.

52.11 Addition and subtraction

```
void _padic_mat_add(padic_mat_t C, const padic_mat_t A,
   const padic_mat_t B, const padic_ctx_t ctx)
```

Sets C to the exact sum $A + B$, ensuring that the result is in canonical form.

```
void padic_mat_add(padic_mat_t C, const padic_mat_t A,
   const padic_mat_t B, const padic_ctx_t ctx)
```

Sets C to the sum $A + B$ modulo p^N.

```
void _padic_mat_sub(padic_mat_t C, const padic_mat_t A,
   const padic_mat_t B, const padic_ctx_t ctx)
```

Sets C to the exact difference $A - B$, ensuring that the result is in canonical form.

```
void padic_mat_sub(padic_mat_t C, const padic_mat_t A,
   const padic_mat_t B, const padic_ctx_t ctx)
```

Sets C to $A - B$, ensuring that the result is reduced.

```
void _padic_mat_neg(padic_mat_t B, const padic_mat_t A)
```

Sets B to $-A$ in canonical form.

```
void padic_mat_neg(padic_mat_t B, const padic_mat_t A,
   const padic_ctx_t ctx)
```

Sets B to $-A$, ensuring the result is reduced.

52.12 Scalar operations

```
void _padic_mat_scalar_mul_padic(padic_mat_t B, const
   padic_mat_t A, const padic_t c, const padic_ctx_t ctx)
```

Sets B to cA, ensuring that the result is in canonical form.

```
void padic_mat_scalar_mul_padic(padic_mat_t B, const
   padic_mat_t A, const padic_t c, const padic_ctx_t ctx)
```

Sets B to cA, ensuring that the result is reduced.

```
void _padic_mat_scalar_mul_fmpz(padic_mat_t B, const
   padic_mat_t A, const fmpz_t c, const padic_ctx_t ctx)
```

Sets B to cA, ensuring that the result is in canonical form.

```
void padic_mat_scalar_mul_fmpz(padic_mat_t B, const
   padic_mat_t A, const fmpz_t c, const padic_ctx_t ctx)
```

Sets B to cA, ensuring that the result is reduced.

```
void padic_mat_scalar_div_fmpz(padic_mat_t B, const
   padic_mat_t A, const fmpz_t c, const padic_ctx_t ctx)
```

Sets B to $c^{-1}A$, assuming that $c \neq 0$. Ensures that the result B is reduced.

52.13 Multiplication

```
void _padic_mat_mul(padic_mat_t C, const padic_mat_t A,
   const padic_mat_t B, const padic_ctx_t ctx)
```

Sets C to the product AB of the two matrices A and B, ensuring that C is in canonical form.

```
void padic_mat_mul(padic_mat_t C, const padic_mat_t A,
   const padic_mat_t B, const padic_ctx_t ctx)
```

Sets C to the product AB of the two matrices A and B, ensuring that C is reduced.

§53. padic_poly

53.1 Module documentation

We represent a polynomial in $\mathbf{Q}_p[x]$ as a product $p^v f(x)$, where p is a prime number, $v \in \mathbf{Z}$ and $f(x) \in \mathbf{Z}[x]$.

As a data structure, we call this polynomial *normalised* if the polynomial $f(x)$ is *normalised*, that is, if the top coefficient is non-zero.

We say this polynomial is in *canonical form* if one of the coefficients of $f(x)$ is a p-adic unit. If $f(x)$ is the zero polynomial, we require that $v = 0$.

We say this polynomial is *reduced* modulo p^N if it is canonical form and if all coefficients lie in the range $[0, p^N)$.

53.2 Memory management

```
void padic_poly_init(padic_poly_t poly)
```

Initialises `poly` for use, setting its length to zero. The precision of the polynomial is set to `PADIC_DEFAULT_PREC`. A corresponding call to `padic_poly_clear()` must be made after finishing with the `padic_poly_t` to free the memory used by the polynomial.

```
void padic_poly_init2(padic_poly_t poly, slong alloc, slong
   prec)
```

Initialises `poly` with space for at least `alloc` coefficients and sets the length to zero. The allocated coefficients are all set to zero. The precision is set to `prec`.

```
void padic_poly_realloc(padic_poly_t poly, slong alloc,
   const fmpz_t p)
```

Reallocates the given polynomial to have space for `alloc` coefficients. If `alloc` is zero the polynomial is cleared and then reinitialised. If the current length is greater than `alloc` the polynomial is first truncated to length `alloc`.

```
void padic_poly_fit_length(padic_poly_t poly, slong len)
```

If `len` is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least `len` coefficients. No data is lost when calling this function.

The function efficiently deals with the case where `fit_length` is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

```
void _padic_poly_set_length(padic_poly_t poly, slong len)
```

Demotes the coefficients of `poly` beyond `len` and sets the length of `poly` to `len`.

Note that if the current length is greater than `len` the polynomial may no slonger be in canonical form.

```
void padic_poly_clear(padic_poly_t poly)
```

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

```
void _padic_poly_normalise(padic_poly_t poly)
```

Sets the length of `poly` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

```
void _padic_poly_canonicalise(fmpz *poly, slong *v, slong
   len, const fmpz_t p)

void padic_poly_canonicalise(padic_poly_t poly, const
   fmpz_t p)
```

Brings the polynomial `poly` into canonical form, assuming that it is normalised already. Does *not* carry out any reduction.

```
void padic_poly_reduce(padic_poly_t poly, const padic_ctx_t
   ctx)
```

Reduces the polynomial `poly` modulo p^N, assuming that it is in canonical form already.

```
void padic_poly_truncate(padic_poly_t poly, slong n, const
   fmpz_t p)
```

Truncates the polynomial to length at most n.

53.3 Polynomial parameters

```
slong padic_poly_degree(padic_poly_t poly)
```

Returns the degree of the polynomial `poly`.

```
slong padic_poly_length(padic_poly_t poly)
```

Returns the length of the polynomial `poly`.

```
slong padic_poly_val(padic_poly_t poly)
```

Returns the valuation of the polynomial `poly`, which is defined to be the minimum valuation of all its coefficients.

The valuation of the zero polynomial is 0.

Note that this is implemented as a macro and can be used as either a `lvalue` or a `rvalue`.

```
slong padic_poly_prec(padic_poly_t poly)
```

Returns the precision of the polynomial `poly`.

Note that this is implemented as a macro and can be used as either a `lvalue` or a `rvalue`.

Note that increasing the precision might require a call to `padic_poly_reduce()`.

53.4 Randomisation

```
void padic_poly_randtest(padic_poly_t f, flint_rand_t
   state, slong len, const padic_ctx_t ctx)
```

Sets f to a random polynomial of length at most `len` with entries reduced modulo p^N.

```
void padic_poly_randtest_not_zero(padic_poly_t f,
   flint_rand_t state, slong len, const padic_ctx_t ctx)
```

Sets f to a non-zero random polynomial of length at most `len` with entries reduced modulo p^N.

```
void padic_poly_randtest_val(padic_poly_t f, flint_rand_t
   state, slong val, slong len, const padic_ctx_t ctx)
```

Sets f to a random polynomial of length at most `len` with at most the prescribed valuation `val` and entries reduced modulo p^N.

Specifically, we aim to set the valuation to be exactly equal to `val`, but do not check for additional cancellation when creating the coefficients.

53.5 Assignment and basic manipulation

```
void padic_poly_set_padic(padic_poly_t poly, const padic_t
   x, const padic_ctx_t ctx)
```

Sets the polynomial `poly` to the p-adic number x, reduced to the precision of the polynomial.

```
void padic_poly_set(padic_poly_t poly1, const padic_poly_t
   poly2, const padic_ctx_t ctx)
```

Sets the polynomial `poly1` to the polynomial `poly2`, reduced to the precision of `poly1`.

```
void padic_poly_set_si(padic_poly_t poly, slong x, const
   padic_ctx_t ctx)
```

Sets the polynomial `poly` to the `signed slong` integer x reduced to the precision of the polynomial.

```
void padic_poly_set_ui(padic_poly_t poly, ulong x, const
   padic_ctx_t ctx)
```

Sets the polynomial `poly` to the `unsigned slong` integer x reduced to the precision of the polynomial.

```
void padic_poly_set_fmpz(padic_poly_t poly, const fmpz_t x,
   const padic_ctx_t ctx)
```

Sets the polynomial `poly` to the integer x reduced to the precision of the polynomial.

```
void padic_poly_set_fmpq(padic_poly_t poly, const fmpq_t x,
   const padic_ctx_t ctx)
```

Sets the polynomial `poly` to the value of the rational x, reduced to the precision of the polynomial.

```
void padic_poly_set_fmpz_poly(padic_poly_t rop, const
   fmpz_poly_t op, const padic_ctx_t ctx)
```

Sets the polynomial `rop` to the integer polynomial `op` reduced to the precision of the polynomial.

```
void padic_poly_set_fmpq_poly(padic_poly_t rop, const
   fmpq_poly_t op, const padic_ctx_t ctx)
```

Sets the polynomial `rop` to the value of the rational polynomial `op`, reduced to the precision of the polynomial.

```
int padic_poly_get_fmpz_poly(fmpz_poly_t rop, const
   padic_poly_t op, const padic_ctx_t ctx)
```

Sets the integer polynomial `rop` to the value of the p-adic polynomial `op` and returns 1 if the polynomial is p-adically integral. Otherwise, returns 0.

```
void padic_poly_get_fmpq_poly(fmpq_poly_t rop, const
   padic_poly_t op, const padic_ctx_t ctx)
```

Sets `rop` to the rational polynomial corresponding to the p-adic polynomial `op`.

```
void padic_poly_zero(padic_poly_t poly)
```

Sets `poly` to the zero polynomial.

```
void padic_poly_one(padic_poly_t poly)
```

Sets `poly` to the constant polynomial 1, reduced to the precision of the polynomial.

```
void padic_poly_swap(padic_poly_t poly1, padic_poly_t poly2)
```

Swaps the two polynomials `poly1` and `poly2`, including their precisions.

This is done efficiently by swapping pointers.

53.6 Getting and setting coefficients

```
void padic_poly_get_coeff_padic(padic_t c, const
   padic_poly_t poly, slong n, const padic_ctx_t ctx)
```

Sets c to the coefficient of x^n in the polynomial, reduced modulo the precision of c.

```
void padic_poly_set_coeff_padic(padic_poly_t f, slong n,
   const padic_t c, const padic_ctx_t ctx)
```

Sets the coefficient of x^n in the polynomial f to c, reduced to the precision of the polynomial f.

Note that this operation can take linear time in the length of the polynomial.

53.7 Comparison

```
int padic_poly_equal(const padic_poly_t poly1, const
   padic_poly_t poly2)
```

Returns whether the two polynomials `poly1` and `poly2` are equal.

```
int padic_poly_is_zero(const padic_poly_t poly)
```

Returns whether the polynomial `poly` is the zero polynomial.

```
int padic_poly_is_one(const padic_poly_t poly, const
    padic_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal to the constant polynomial 1, taking the precision of the polynomial into account.

53.8 Addition and subtraction

```
void _padic_poly_add(fmpz *rop, slong *rval, slong N, const
    fmpz *op1, slong val1, slong len1, slong N1, const fmpz
    *op2, slong val2, slong len2, slong N2, const
    padic_ctx_t ctx)
```

Sets `(rop, *val, FLINT_MAX(len1, len2)` to the sum of `(op1, val1, len1)` and `(op2, val2, len2)`.

Assumes that the input is reduced and guarantees that this is also the case for the output.

Assumes that $\min\{v_1, v_2\} < N$.

Supports aliasing between the output and input arguments.

```
void padic_poly_add(padic_poly_t f, const padic_poly_t g,
    const padic_poly_t h, const padic_ctx_t ctx);
```

Sets f to the sum $g + h$.

```
void _padic_poly_sub(fmpz *rop, slong *rval, const fmpz
    *op1, slong val1, slong len1, const fmpz *op2, slong
    val2, slong len2, const padic_ctx_t ctx);
```

Sets `(rop, *val, FLINT_MAX(len1, len2)` to the difference of `(op1, val1, len1)` and `(op2, val2, len2)`.

Assumes that the input is reduced and guarantees that this is also the case for the output.

Assumes that $\min\{v_1, v_2\} < N$.

Support aliasing between the output and input arguments.

```
void padic_poly_sub(padic_poly_t f, const padic_poly_t g,
    const padic_poly_t h, const padic_ctx_t ctx);
```

Sets f to the difference $g - h$.

```
void padic_poly_neg(padic_poly_t f, const padic_poly_t g,
    const padic_ctx_t ctx);
```

Sets f to $-g$.

53.9 Scalar multiplication

```
void _padic_poly_scalar_mul_padic(fmpz *rop, slong *rval,
    const fmpz *op, slong val, slong len, const padic_t c,
    const padic_ctx_t ctx)
```

Sets `(rop, *rval, len)` to `(op, val, len)` multiplied by the scalar c.

The result will only be correctly reduced if the polynomial is non-zero. Otherwise, the array `(rop, len)` will be set to zero but the valuation `*rval` might be wrong.

```
void padic_poly_scalar_mul_padic(padic_poly_t rop, const
   padic_poly_t op, const padic_t c, const padic_ctx_t ctx)
```

Sets the polynomial `rop` to the product of the polynomial `op` and the p-adic number c, reducing the result modulo p^N.

53.10 Multiplication

```
void _padic_poly_mul(fmpz *rop, slong *rval, slong N, const
   fmpz *op1, slong val1, slong len1, const fmpz *op2,
   slong val2, slong len2, const padic_ctx_t ctx)
```

Sets `(rop, *rval, len1 + len2 - 1)` to the product of `(op1, val1, len1)` and `(op2, val2, len2)`.

Assumes that the resulting valuation `*rval`, which is the sum of the valuations `val1` and `val2`, is less than the precision N of the context.

Assumes that `len1 >= len2 > 0`.

```
void padic_poly_mul(padic_poly_t res, const padic_poly_t
   poly1, const padic_poly_t poly2, const padic_ctx_t ctx)
```

Sets the polynomial `res` to the product of the two polynomials `poly1` and `poly2`, reduced modulo p^N.

53.11 Powering

```
void _padic_poly_pow(fmpz *rop, slong *rval, slong N, const
   fmpz *op, slong val, slong len, ulong e, const
   padic_ctx_t ctx)
```

Sets the polynomial `(rop, *rval, e (len - 1)+ 1)` to the polynomial `(op, val, len)` raised to the power e.

Assumes that $e > 1$ and `len > 0`.

Does not support aliasing between the input and output arguments.

```
void padic_poly_pow(padic_poly_t rop, const padic_poly_t
   op, ulong e, const padic_ctx_t ctx)
```

Sets the polynomial `rop` to the polynomial `op` raised to the power e, reduced to the precision in `rop`.

In the special case $e = 0$, sets `rop` to the constant polynomial one reduced to the precision of `rop`. Also note that when $e = 1$, this operation sets `rop` to `op` and then reduces `rop`.

When the valuation of the input polynomial is negative, this results in a loss of p-adic precision. Suppose that the input polynomial is given to precision N and has valuation $v < 0$. The result then has valuation $ev < 0$ but is only correct to precision $N + (e - 1)v$.

53.12 Series inversion

```
void padic_poly_inv_series(padic_poly_t g, const
   padic_poly_t f, slong n, const padic_ctx_t ctx)
```

Computes the power series inverse g of f modulo X^n, where $n \geq 1$.

Given the polynomial $f \in \mathbf{Q}[X] \subset \mathbf{Q}_p[X]$, there exists a unique polynomial $f^{-1} \in \mathbf{Q}[X]$ such that $ff^{-1} = 1$ modulo X^n. This function sets g to f^{-1} reduced modulo p^N.

Assumes that the constant coefficient of f is non-zero.

Moreover, assumes that the valuation of the constant coefficient of f is minimal among the coefficients of f.

Note that the result g is zero if and only if $-\operatorname{ord}_p(f) \geq N$.

53.13 Derivative

```
void _padic_poly_derivative(fmpz *rop, slong *rval, slong
   N, const fmpz *op, slong val, slong len, const
   padic_ctx_t ctx)
```

Sets `(rop, rval)` to the derivative of `(op, val)` reduced modulo p^N.

Supports aliasing of the input and the output parameters.

```
void padic_poly_derivative(padic_poly_t rop, const
   padic_poly_t op, const padic_ctx_t ctx)
```

Sets `rop` to the derivative of `op`, reducing the result modulo the precision of `rop`.

53.14 Shifting

```
void padic_poly_shift_left(padic_poly_t rop, const
   padic_poly_t op, slong n, const padic_ctx_t ctx)
```

Notationally, sets the polynomial `rop` to the polynomial `op` multiplied by x^n, where $n \geq 0$, and reduces the result.

```
void padic_poly_shift_right(padic_poly_t rop, const
   padic_poly_t op, slong n)
```

Notationally, sets the polynomial `rop` to the polynomial `op` after floor division by x^n, where $n \geq 0$, ensuring the result is reduced.

53.15 Evaluation

```
void _padic_poly_evaluate_padic(fmpz_t u, slong *v, slong
   N, const fmpz *poly, slong val, slong len, const fmpz_t
   a, slong b, const padic_ctx_t ctx)

void padic_poly_evaluate_padic(padic_t y, const
   padic_poly_t poly, const padic_t a, const padic_ctx_t
   ctx)
```

Sets the p-adic number `y` to `poly` evaluated at a, reduced in the given context.

Suppose that the polynomial can be written as $F(X) = p^w f(X)$ with $\operatorname{ord}_p(f) = 1$, that $\operatorname{ord}_p(a) = b$ and that both are defined to precision N. Then f is defined to precision $N - w$ and so $f(a)$ is defined to precision $N - w$ when a is integral and $N - w + (n-1)b$ when $b < 0$, where $n = \deg(f)$. Thus, $y = F(a)$ is defined to precision N when a is integral and $N + (n-1)b$ when $b < 0$.

53.16 Composition

```
void _padic_poly_compose(fmpz *rop, slong *rval, slong N,
   const fmpz *op1, slong val1, slong len1, const fmpz
   *op2, slong val2, slong len2, const padic_ctx_t ctx)
```

Sets `(rop, *rval, (len1-1)*(len2-1)+1)` to the composition of the two input polynomials, reducing the result modulo p^N.

Assumes that `len1` is non-zero.

Does not support aliasing.

```
void padic_poly_compose(padic_poly_t rop, const
   padic_poly_t op1, const padic_poly_t op2, const
   padic_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`, reducing the result in the given context.

To be clear about the order of composition, let $f(X)$ and $g(X)$ denote the polynomials `op1` and `op2`, respectively. Then `rop` is set to $f(g(X))$.

```
void _padic_poly_compose_pow(fmpz *rop, slong *rval, slong
   N, const fmpz *op, slong val, slong len, slong k, const
   padic_ctx_t ctx)
```

Sets `(rop, *rval, (len - 1)*k + 1)` to the composition of `(op, val, len)` and the monomial x^k, where $k \geq 1$.

Assumes that `len` is positive.

Supports aliasing between the input and output polynomials.

```
void padic_poly_compose_pow(padic_poly_t rop, const
   padic_poly_t op, slong k, const padic_ctx_t ctx)
```

Sets `rop` to the composition of `op` and the monomial x^k, where $k \geq 1$.

Note that no reduction takes place.

53.17 Input and output

```
int padic_poly_debug(const padic_poly_t poly)
```

Prints the data defining the p-adic polynomial `poly` in a simple format useful for debugging purposes.

In the current implementation, always returns 1.

```
int _padic_poly_fprint(FILE *file, const fmpz *poly, slong
   val, slong len, const padic_ctx_t ctx)

int padic_poly_fprint(FILE *file, const padic_poly_t poly,
   const padic_ctx_t ctx)
```

Prints a simple representation of the polynomial `poly` to the stream `file`.

A non-zero polynomial is represented by the number of coeffients, two spaces, followed by a list of the coefficients, which are printed in a way depending on the print mode,

- In the `PADIC_TERSE` mode, the coefficients are printed as rational numbers.
- The `PADIC_SERIES` mode is currently not supported and will raise an abort signal.
- In the `PADIC_VAL_UNIT` mode, the coefficients are printed in the form $p^v u$.

The zero polynomial is represented by `"0"`.

In the current implementation, always returns 1.

```
int _padic_poly_print(const fmpz *poly, slong val, slong
    len, const padic_ctx_t ctx)

int padic_poly_print(const padic_poly_t poly, const
    padic_ctx_t ctx)
```

Prints a simple representation of the polynomial `poly` to `stdout`.

In the current implementation, always returns 1.

```
int _padic_poly_fprint_pretty(FILE *file, const fmpz *poly,
    slong val, slong len, const char *var, const padic_ctx_t
    ctx)

int padic_poly_fprint_pretty(FILE *file, const padic_poly_t
    poly, const char *var, const padic_ctx_t ctx)

int _padic_poly_print_pretty(FILE *file, const fmpz *poly,
    slong val, slong len, const char *var, const padic_ctx_t
    ctx)

int padic_poly_print_pretty(const padic_poly_t poly, const
    char *var, const padic_ctx_t ctx)
```

53.18 Testing

```
int _padic_poly_is_canonical(const fmpz *op, slong val,
    slong len, const padic_ctx_t ctx);

int padic_poly_is_canonical(const padic_poly_t op, const
    padic_ctx_t ctx);

int _padic_poly_is_reduced(const fmpz *op, slong val, slong
    len, slong N, const padic_ctx_t ctx);

int padic_poly_is_reduced(const padic_poly_t op, const
    padic_ctx_t ctx);
```

§54. qadic

54.1 Data structures

We represent an element of the extension $\mathbf{Q}_q \cong \mathbf{Q}_p[X]/(f(X))$ as a polynomial in $\mathbf{Q}_p[X]$ of degree less than $\deg(f)$.

As such, `qadic_struct` and `qadic_t` are typedef'ed as `padic_poly_struct` and `padic_poly_t`.

54.2 Context

We represent an unramified extension of $\mathbf{Q}_p$ via $\mathbf{Q}_q \cong \mathbf{Q}_p[X]/(f(X))$, where $f \in \mathbf{Q}_p[X]$ is a monic, irreducible polynomial which we assume to actually be in $\mathbf{Z}[X]$.

The first field in the context structure is a p-adic context struct `pctx`, which contains data about the prime p, precomputed powers, the printing mode etc.

The polynomial f is represented as a sparse polynomial using two arrays j and a of length `len`, where $f(X) = \sum_i a_i X^{j_i}$. We also assume that the array j is sorted in ascending order.

We choose this data structure to improve reduction modulo $f(X)$ in $\mathbf{Q}_p[X]$, assuming a sparse polynomial $f(X)$ is chosen.

The field `var` contains the name of a generator of the extension, which is used when printing the elements.

```
void qadic_ctx_init_conway(qadic_ctx_t ctx, const fmpz_t p,
   slong d, slong min, slong max, const char *var, enum
   padic_print_mode mode)
```

Initialises the context `ctx` with prime p, extension degree d, variable name `var` and printing mode `mode`.

Stores powers of p with exponents between `min` (inclusive) and `max` exclusive. Assumes that `min` is at most `max`.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

Assumes that the printing mode is one of `PADIC_TERSE`, `PADIC_SERIES`, or `PADIC_VAL_UNIT`.

This function also carries out some relevant precomputation for arithmetic in $\mathbf{Q}_p/(p^N)$ such as powers of p close to p^N.

```
void qadic_ctx_clear(qadic_ctx_t ctx);
```

Clears all memory that has been allocated as part of the context.

```
slong qadic_ctx_degree(const qadic_ctx_t ctx)
```

Returns the extension degree.

```
static __inline__ void qadic_ctx_print(const qadic_ctx_t
    ctx)
```

Prints the data from the given context.

54.3 Memory management

```
void qadic_init(qadic_t rop)
```

Initialises the element `rop`, setting its value to 0.

```
void qadic_init2(qadic_t rop, slong prec)
```

Initialises the element `rop` with the given output precision, setting the value to 0.

```
void qadic_clear(qadic_t rop)
```

Clears the element `rop`.

```
void _fmpz_poly_reduce(fmpz *R, slong lenR, const fmpz *a,
    const slong *j, slong len)
```

Reduces a polynomial `(R, lenR)` modulo a sparse monic polynomial $f(X) = \sum_i a_i X^{j_i}$ of degree at least 2.

Assumes that the array j of positive length `len` is sorted in ascending order.

Allows zero-padding in `(R, lenR)`.

```
void _fmpz_mod_poly_reduce(fmpz *R, slong lenR, const fmpz
    *a, const slong *j, slong len, const fmpz_t p)
```

Reduces a polynomial `(R, lenR)` modulo a sparse monic polynomial $f(X) = \sum_i a_i X^{j_i}$ of degree at least 2 in $\mathbf{Z}/(p)$, where p is typically a prime power.

Assumes that the array j of positive length `len` is sorted in ascending order.

Allows zero-padding in `(R, lenR)`.

```
void qadic_reduce(qadic_t rop, const qadic_ctx_t ctx)
```

Reduces `rop` modulo $f(X)$ and p^N.

54.4 Properties

```
slong qadic_val(const qadic_t op)
```

Returns the valuation of `op`.

```
slong qadic_prec(const qadic_t op)
```

Returns the precision of `op`.

54.5 Randomisation

```
void qadic_randtest(qadic_t rop, flint_rand_t state, const
    qadic_ctx_t ctx)
```

Generates a random element of $\mathbf{Q}_q$.

```
void qadic_randtest_not_zero(qadic_t rop, flint_rand_t
   state, const qadic_ctx_t ctx)
```

Generates a random non-zero element of $\mathbf{Q}_q$.

```
void qadic_randtest_val(qadic_t rop, flint_rand_t state,
   slong v, const qadic_ctx_t ctx)
```

Generates a random element of $\mathbf{Q}_q$ with prescribed valuation `val`.

Note that if $v \geq N$ then the element is necessarily zero.

```
void qadic_randtest_int(qadic_t rop, flint_rand_t state,
   const qadic_ctx_t ctx)
```

Generates a random element of $\mathbf{Q}_q$ with non-negative valuation.

54.6 Assignments and conversions

```
void qadic_set(qadic_t rop, const qadic_t op)
```

Sets `rop` to `op`.

```
void qadic_zero(qadic_t rop)
```

Sets `rop` to zero.

```
void qadic_one(qadic_t rop, const qadic_ctx_t ctx)
```

Sets `rop` to one, reduced in the given context.

Note that if the precision N is non-positive then `rop` is actually set to zero.

```
void qadic_gen(qadic_t rop, const qadic_ctx_t ctx)
```

Sets `rop` to the generator X for the extension when $N > 0$, and zero otherwise. If the extension degree is one, raises an abort signal.

```
void qadic_set_ui(qadic_t rop, ulong op, const qadic_ctx_t
   ctx)
```

Sets `rop` to the integer `op`, reduced in the context.

```
int qadic_get_padic(padic_t rop, const qadic_t op, const
   qadic_ctx_t ctx)
```

If the element `op` lies in $\mathbf{Q}_p$, sets `rop` to its value and returns 1; otherwise, returns 0.

54.7 Comparison

```
int qadic_is_zero(const qadic_t op)
```

Returns whether `op` is equal to zero.

```
int qadic_is_one(const qadic_t op, const qadic_ctx_t ctx)
```

Returns whether `op` is equal to one in the given context.

```
int qadic_equal(const qadic_t op1, const qadic_t op2)
```

Returns whether op1 and op2 are equal.

54.8 Basic arithmetic

```
void qadic_add(qadic_t rop, const qadic_t op1, const
   qadic_t op2, const qadic_ctx_t ctx)
```

Sets rop to the sum of op1 and op2.

Assumes that both op1 and op2 are reduced in the given context and ensures that rop is, too.

```
void qadic_sub(qadic_t rop, const qadic_t op1, const
   qadic_t op2, const qadic_ctx_t ctx)
```

Sets rop to the difference of op1 and op2.

Assumes that both op1 and op2 are reduced in the given context and ensures that rop is, too.

```
void qadic_neg(qadic_t rop, const qadic_t op, const
   qadic_ctx_t ctx)
```

Sets rop to the negative of op.

Assumes that op is reduced in the given context and ensures that rop is, too.

```
void qadic_mul(qadic_t rop, const qadic_t op1, const
   qadic_t op2, const qadic_ctx_t ctx)
```

Sets rop to the product of op1 and op2, reducing the output in the given context.

```
void _qadic_inv(fmpz *rop, const fmpz *op, slong len, const
   fmpz *a, const slong *j, slong lena, const fmpz_t p,
   slong N)
```

Sets (rop, d) to the inverse of (op, len) modulo $f(X)$ given by (a,j,lena) and p^N.

Assumes that (op,len) has valuation 0, that is, that it represents a p-adic unit.

Assumes that len is at most d.

Does not support aliasing.

```
void qadic_inv(qadic_t rop, const qadic_t op, const
   qadic_ctx_t ctx)
```

Sets rop to the inverse of op, reduced in the given context.

```
void _qadic_pow(fmpz *rop, const fmpz *op, slong len, const
   fmpz_t e, const fmpz *a, const slong *j, slong lena,
   const fmpz_t p)
```

Sets (rop, 2*d-1) to (op,len) raised to the power e, reduced modulo $f(X)$ given by (a, j, lena) and p, which is expected to be a prime power.

Assumes that $e \geq 0$ and that len is positive and at most d.

Although we require that rop provides space for $2d - 1$ coefficients, the output will be reduces modulo $f(X)$, which is a polynomial of degree d.

Does not support aliasing.

```
void qadic_pow(qadic_t rop, const qadic_t op, const fmpz_t
    e, const qadic_ctx_t ctx)
```

Sets `rop` the `op` raised to the power e.

Currently assumes that $e \geq 0$.

Note that for any input `op`, `rop` is set to one in the given context whenever $e = 0$.

54.9 Special functions

```
void _qadic_exp_rectangular(fmpz *rop, const fmpz *op,
    slong v, slong len, const fmpz *a, const slong *j, slong
    lena, const fmpz_t p, slong N, const fmpz_t pN)
```

Sets `(rop, 2*d - 1)` to the exponential of `(op, v, len)` reduced modulo p^N, assuming that the series converges.

Assumes that `(op, v, len)` is non-zero.

Does not support aliasing.

```
int qadic_exp_rectangular(qadic_t rop, const qadic_t op,
    const qadic_ctx_t ctx)
```

Returns whether the exponential series converges at `op` and sets `rop` to its value reduced modulo in the given context.

```
void _qadic_exp_balanced(fmpz *rop, const fmpz *x, slong v,
    slong len, const fmpz *a, const slong *j, slong lena,
    const fmpz_t p, slong N, const fmpz_t pN)
```

Sets `(rop, d)` to the exponential of `(op, v, len)` reduced modulo p^N, assuming that the series converges.

Assumes that `len` is in $[1, d)$ but supports zero padding, including the special case when `(op, len)` is zero.

Supports aliasing between `rop` and `op`.

```
int qadic_exp_balanced(qadic_t rop, const qadic_t op, const
    qadic_ctx_t ctx)
```

Returns whether the exponential series converges at `op` and sets `rop` to its value reduced modulo in the given context.

```
void _qadic_exp(fmpz *rop, const fmpz *op, slong v, slong
    len, const fmpz *a, const slong *j, slong lena, const
    fmpz_t p, slong N)
```

Sets `(rop, 2*d - 1)` to the exponential of `(op, v, len)` reduced modulo p^N, assuming that the series converges.

Assumes that `(op, v, len)` is non-zero.

Does not support aliasing.

```
int qadic_exp(qadic_t rop, const qadic_t op, const
    qadic_ctx_t ctx)
```

Returns whether the exponential series converges at `op` and sets `rop` to its value reduced modulo in the given context.

The exponential series converges if the valuation of `op` is at least 2 or 1 when p is even or odd, respectively.

```
void _qadic_log_rectangular(fmpz *z, const fmpz *y, slong
   v, slong len, const fmpz *a, const slong *j, slong lena,
   const fmpz_t p, slong N, const fmpz_t pN)
```

Computes

$$z = -\sum_{i=1}^{\infty} \frac{y^i}{i} \pmod{p^N}.$$

Note that this can be used to compute the p-adic logarithm via the equation

$$\begin{aligned}\log(x) &= \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i} \\ &= -\sum_{i=1}^{\infty} \frac{(1-x)^i}{i}.\end{aligned}$$

Assumes that $y = 1 - x$ is non-zero and that $v = \mathrm{ord}_p(y)$ is at least 1 when p is odd and at least 2 when $p = 2$ so that the series converges.

Assumes that y is reduced modulo p^N.

Assumes that $v < N$, and in particular $N \geq 2$.

Supports aliasing between y and z.

```
int qadic_log_rectangular(qadic_t rop, const qadic_t op,
   const padic_ctx_t ctx)
```

Returns whether the p-adic logarithm function converges at `op`, and if so sets `rop` to its value.

```
void _qadic_log_balanced(fmpz *z, const fmpz *y, slong len,
   const fmpz *a, const slong *j, slong lena, const fmpz_t
   p, slong N, const fmpz_t pN)
```

Computes (z, d) as

$$z = -\sum_{i=1}^{\infty} \frac{y^i}{i} \pmod{p^N}.$$

Assumes that $v = \mathrm{ord}_p(y)$ is at least 1 when p is odd and at least 2 when $p = 2$ so that the series converges.

Supports aliasing between z and y.

```
int qadic_log_balanced(qadic_t rop, const qadic_t op, const
   qadic_ctx_t ctx)
```

Returns whether the p-adic logarithm function converges at `op`, and if so sets `rop` to its value.

```
void _qadic_log(fmpz *z, const fmpz *y, slong v, slong len,
   const fmpz *a, const slong *j, slong lena, const fmpz_t
   p, slong N, const fmpz_t pN)
```

Computes (z, d) as

$$z = -\sum_{i=1}^{\infty} \frac{y^i}{i} \pmod{p^N}.$$

Note that this can be used to compute the p-adic logarithm via the equation

$$\log(x) = \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i}$$

$$= -\sum_{i=1}^{\infty} \frac{(1-x)^i}{i}.$$

Assumes that $y = 1 - x$ is non-zero and that $v = \mathrm{ord}_p(y)$ is at least 1 when p is odd and at least 2 when $p = 2$ so that the series converges.

Assumes that (y, d) is reduced modulo p^N.

Assumes that $v < N$, and hence in particular $N \geq 2$.

Supports aliasing between z and y.

```
int qadic_log(qadic_t rop, const qadic_t op, const
    qadic_ctx_t ctx)
```

Returns whether the p-adic logarithm function converges at `op`, and if so sets `rop` to its value.

The p-adic logarithm function is defined by the usual series

$$\log_p(x) = \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i}$$

but this only converges when $\mathrm{ord}_p(x)$ is at least 2 or 1 when $p = 2$ or $p > 2$, respectively.

```
void _qadic_frobenius_a(fmpz *rop, slong e, const fmpz *a,
    const slong *j, slong lena, const fmpz_t p, slong N)
```

Computes $\sigma^e(X) \bmod p^N$ where X is such that $\mathbf{Q}_q \cong \mathbf{Q}_p[X]/(f(X))$.

Assumes that the precision N is at least 2 and that the extension is non-trivial, i.e. $d \geq 2$.

Assumes that $0 < e < d$.

Sets `(rop, 2*d-1)`, although the actual length of the output will be at most d.

```
void _qadic_frobenius(fmpz *rop, const fmpz *op, slong len,
    slong e, const fmpz *a, const slong *j, slong lena,
    const fmpz_t p, slong N)
```

Sets `(rop, 2*d-1)` to Σ evaluated at `(op, len)`.

Assumes that `len` is positive but at most d.

Assumes that $0 < e < d$.

Does not support aliasing.

```
void qadic_frobenius(qadic_t rop, const qadic_t op, slong
    e, const qadic_ctx_t ctx)
```

Evaluates the homomorphism Σ^e at `op`.

Recall that $\mathbf{Q}_q/\mathbf{Q}_p$ is Galois with Galois group $\langle\Sigma\rangle \cong \langle\sigma\rangle$, which is also isomorphic to $\mathbf{Z}/d\mathbf{Z}$, where $\sigma \in \mathrm{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ is the Frobenius element $\sigma\colon x \mapsto x^p$ and Σ is its lift to $\mathrm{Gal}(\mathbf{Q}_q/\mathbf{Q}_p)$.

This functionality is implemented as `GaloisImage()` in Magma.

```
void _qadic_teichmuller(fmpz *rop, const fmpz *op, slong
    len, const fmpz *a, const slong *j, slong lena, const
    fmpz_t p, slong N)
```

Sets `(rop, d)` to the Teichmüller lift of `(op, len)` modulo p^N.

Does not support aliasing.

```
void qadic_teichmuller(qadic_t rop, const qadic_t op, const
   qadic_ctx_t ctx)
```

Sets `rop` to the Teichmüller lift of `op` to the precision given in the context.

For a unit `op`, this is the unique $(q-1)$th root of unity which is congruent to `op` modulo p.

Sets `rop` to zero if `op` is zero in the given context.

Raises an exception if the valuation of `op` is negative.

```
void _qadic_trace(fmpz_t rop, const fmpz *op, slong len,
   const fmpz *a, const slong *j, slong lena, const fmpz_t
   pN)
```

```
void qadic_trace(padic_t rop, const qadic_t op, const
   qadic_ctx_t ctx)
```

Sets `rop` to the trace of `op`.

For an element $a \in \mathbf{Q}_q$, multiplication by a defines a $\mathbf{Q}_p$-linear map on $\mathbf{Q}_q$. We define the trace of a as the trace of this map. Equivalently, if Σ generates $\mathrm{Gal}(\mathbf{Q}_q/\mathbf{Q}_p)$ then the trace of a is equal to $\sum_{i=0}^{d-1} \Sigma^i(a)$.

```
void _qadic_norm(fmpz_t rop, const fmpz *op, slong len,
   const fmpz *a, const slong *j, slong lena, const fmpz_t
   p, slong N)
```

Sets `rop` to the norm of the element `(op,len)` in $\mathbf{Z}_q$ to precision N, where `len` is at least one.

The result will be reduced modulo p^N.

Note that whenever `(op,len)` is a unit, so is its norm. Thus, the output `rop` of this function will typically not have to be canonicalised or reduced by the caller.

```
void qadic_norm(padic_t rop, const qadic_t op, const
   qadic_ctx_t ctx)
```

Computes the norm of `op` to the given precision.

Algorithm selection is automatic depending on the input.

```
void qadic_norm_analytic(padic_t rop, const qadic_t op,
   const qadic_ctx_t ctx)
```

Whenever `op` has valuation greater than $(p-1)^{-1}$, this routine computes its norm `rop` via

$$\mathrm{N}(x) = \exp\Big(\big(\mathrm{Tr}\log(x)\big)\Big).$$

In the special case that `op` lies in $\mathbf{Q}_p$, returns its norm as $\mathrm{N}(x) = x^d$, where d is the extension degree.

Otherwise, raises an `abort` signal.

The complexity of this implementation is quasi-linear in d and N, and polynomial in $\log p$.

```
void qadic_norm_resultant(padic_t rop, const qadic_t op,
   const qadic_ctx_t ctx)
```

Sets `rop` to the norm of `op`, using the formula

$$\mathrm{N}(x) = \ell(f)^{-\deg(a)} \operatorname{Res}(f(X), a(X)),$$

where $\mathbf{Q}_q \cong \mathbf{Q}_p[X]/(f(X))$, $\ell(f)$ is the leading coefficient of $f(X)$, and $a(X) \in mathbf{Q}_p[X]$ denotes the same polynomial as x.

The complexity of the current implementation is given by $\mathcal{O}(d^4 M(N \log p))$, where $M(n)$ denotes the complexity of multiplying to n-bit integers.

54.10 Output

```
int qadic_fprint_pretty(FILE *file, const qadic_t op, const
    qadic_ctx_t ctx)
```

Prints a pretty representation of `op` to `file`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

```
int qadic_print_pretty(const qadic_t op, const qadic_ctx_t
    ctx)
```

Prints a pretty representation of `op` to `stdout`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

§55. arith

Arithmetic functions

55.1 Introduction

This module implements arithmetic functions, number-theoretic and combinatorial special number sequences and polynomials.

55.2 Primorials

```
void arith_primorial(fmpz_t res, slong n)
```

Sets `res` to "n primorial" or $n\#$, the product of all prime numbers less than or equal to n.

55.3 Harmonic numbers

```
void _arith_harmonic_number(fmpz_t num, fmpz_t den, slong n)
```

Sets `(num, den)` to the reduced numerator and denominator of the n-th harmonic number $H_n = 1 + 1/2 + 1/3 + \cdots + 1/n$. The result is zero if $n \leq 0$.

Table lookup is used for H_n whose numerator and denominator fit in single limb. For larger n, the function `flint_mpn_harmonic_odd_balanced()` is used.

```
void arith_harmonic_number(fmpq_t x, slong n)
```

Sets `x` to the n-th harmonic number. This function is equivalent to `_arith_harmonic_number` apart from the output being a single `fmpq_t` variable.

55.4 Stirling numbers

```
void arith_stirling_number_1u(fmpz_t s, slong n, slong k)
```

```
void arith_stirling_number_1(fmpz_t s, slong n, slong k)
```

```
void arith_stirling_number_2(fmpz_t s, slong n, slong k)
```

Sets s to $S(n,k)$ where $S(n,k)$ denotes an unsigned Stirling number of the first kind $|S_1(n,k)|$, a signed Stirling number of the first kind $S_1(n,k)$, or a Stirling number of the second kind $S_2(n,k)$. The Stirling numbers are defined using the generating functions

$$x_{(n)} = \sum_{k=0}^{n} S_1(n,k) x^k$$
$$x^{(n)} = \sum_{k=0}^{n} |S_1(n,k)| x^k$$
$$x^n = \sum_{k=0}^{n} S_2(n,k) x_{(k)}$$

where $x_{(n)} = x(x-1)(x-2)\cdots(x-n+1)$ is a falling factorial and $x^{(n)} = x(x+1)(x+2)\cdots(x+n-1)$ is a rising factorial. $S(n,k)$ is taken to be zero if $n < 0$ or $k < 0$.

These three functions are useful for computing isolated Stirling numbers efficiently. To compute a range of numbers, the vector or matrix versions should generally be used.

```
void arith_stirling_number_1u_vec(fmpz * row, slong n,
    slong klen)

void arith_stirling_number_1_vec(fmpz * row, slong n, slong
    klen)

void arith_stirling_number_2_vec(fmpz * row, slong n, slong
    klen)
```

Computes the row of Stirling numbers `S(n,0), S(n,1), S(n,2), ..., S(n,klen-1)`.

To compute a full row, this function can be called with `klen = n+1`. It is assumed that `klen` is at most $n+1$.

```
void arith_stirling_number_1u_vec_next(fmpz * row, fmpz *
    prev, slong n, slong klen)

void arith_stirling_number_1_vec_next(fmpz * row, fmpz *
    prev, slong n, slong klen)

void arith_stirling_number_2_vec_next(fmpz * row, fmpz *
    prev, slong n, slong klen)
```

Given the vector `prev` containing a row of Stirling numbers `S(n-1,0), S(n-1,1), S(n-1,2), ..., S(n-1,klen-1)`, computes and stores in the row argument `S(n,0), S(n,1), S(n,2), ..., S(n,klen-1)`.

If `klen` is greater than `n`, the output ends with `S(n,n) = 1` followed by `S(n,n+1)= S(n,n+2)= ... = 0`. In this case, the input only needs to have length `n-1`; only the input entries up to `S(n-1,n-2)` are read.

The `row` and `prev` arguments are permitted to be the same, meaning that the row will be updated in-place.

```
void arith_stirling_matrix_1u(fmpz_mat_t mat)

void arith_stirling_matrix_1(fmpz_mat_t mat)

void arith_stirling_matrix_2(fmpz_mat_t mat)
```

For an arbitrary m-by-n matrix, writes the truncation of the infinite Stirling number matrix

```
row 0   : S(0,0)
row 1   : S(1,0), S(1,1)
row 2   : S(2,0), S(2,1), S(2,2)
row 3   : S(3,0), S(3,1), S(3,2), S(3,3)
```

up to row $m-1$ and column $n-1$ inclusive. The upper triangular part of the matrix is zeroed.

For any n, the S_1 and S_2 matrices thus obtained are inverses of each other.

55.5 Bell numbers

`void arith_bell_number(fmpz_t b, ulong n)`

Sets b to the Bell number B_n, defined as the number of partitions of a set with n members. Equivalently, $B_n = \sum_{k=0}^{n} S_2(n,k)$ where $S_2(n,k)$ denotes a Stirling number of the second kind.

This function automatically selects between table lookup, binary splitting, and the multimodular algorithm.

`void arith_bell_number_bsplit(fmpz_t res, ulong n)`

Computes the Bell number B_n by evaluating a precise truncation of the series $B_n = e^{-1} \sum_{k=0}^{\infty} \frac{k^n}{k!}$ using binary splitting.

`void arith_bell_number_multi_mod(fmpz_t res, ulong n)`

Computes the Bell number B_n using a multimodular algorithm.

This function evaluates the Bell number modulo several limb-size primes using `arith_bell_number_nmod` and reconstructs the integer value using the fast Chinese remainder algorithm. A bound for the number of needed primes is computed using `arith_bell_number_size`.

`void arith_bell_number_vec(fmpz * b, slong n)`

Sets b to the vector of Bell numbers $B_0, B_1, \ldots, B_{n-1}$ inclusive. Automatically switches between the `recursive` and `multi_mod` algorithms depending on the size of n.

`void arith_bell_number_vec_recursive(fmpz * b, slong n)`

Sets b to the vector of Bell numbers $B_0, B_1, \ldots, B_{n-1}$ inclusive. This function uses table lookup if B_{n-1} fits in a single word, and a standard triangular recurrence otherwise.

`void arith_bell_number_vec_multi_mod(fmpz * b, slong n)`

Sets b to the vector of Bell numbers $B_0, B_1, \ldots, B_{n-1}$ inclusive.

This function evaluates the Bell numbers modulo several limb-size primes using `arith_bell_number_nmod_vec` and reconstructs the integer values using the fast Chinese remainder algorithm. A bound for the number of needed primes is computed using `arith_bell_number_size`.

`mp_limb_t bell_number_nmod(ulong n, nmod_t mod)`

Computes the Bell number B_n modulo a prime p given by `mod`

After handling special cases, we use the formula

$$B_n = \sum_{k=0}^{n} \frac{(n-k)^n}{(n-k)!} \sum_{j=0}^{k} \frac{(-1)^j}{j!}.$$

We arrange the operations in such a way that we only have to multiply (and not divide) in the main loop. As a further optimisation, we use sieving to reduce the number of powers that need to be evaluated. This results in $O(n)$ memory usage.

The divisions by factorials require $n > p$, so we fall back to calling `bell_number_nmod_vec_recursive` and reading off the last entry when $p \leq n$.

```
void arith_bell_number_nmod_vec(mp_ptr b, slong n, nmod_t
    mod)
```

Sets b to the vector of Bell numbers $B_0, B_1, \ldots, B_{n-1}$ inclusive modulo a prime p given by `mod`. Automatically switches between the `recursive` and `series` algorithms depending on the size of n and whether p is large enough for the series algorithm to work.

```
void arith_bell_number_nmod_vec_recursive(mp_ptr b, slong
    n, nmod_t mod)
```

Sets b to the vector of Bell numbers $B_0, B_1, \ldots, B_{n-1}$ inclusive modulo a prime p given by `mod`. This function uses table lookup if B_{n-1} fits in a single word, and a standard triangular recurrence otherwise.

```
void arith_bell_number_nmod_vec_series(mp_ptr b, slong n,
    nmod_t mod)
```

Sets b to the vector of Bell numbers $B_0, B_1, \ldots, B_{n-1}$ inclusive modulo a prime p given by `mod`. This function expands the exponential generating function

$$\sum_{k=0}^{\infty} \frac{B_n}{n!} x^n = \exp(e^x - 1).$$

We require that $p \geq n$.

```
double arith_bell_number_size(ulong n)
```

Returns b such that $B_n < 2^{\lfloor b \rfloor}$, using the inequality

$$B_n < \left(\frac{0.792n}{\log(n+1)} \right)^n$$

which is given in [5].

55.6 Bernoulli numbers and polynomials

```
void _arith_bernoulli_number(fmpz_t num, fmpz_t den, ulong
    n)
```

Sets `(num, den)` to the reduced numerator and denominator of the n-th Bernoulli number. As presently implemented, this function simply calls `_arith_bernoulli_number_zeta`.

```
void arith_bernoulli_number(fmpq_t x, ulong n)
```

Sets `x` to the n-th Bernoulli number. This function is equivalent to `_arith_bernoulli_number` apart from the output being a single `fmpq_t` variable.

```
void _arith_bernoulli_number_vec(fmpz * num, fmpz * den,
    slong n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of the Bernoulli numbers $B_0, B_1, B_2, \ldots, B_{n-1}$ inclusive. This function automatically chooses between the `recursive`, `zeta` and `multi_mod` algorithms according to the size of n.

```
void arith_bernoulli_number_vec(fmpq * x, slong n)
```

Sets the `x` to the vector of Bernoulli numbers $B_0, B_1, B_2, \ldots, B_{n-1}$ inclusive. This function is equivalent to `_arith_bernoulli_number_vec` apart from the output being a single `fmpq` vector.

```
void arith_bernoulli_number_denom(fmpz_t den, ulong n)
```

Sets `den` to the reduced denominator of the n-th Bernoulli number B_n. For even n, the denominator is computed as the product of all primes p for which $p - 1$ divides n; this property is a consequence of the von Staudt-Clausen theorem. For odd n, the denominator is trivial (`den` is set to 1 whenever $B_n = 0$). The initial sequence of values smaller than 2^{32} are looked up directly from a table.

```
double arith_bernoulli_number_size(ulong n)
```

Returns b such that $|B_n| < 2^{\lfloor b \rfloor}$, using the inequality

$$|B_n| < \frac{4n!}{(2\pi)^n}$$

and $n! \leq (n+1)^{n+1} e^{-n}$. No special treatment is given to odd n. Accuracy is not guaranteed if $n > 10^{14}$.

```
void arith_bernoulli_polynomial(fmpq_poly_t poly, ulong n)
```

Sets `poly` to the Bernoulli polynomial of degree n, $B_n(x) = \sum_{k=0}^{n} \binom{n}{k} B_k x^{n-k}$ where B_k is a Bernoulli number. This function basically calls `arith_bernoulli_number_vec` and then rescales the coefficients efficiently.

```
void _arith_bernoulli_number_zeta(fmpz_t num, fmpz_t den,
    ulong n)
```

Sets `(num, den)` to the reduced numerator and denominator of the n-th Bernoulli number.

This function first computes the exact denominator and a bound for the size of the numerator. It then computes an approximation of $|B_n| = 2n!\zeta(n)/(2\pi)^n$ as a floating-point number and multiplies by the denominator to to obtain a real number that rounds to the exact numerator. For tiny n, the numerator is looked up from a table to avoid unnecessary overhead.

```
void _arith_bernoulli_number_vec_recursive(fmpz * num, fmpz
    * den, slong n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of $B_0, B_1, B_2, \ldots, B_{n-1}$ inclusive.

The first few entries are computed using `arith_bernoulli_number`, and then Ramanujan's recursive formula expressing B_m as a sum over B_k for k congruent to m modulo 6 is applied repeatedly.

To avoid costly GCDs, the numerators are transformed internally to a common denominator and all operations are performed using integer arithmetic. This makes the algorithm fast for small n, say $n < 1000$. The common denominator is calculated directly as the primorial of $n + 1$.

```
void _arith_bernoulli_number_vec_zeta(fmpz * num, fmpz *
    den, slong n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of $B_0, B_1, B_2, \ldots, B_{n-1}$ inclusive. Uses repeated direct calls to `_arith_bernoulli_number_zeta`.

```
void _arith_bernoulli_number_vec_multi_mod(fmpz * num, fmpz
    * den, slong n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of $B_0, B_1, B_2, \ldots, B_{n-1}$ inclusive. Uses the generating function

$$\frac{x^2}{\cosh(x) - 1} = \sum_{k=0}^{\infty} \frac{(2-4k)B_{2k}}{(2k)!} x^{2k}$$

which is evaluated modulo several limb-size primes using `nmod_poly` arithmetic to yield the numerators of the Bernoulli numbers after multiplication by the denominators and CRT reconstruction. This formula, given (incorrectly) in [7], saves about half of the time compared to the usual generating function $x/(e^x - 1)$ since the odd terms vanish.

55.7 Euler numbers and polynomials

Euler numbers are the integers E_n defined by

$$\frac{1}{\cosh(t)} = \sum_{n=0}^{\infty} \frac{E_n}{n!} t^n.$$

With this convention, the odd-indexed numbers are zero and the even ones alternate signs, viz. $E_0, E_1, E_2, \ldots = 1, 0, -1, 0, 5, 0, -61, 0, 1385, 0, \ldots$. The corresponding Euler polynomials are defined by

$$\frac{2e^{xt}}{e^t + 1} = \sum_{n=0}^{\infty} \frac{E_n(x)}{n!} t^n.$$

```
void arith_euler_number(fmpz_t res, ulong n)
```

Sets `res` to the Euler number E_n. Currently calls `_arith_euler_number_zeta`.

```
void arith_euler_number_vec(fmpz * res, slong n)
```

Computes the Euler numbers $E_0, E_1, \ldots, E_{n-1}$ for $n \geq 0$ and stores the result in `res`, which must be an initialised `fmpz` vector of sufficient size.

This function evaluates the even-index E_k modulo several limb-size primes using the generating function and `nmod_poly` arithmetic. A tight bound for the number of needed primes is computed using `arith_euler_number_size`, and the final integer values are recovered using balanced CRT reconstruction.

```
double arith_euler_number_size(ulong n)
```

Returns b such that $|E_n| < 2^{\lfloor b \rfloor}$, using the inequality

$$|E_n| < \frac{2^{n+2} n!}{\pi^{n+1}}$$

and $n! \leq (n+1)^{n+1} e^{-n}$. No special treatment is given to odd n. Accuracy is not guaranteed if $n > 10^{14}$.

```
void euler_polynomial(fmpq_poly_t poly, ulong n)
```

Sets `poly` to the Euler polynomial $E_n(x)$. Uses the formula

$$E_n(x) = \frac{2}{n+1}\left(B_{n+1}(x) - 2^{n+1}B_{n+1}\left(\frac{x}{2}\right)\right),$$

with the Bernoulli polynomial $B_{n+1}(x)$ evaluated once using `bernoulli_polynomial` and then rescaled.

```
void _arith_euler_number_zeta(fmpz_t res, ulong n)
```

Sets `res` to the Euler number E_n. For even n, this function uses the relation

$$|E_n| = \frac{2^{n+2}n!}{\pi^{n+1}}L(n+1)$$

where $L(n+1)$ denotes the Dirichlet L-function with character $\chi = \{0, 1, 0, -1\}$.

55.8 Legendre polynomials

```
void arith_legendre_polynomial(fmpq_poly_t poly, ulong n)
```

Sets `poly` to the n-th Legendre polynomial

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}\left[\left(x^2-1\right)^n\right].$$

The coefficients are calculated using a hypergeometric recurrence. To improve performance, the common denominator is computed in one step and the coefficients are evaluated using integer arithmetic. The denominator is given by $\gcd(n!, 2^n) = 2^{\lfloor n/2\rfloor + \lfloor n/4\rfloor + \cdots}$.

```
void arith_chebyshev_t_polynomial(fmpz_poly_t poly, ulong n)
```

Sets `poly` to the Chebyshev polynomial of the first kind $T_n(x)$, defined formally by $T_n(x) = \cos(n\cos^{-1}(x))$. The coefficients are calculated using a hypergeometric recurrence.

```
void arith_chebyshev_u_polynomial(fmpz_poly_t poly, ulong n)
```

Sets `poly` to the Chebyshev polynomial of the first kind $U_n(x)$, which satisfies $(n+1)U_n(x) = T'_{n+1}(x)$. The coefficients are calculated using a hypergeometric recurrence.

55.9 Multiplicative functions

```
void arith_euler_phi(fmpz_t res, const fmpz_t n)
```

Sets `res` to the Euler totient function $\phi(n)$, counting the number of positive integers less than or equal to n that are coprime to n.

```
int arith_moebius_mu(const fmpz_t n)
```

Computes the Moebius function $\mu(n)$, which is defined as $\mu(n) = 0$ if n has a prime factor of multiplicity greater than 1, $\mu(n) = -1$ if n has an odd number of distinct prime factors, and $\mu(n) = 1$ if n has an even number of distinct prime factors. By convention, $\mu(0) = 0$.

```
void arith_divisor_sigma(fmpz_t res, const fmpz_t n, ulong
    k)
```

Sets `res` to $\sigma_k(n)$, the sum of kth powers of all divisors of n.

`void arith_divisors(fmpz_poly_t res, const fmpz_t n)`

Set the coefficients of the polynomial `res` to the divisors of n, including 1 and n itself, in ascending order.

`void arith_ramanujan_tau(fmpz_t res, const fmpz_t n)`

Sets `res` to the Ramanujan tau function $\tau(n)$ which is the coefficient of q^n in the series expansion of $f(q) = q\prod_{k\geq 1}(1-q^k)^{24}$.

We factor n and use the identity $\tau(pq) = \tau(p)\tau(q)$ along with the recursion $\tau(p^{r+1}) = \tau(p)\tau(p^r) - p^{11}\tau(p^{r-1})$ for prime powers.

The base values $\tau(p)$ are obtained using the function `arith_ramanujan_tau_series()`. Thus the speed of `arith_ramanujan_tau()` depends on the largest prime factor of n.

Future improvement: optimise this function for small n, which could be accomplished using a lookup table or by calling `arith_ramanujan_tau_series()` directly.

`void arith_ramanujan_tau_series(fmpz_poly_t res, slong n)`

Sets `res` to the polynomial with coefficients $\tau(0), \tau(1), \ldots, \tau(n-1)$, giving the initial n terms in the series expansion of $f(q) = q\prod_{k\geq 1}(1-q^k)^{24}$.

We use the theta function identity

$$f(q) = q\left(\sum_{k\geq 0}(-1)^k(2k+1)q^{k(k+1)/2}\right)^8$$

which is evaluated using three squarings. The first squaring is done directly since the polynomial is very sparse at this point.

55.10 Cyclotomic polynomials

`void _arith_cyclotomic_polynomial(fmpz * a, ulong n, mp_ptr factors, slong num_factors, ulong phi)`

Sets `a` to the lower half of the cyclotomic polynomial $\Phi_n(x)$, given $n \geq 3$ which must be squarefree.

A precomputed array containing the prime factors of n must be provided, as well as the value of the Euler totient function $\phi(n)$ as `phi`. If n is even, 2 must be the first factor in the list.

The degree of $\Phi_n(x)$ is exactly $\phi(n)$. Only the low $(\phi(n)+1)/2$ coefficients are written; the high coefficients can be obtained afterwards by copying the low coefficients in reverse order, since $\Phi_n(x)$ is a palindrome for $n \neq 1$.

We use the sparse power series algorithm described as Algorithm 4 [3]. The algorithm is based on the identity

$$\Phi_n(x) = \prod_{d|n}(x^d-1)^{\mu(n/d)}.$$

Treating the polynomial as a power series, the multiplications and divisions can be done very cheaply using repeated additions and subtractions. The complexity is $O(2^k\phi(n))$ where k is the number of prime factors in n.

To improve efficiency for small n, we treat the `fmpz` coefficients as machine integers when there is no risk of overflow. The following bounds are given in Table 6 of [3]:

For $n < 10163195$, the largest coefficient in any $\Phi_n(x)$ has 27 bits, so machine arithmetic is safe on 32 bits.

For $n < 169828113$, the largest coefficient in any $\Phi_n(x)$ has 60 bits, so machine arithmetic is safe on 64 bits.

Further, the coefficients are always ± 1 or 0 if there are exactly two prime factors, so in this case machine arithmetic can be used as well.

Finally, we handle two special cases: if there is exactly one prime factor $n = p$, then $\Phi_n(x) = 1 + x + x^2 + \ldots + x^{n-1}$, and if $n = 2m$, we use $\Phi_n(x) = \Phi_m(-x)$ to fall back to the case when n is odd.

`void arith_cyclotomic_polynomial(fmpz_poly_t poly, ulong n)`

Sets `poly` to the nth cyclotomic polynomial, defined as

$$\Phi_n(x) = \prod_{\omega} (x - \omega)$$

where ω runs over all the nth primitive roots of unity.

We factor n into $n = qs$ where q is squarefree, and compute $\Phi_q(x)$. Then $\Phi_n(x) = \Phi_q(x^s)$.

`void _arith_cos_minpoly(fmpz * coeffs, slong d, ulong n)`

For $n \geq 1$, sets `(coeffs, d+1)` to the minimal polynomial $\Psi_n(x)$ of $\cos(2\pi/n)$, scaled to have integer coefficients by multiplying by 2^d (2^{d-1} when n is a power of two).

The polynomial $\Psi_n(x)$ is described in [28]. As proved in that paper, the roots of $\Psi_n(x)$ for $n \geq 3$ are $\cos(2\pi k/n)$ where $0 \leq k < d$ and where $\gcd(k, n) = 1$.

To calculate $\Psi_n(x)$, we compute the roots numerically with MPFR and use a balanced product tree to form a polynomial with fixed-point coefficients, i.e. an approximation of $2^p 2^d \Psi_n(x)$.

To determine the precision p, we note that the coefficients in $\prod_{i=1}^{d}(x-\alpha)$ can be bounded by the central coefficient in the binomial expansion of $(x+1)^d$.

When n is an odd prime, we use a direct formula for the coefficients (`http://mathworld.wolfram.com/TrigonometryAngles.html`).

`void arith_cos_minpoly(fmpz_poly_t poly, ulong n)`

Sets `poly` to the minimal polynomial $\Psi_n(x)$ of $\cos(2\pi/n)$, scaled to have integer coefficients. This polynomial has degree 1 if $n = 1$ or $n = 2$, and degree $\phi(n)/2$ otherwise.

We allow $n = 0$ and define $\Psi_0 = 1$.

55.11 Swinnerton-Dyer polynomials

`void arith_swinnerton_dyer_polynomial(fmpz_poly_t poly, ulong n)`

Sets `poly` to the Swinnerton-Dyer polynomial S_n, defined as the integer polynomial

$$S_n = \prod (x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \ldots \pm \sqrt{p_n})$$

where p_n denotes the n-th prime number and all combinations of signs are taken. This polynomial has degree 2^n and is irreducible over the integers.

55.12 Landau's function

```
void arith_landau_function_vec(fmpz * res, slong len)
```

Computes the first `len` values of Landau's function $g(n)$ starting with $g(0)$. Landau's function gives the largest order of an element of the symmetric group S_n.

Implements the "basic algorithm" given in [10]. The running time is $O(n^{3/2}/\sqrt{\log n})$.

55.13 Dedekind sums

Most of the definitions and relations used in the following section are given by Apostol [2]. The Dedekind sum $s(h,k)$ is defined for all integers h and k as

$$s(h,k) = \sum_{i=1}^{k-1} \left(\left(\frac{i}{k}\right)\right)\left(\left(\frac{hi}{k}\right)\right)$$

where

$$((x)) = \begin{cases} x - \lfloor x \rfloor - 1/2 & \text{if } x \in \mathbb{Q} \setminus \mathbb{Z} \\ 0 & \text{if } x \in \mathbb{Z}. \end{cases}$$

If $0 < h < k$ and $(h,k) = 1$, this reduces to

$$s(h,k) = \sum_{i=1}^{k-1} \frac{i}{k}\left(\frac{hi}{k} - \left\lfloor \frac{hi}{k} \right\rfloor - \frac{1}{2}\right).$$

The main formula for evaluating the series above is the following. Letting $r_0 = k$, $r_1 = h$, $r_2, r_3, \ldots, r_n, r_{n+1} = 1$ be the remainder sequence in the Euclidean algorithm for computing GCD of h and k,

$$s(h,k) = \frac{1-(-1)^n}{8} - \frac{1}{12}\sum_{i=1}^{n+1}(-1)^i\left(\frac{1 + r_i^2 + r_{i-1}^2}{r_i r_{i-1}}\right).$$

Writing $s(h,k) = p/q$, some useful properties employed are $|s| < k/12$, $q|6k$ and $2|p| < k^2$.

```
void arith_dedekind_sum_naive(fmpq_t s, const fmpz_t h,
    const fmpz_t k)
```

Computes $s(h,k)$ for arbitrary h and k using a straightforward implementation of the defining sum using `fmpz` arithmetic. This function is slow except for very small k and is mainly intended to be used for testing purposes.

```
double arith_dedekind_sum_coprime_d(double h, double k)
```

Returns an approximation of $s(h,k)$ computed by evaluating the remainder sequence sum using double-precision arithmetic. Assumes that $0 < h < k$ and $(h,k) = 1$, and that h, k and their remainders can be represented exactly as doubles, e.g. $k < 2^{53}$.

We give a rough error analysis with IEEE double precision arithmetic, assuming $2k^2 < 2^{53}$. By assumption, the terms in the sum evaluate exactly apart from the division. Thus each term is bounded in magnitude by $2k$ and its absolute error is bounded by $k2^{-52}$. By worst-case analysis of the Euclidean algorithm, we also know that no more than 40 terms will be added.

It follows that the absolute error is at most $Ck2^{-53}$ for some constant C. If we multiply the output by $6k$ in order to obtain an integer numerator, the order of magnitude of the error is around $6Ck^2 2^{-53}$, so rounding to the nearest integer gives a correct numerator

whenever $k < 2^{26-d}$ for some small number of guard bits d. A computation has shown that $d = 5$ is sufficient, i.e. this function can be used for exact computation when $k < 2^{21} \approx 2 \times 10^6$. This bound can likely be improved.

```
void arith_dedekind_sum_coprime_large(fmpq_t s, const
    fmpz_t h, const fmpz_t k)
```

Computes $s(h, k)$ for h and k satisfying $0 \le h \le k$ and $(h, k) = 1$. This function effectively evaluates the remainder sequence sum using `fmpz` arithmetic, without optimising for any special cases. To avoid rational arithmetic, we use the integer algorithm of Knuth [19].

```
void arith_dedekind_sum_coprime(fmpq_t s, const fmpz_t h,
    const fmpz_t k)
```

Computes $s(h, k)$ for h and k satisfying $0 \le h \le k$ and $(h, k) = 1$.

This function calls `arith_dedekind_sum_coprime_d` if k is small enough for a double-precision estimate of the sum to yield a correct numerator upon multiplication by $6k$ and rounding to the nearest integer. Otherwise, it calls `arith_dedekind_sum_coprime_large`.

```
void arith_dedekind_sum(fmpq_t s, const fmpz_t h, const
    fmpz_t k)
```

Computes $s(h, k)$ for arbitrary h and k. If the caller can guarantee $0 < h < k$ and $(h, k) = 1$ ahead of time, it is always cheaper to call `arith_dedekind_sum_coprime`.

This function uses the following identities to reduce the general case to the situation where $0 < h < k$ and $(h, k) = 1$: If $k \le 2$ or $h = 0$, $s(h, k) = 0$. If $h < 0$, $s(h, k) = -s(-h, k)$. For any $q > 0$, $s(qh, qk) = s(h, k)$. If $0 < k < h$ and $(h, k) = 1$, $s(h, k) = (1 + h(h - 3k) + k^2)/(12hk) - t(k, h)$.

55.14 Number of partitions

```
void arith_number_of_partitions_vec(fmpz * res, slong len)
```

Computes first `len` values of the partition function $p(n)$ starting with $p(0)$. Uses inversion of Euler's pentagonal series.

```
void arith_number_of_partitions_nmod_vec(mp_ptr res, slong
    len, nmod_t mod)
```

Computes first `len` values of the partition function $p(n)$ starting with $p(0)$, modulo the modulus defined by `mod`. Uses inversion of Euler's pentagonal series.

```
void arith_hrr_expsum_factored(trig_prod_t prod, mp_limb_t
    k, mp_limb_t n)
```

Symbolically evaluates the exponential sum

$$A_k(n) = \sum_{h=0}^{k-1} \exp\left(\pi i \left[s(h,k) - \frac{2hn}{k}\right]\right)$$

appearing in the Hardy-Ramanujan-Rademacher formula, where $s(h, k)$ is a Dedekind sum.

Rather than evaluating the sum naively, we factor $A_k(n)$ into a product of cosines based on the prime factorisation of k. This process is based on the identities given in [29].

The special `trig_prod_t` structure `prod` represents a product of cosines of rational arguments, multiplied by an algebraic prefactor. It must be pre-initialised with `trig_prod_init`.

This function assumes that $24k$ and $24n$ do not overflow a single limb. If n is larger, it can be pre-reduced modulo k, since $A_k(n)$ only depends on the value of n mod k.

`void arith_number_of_partitions_mpfr(mpfr_t x, ulong n)`

Sets the pre-initialised MPFR variable x to the exact value of $p(n)$. The value is computed using the Hardy-Ramanujan-Rademacher formula.

The precision of x will be changed to allow $p(n)$ to be represented exactly. The interface of this function may be updated in the future to allow computing an approximation of $p(n)$ to smaller precision.

The Hardy-Ramanujan-Rademacher formula is given with error bounds in [25]. We evaluate it in the form

$$p(n) = \sum_{k=1}^{N} B_k(n) U(C/k) + R(n, N)$$

where

$$U(x) = \cosh(x) + \frac{\sinh(x)}{x}, \quad C = \frac{\pi}{6}\sqrt{24n-1}$$

$$B_k(n) = \sqrt{\frac{3}{k}} \frac{4}{24n-1} A_k(n)$$

and where $A_k(n)$ is a certain exponential sum. The remainder satisfies

$$|R(n,N)| < \frac{44\pi^2}{225\sqrt{3}} N^{-1/2} + \frac{\pi\sqrt{2}}{75} \left(\frac{N}{n-1}\right)^{1/2} \sinh\left(\pi\sqrt{\frac{2}{3}}\frac{\sqrt{n}}{N}\right).$$

We choose N such that $|R(n, N)| < 0.25$, and a working precision at term k such that the absolute error of the term is expected to be less than $0.25/N$. We also use a summation variable with increased precision, essentially making additions exact. Thus the sum of errors adds up to less than 0.5, giving the correct value of $p(n)$ when rounding to the nearest integer.

The remainder estimate at step k provides an upper bound for the size of the k-th term. We add $\log_2 N$ bits to get low bits in the terms below $0.25/N$ in magnitude.

Using `arith_hrr_expsum_factored`, each $B_k(n)$ evaluation is broken down to a product of cosines of exact rational multiples of π. We transform all angles to $(0, \pi/4)$ for optimal accuracy.

Since the evaluation of each term involves only $O(\log k)$ multiplications and evaluations of trigonometric functions of small angles, the relative rounding error is at most a few bits. We therefore just add an additional $\log_2(C/k)$ bits for the $U(x)$ when x is large. The cancellation of terms in $U(x)$ is of no concern, since Rademacher's bound allows us to terminate before x becomes small.

This analysis should be performed in more detail to give a rigorous error bound, but the precision currently implemented is almost certainly sufficient, not least considering that Rademacher's remainder bound significantly overshoots the actual values.

To improve performance, we switch to doubles when the working precision becomes small enough. We also use a separate accumulator variable which gets added to the main sum periodically, in order to avoid costly updates of the full-precision result when n is large.

`void arith_number_of_partitions(fmpz_t x, ulong n)`

Sets x to $p(n)$, the number of ways that n can be written as a sum of positive integers without regard to order.

This function uses a lookup table for $n < 128$ (where $p(n) < 2^{32}$), and otherwise calls `arith_number_of_partitions_mpfr`.

55.15 Sums of squares

`void arith_sum_of_squares(fmpz_t r, ulong k, const fmpz_t n)`

Sets r to the number of ways $r_k(n)$ in which n can be represented as a sum of k squares.

If $k = 2$ or $k = 4$, we write $r_k(n)$ as a divisor sum.

Otherwise, we either recurse on k or compute the theta function expansion up to $O(x^{n+1})$ and read off the last coefficient. This is generally optimal.

`void arith_sum_of_squares_vec(fmpz * r, ulong k, slong n)`

For $i = 0, 1, \ldots, n-1$, sets r_i to the number of representations of i a sum of k squares, $r_k(i)$. This effectively computes the q-expansion of $\vartheta_3(q)$ raised to the kth power, i.e.

$$\vartheta_3^k(q) = \left(\sum_{i=-\infty}^{\infty} q^{i^2} \right)^k.$$

55.16 MPFR extras

`void mpfr_pi_chudnovsky(mpfr_t x, mpfr_rnd_t rnd)`

Sets `x` to π, rounded in the direction `rnd`.

Uses the Chudnovsky algorithm, which typically is about four times faster than the MPFR default function. As currently implemented, the value is not cached for repeated use.

§56. ulong_extras

Unsigned single limb arithmetic

56.1 Introduction

This module implements functions for single limb unsigned integers, including arithmetic with a precomputed inverse and modular arithmetic.

The module includes functions for square roots, factorisation and primality testing. Almost all the functions in this module are highly developed and extremely well optimised.

The basic type is the `mp_limb_t` as defined by MPIR. Functions which take a precomputed inverse either have the suffix `preinv` and take an `mp_limb_t` precomputed inverse as computed by `n_preinvert_limb` or have the suffix `_precomp` and accept a `double` precomputed inverse as computed by `n_precompute_inverse`.

Sometimes three functions with similar names are provided for the same task, e.g. `n_mod_precomp`, `n_mod2_precomp` and `n_mod2_preinv`. If the part of the name that designates the functionality ends in 2 then the function has few if any limitations on its inputs. Otherwise the function may have limitations such as being limited to 52 or 53 bits. In practice we found that the preinv functions are generally faster anyway, so most times it pays to just use the `n_blah2_preinv` variants.

Some functions with the `n_ll_` or `n_lll_` prefix accept parameters of two or three limbs respectively.

56.2 Simple example

The following example computes $ab \pmod{n}$ using a precomputed inverse, where $a = 12345678, b = 87654321$ and $n = 111111111$.

```
#include <stdio.h>
#include "ulong_extras.h"
...
mp_limb_t r, a, b, n, ninv;

a = UWORD(12345678);
b = UWORD(87654321);
n = UWORD(111111111);
ninv = n_preinvert_limb(n);
```

```
r = n_mulmod2_preinv(a, b, n, ninv);

flint_printf("%wu*%wu mod %wu is %wu\n", a, b, n, r);
```

The output is:

```
12345678*87654321 mod 111111111 is 23456790
```

56.3 Random functions

`void n_randinit(flint_rand_t state)`

Initialise a random state for use in random functions.

`void n_randclear(flint_rand_t state)`

Release any memory used by a random state.

`mp_limb_t n_randlimb(flint_rand_t state)`

Returns a uniformly pseudo random limb.

The algorithm generates two random half limbs s_j, $j = 0, 1$, by iterating respectively $v_{i+1} = (v_i a + b) \bmod p_j$ for some initial seed v_0, randomly chosen values a and b and `p_0 = 4294967311 = nextprime(2^32)` on a 64-bit machine and `p_0 = nextprime(2^16)` on a 32-bit machine and `p_1 = nextprime(p_0)`.

`mp_limb_t n_randbits(flint_rand_t state, unsigned int bits)`

Returns a uniformly pseudo random number with the given number of bits. The most significant bit is always set, unless zero is passed, in which case zero is returned.

`mp_limb_t n_randtest_bits(flint_rand_t state, int bits)`

Returns a uniformly pseudo random number with the given number of bits. The most significant bit is always set, unless zero is passed, in which case zero is returned. The probability of a value with a sparse binary representation being returned is increased. This function is intended for use in test code.

`mp_limb_t n_randint(flint_rand_t state, mp_limb_t limit)`

Returns a uniformly pseudo random number up to but not including the given limit. If zero is passed as a parameter, an entire random limb is returned.

`mp_limb_t n_randtest(flint_rand_t state)`

Returns a pseudo random number with a random number of bits, from 0 to `FLINT_BITS`. The probability of the special values 0, 1, `COEFF_MAX` and `WORD_MAX` is increased as is the probability of a value with sparse binary representation. This random function is mainly used for testing purposes. This function is intended for use in test code.

`mp_limb_t n_randtest_not_zero(flint_rand_t state)`

As for `n_randtest()`, but does not return 0. This function is intended for use in test code.

`mp_limb_t n_randprime(flint_rand_t state, unsigned slong bits, int proved)`

Returns a random prime number (proved = 1) or probable prime (proved = 0) with `bits` bits, where `bits` must be at least 2 and at most `FLINT_BITS`.

`mp_limb_t n_randtest_prime(flint_rand_t state, int proved)`

Returns a random prime number (proved = 1) or probable prime (proved = 0) with size randomly chosen between 2 and `FLINT_BITS` bits. This function is intended for use in test code.

56.4 Basic arithmetic

`mp_limb_t n_pow(mp_limb_t n, ulong exp)`

Returns `n^exp`. No checking is done for overflow. The exponent may be zero. We define $0^0 = 1$.

The algorithm simply uses a for loop. Repeated squaring is unlikely to speed up this algorithm.

`mp_limb_t n_flog(mp_limb_t n, mp_limb_t b)`

Returns $\lfloor \log_b x \rfloor$.

Assumes that $x \geq 1$ and $b \geq 2$.

`mp_limb_t n_clog(mp_limb_t n, mp_limb_t b)`

Returns $\lceil \log_b x \rceil$.

Assumes that $x \geq 1$ and $b \geq 2$.

56.5 Miscellaneous

`ulong n_revbin(ulong in, ulong bits)`

Returns the binary reverse of `in`, assuming it is the given number of bits in length, e.g. `n_revbin(10110, 6)` will return `110100`.

`int n_sizeinbase(mp_limb_t n, int base)`

Returns the exact number of digits needed to represent n as a string in base `base` assumed to be between 2 and 36. Returns 1 when $n = 0$.

56.6 Basic arithmetic with precomputed inverses

`mp_limb_t n_mod_precomp(mp_limb_t a, mp_limb_t n, double ninv)`

Returns a mod n given a precomputed inverse of n computed by `n_precompute_inverse()`. We require `n < 2^FLINT_D_BITS` and `a < 2^(FLINT_BITS-1)` and $0 \leq a < n^2$.

We assume the processor is in the standard round to nearest mode. Thus `ninv` is correct to 53 binary bits, the least significant bit of which we shall call a place, and can be at most half a place out. When a is multiplied by $ninv$, the binary representation of a is exact and the mantissa is less than 2, thus we see that `a * ninv` can be at most one out in the mantissa. We now truncate `a * ninv` to the nearest integer, which is always a round down. Either we already have an integer, or we need to make a change down of at least 1 in the last place. In the latter case we either get precisely the exact quotient or below it as when we rounded the product to the nearest place we changed by at most half a place. In the case that truncating to an integer takes us below the exact quotient, we have rounded down by less than 1 plus half a place. But as the product is less than n and n is less than 2^{53}, half a place is less than 1, thus we are out by less than 2 from the exact quotient, i.e. the quotient we have computed is the quotient we are after or one too

small. That leaves only the case where we had to round up to the nearest place which happened to be an integer, so that truncating to an integer didn't change anything. But this implies that the exact quotient a/n is less than 2^{-54} from an integer. We deal with this rare case by subtracting 1 from the quotient. Then the quotient we have computed is either exactly what we are after, or one too small.

```
mp_limb_t n_mod2_precomp(mp_limb_t a, mp_limb_t n, double
   ninv)
```

Returns a mod n given a precomputed inverse of n computed by `n_precompute_inverse()`. There are no restrictions on a or on n.

As for `n_mod_precomp()` for $n < 2^{53}$ and $a < n^2$ the computed quotient is either what we are after or one too large or small. We deal with these cases. Otherwise we can be sure that the top 52 bits of the quotient are computed correctly. We take the remainder and adjust the quotient by multiplying the remainder by `ninv` to compute another approximate quotient as per `mod_precomp`. Now the remainder may be either negative or positive, so the quotient we compute may be one out in either direction.

```
mp_limb_t n_mod2_preinv(mp_limb_t a, mp_limb_t n, mp_limb_t
   ninv)
```

Returns a mod n given a precomputed inverse of n computed by `n_preinvert_limb()`. There are no restrictions on a or on n.

The old version of this function was implemented simply by making use of `udiv_qrnnd_preinv()`.

The new version uses the new algorithm of Granlund and Möller [13]. First n is normalised and a shifted into two limbs to compensate. Then their algorithm is applied verbatim and the result shifted back.

```
mp_limb_t n_divrem2_precomp(mp_limb_t * q, mp_limb_t a,
   mp_limb_t n, double npre)
```

Returns a mod n given a precomputed inverse of n computed by `n_precompute_inverse()` and sets q to the quotient. There are no restrictions on a or on n.

This is as for `n_mod2_precomp()` with some additional care taken to retain the quotient information. There are also special cases to deal with the case where a is already reduced modulo n and where n is 64 bits and a is not reduced modulo n.

```
mp_limb_t n_ll_mod_preinv(mp_limb_t a_hi, mp_limb_t a_lo,
   mp_limb_t n, mp_limb_t ninv)
```

Returns a mod n given a precomputed inverse of n computed by `n_preinvert_limb()`. There are no restrictions on a, which will be two limbs `(a_hi, a_lo)`, or on n.

The old version of this function merely reduced the top limb `a_hi` modulo n so that `udiv_qrnnd_preinv()` could be used.

The new version reduces the top limb modulo n as per `n_mod2_preinv()` and then the algorithm of Granlund and Möller [13] is used again to reduce modulo n.

```
mp_limb_t n_lll_mod_preinv(mp_limb_t a_hi, mp_limb_t a_mi,
   mp_limb_t a_lo, mp_limb_t n, mp_limb_t ninv)
```

Returns a mod n, where a has three limbs `(a_hi, a_mi, a_lo)`, given a precomputed inverse of n computed by `n_preinvert_limb()`. It is assumed that `a_hi` is reduced modulo n. There are no restrictions on n.

This function uses the algorithm of Granlund and Möller [13] to first reduce the top two limbs modulo n, then does the same on the bottom two limbs.

```
mp_limb_t n_mulmod_precomp(mp_limb_t a, mp_limb_t b,
   mp_limb_t n, double ninv)
```

Returns ab mod n given a precomputed inverse of n computed by `n_precompute_inverse()`. We require `n < 2^FLINT_D_BITS` and $0 \leq a, b < n$.

We assume the processor is in the standard round to nearest mode. Thus `ninv` is correct to 53 binary bits, the least significant bit of which we shall call a place, and can be at most half a place out. The product of a and b is computed with error at most half a place. When `a * b` is multiplied by $ninv$ we find that the exact quotient and computed quotient differ by less than two places. As the quotient is less than n this means that the exact quotient is at most 1 away from the computed quotient. We truncate this quotient to an integer which reduces the value by less than 1. We end up with a value which can be no more than two above the quotient we are after and no less than two below. However an argument similar to that for `n_mod_precomp()` shows that the truncated computed quotient cannot be two smaller than the truncated exact quotient. In other words the computed integer quotient is at most two above and one below the quotient we are after.

```
mp_limb_t n_mulmod2_preinv(mp_limb_t a, mp_limb_t b,
   mp_limb_t n, mp_limb_t ninv)
```

Returns ab mod n given a precomputed inverse of n computed by `n_preinvert_limb()`. There are no restrictions on a, b or on n. This is implemented by multiplying using `umul_ppmm()` and then reducing using `n_ll_mod_preinv()`.

```
mp_limb_t n_mulmod_preinv(mp_limb_t a, mp_limb_t b,
   mp_limb_t n, mp_limb_t ninv, ulong norm)
```

Returns $ab \pmod{n}$ given a precomputed inverse of n computed by `n_preinvert_limb()`, assuming a and b are reduced modulo n and n is normalised, i.e. with most significant bit set. There are no other restrictions on a, b or n.

The value `norm` is provided for convenience. As n is required to be normalised, it may be that a and b have been shifted to the left by `norm` bits before calling the function. Their product then has an extra factor of 2^{norm}. Specifying a nonzero `norm` will shift the product right by this many bits before reducing it.

The algorithm use is that of Granlund and Möller [13].

56.7 Greatest common divisor

```
mp_limb_t n_gcd(mp_limb_t x, mp_limb_t y)
```

Returns the greatest common divisor g of x and y. We require $x \geq y$.

The algorithm is a slight embelishment of the Euclidean algorithm which uses some branches to avoid most divisions.

One wishes to compute the quotient and remainder of u_3/v_3 without division where possible. This is accomplished when $u_3 < 4v_3$, i.e. the quotient is either 1, 2 or 3.

We first compute $s = u_3 - v_3$. If $s < v_3$, i.e. $u_3 < 2v_3$, we know the quotient is 1, else if $s < 2v_3$, i.e. $u_3 < 3v_3$ we know the quotient is 2. In the remaining cases, the quotient must be 3. When the quotient is 4 or above, we use division. However this happens rarely for generic inputs.

```
mp_limb_t n_gcd_full(mp_limb_t x, mp_limb_t y)
```

Returns the greatest common divisor g of x and y. No assumptions are made about x and y.

```
mp_limb_t n_gcdinv(mp_limb_t * a, mp_limb_t x, mp_limb_t y)
```

Returns the greatest common divisor g of x and y and computes a such that $0 \leq a < y$ and $ax = \gcd(x, y) \bmod y$, when this is defined. We require $0 \leq x < y$.

This is merely an adaption of the extended Euclidean algorithm with appropriate normalisation.

```
mp_limb_t n_xgcd(mp_limb_t * a, mp_limb_t * b, mp_limb_t x,
   mp_limb_t y)
```

Returns the greatest common divisor g of x and y and unsigned values a and b such that $ax - by = g$. We require $x \geq y$.

We claim that computing the extended greatest common divisor via the Euclidean algorithm always results in cofactor $|a| < x/2$, $|b| < x/2$, with perhaps some small degenerate exceptions.

We proceed by induction.

Suppose we are at some step of the algorithm, with $x_n = qy_n + r$ with $r \geq 1$, and suppose $1 = sy_n - tr$ with $s < y_n/2$, $t < y_n/2$ by hypothesis.

Write $1 = sy_n - t(x_n - qy_n) = (s + tq)y_n - tx_n$.

It suffices to show that $(s + tq) < x_n/2$ as $t < y_n/2 < x_n/2$, which will complete the induction step.

But at the previous step in the backsubstitution we would have had $1 = sr - cd$ with $s < r/2$ and $c < r/2$.

Then $s + tq < r/2 + y_n/2q = (r + qy_n)/2 = x_n/2$.

See the documentation of `n_gcd()` for a description of the branching in the algorithm, which is faster than using division.

56.8 Jacobi and Kronecker symbols

```
int n_jacobi(mp_limb_signed_t x, mp_limb_t y)
```

Computes the Jacobi symbol of $x \bmod y$. Assumes that y is positive and odd, and for performance reasons that $\gcd(x, y) = 1$.

This is just a straightforward application of the law of quadratic reciprocity. For performance, divisions are replaced with some comparisons and subtractions where possible.

```
int n_jacobi_unsigned(mp_limb_t x, mp_limb_t y)
```

Computes the Jacobi symbol, allowing x to go up to a full limb.

56.9 Modular Arithmetic

```
mp_limb_t n_addmod(mp_limb_t a, mp_limb_t b, mp_limb_t n)
```

Returns $(a + b) \bmod n$.

```
mp_limb_t n_submod(mp_limb_t a, mp_limb_t b, mp_limb_t n)
```

Returns $(a - b) \bmod n$.

```
mp_limb_t n_invmod(mp_limb_t x, mp_limb_t y)
```

Returns a value a such that $0 \leq a < y$ and $ax = \gcd(x, y) \bmod y$, when this is defined. We require $0 \leq x < y$.

Specifically, when x is coprime to y, a is the inverse of x in $\mathbf{Z}/y\mathbf{Z}$.

This is merely an adaption of the extended Euclidean algorithm with appropriate normalisation.

```
mp_limb_t n_powmod_precomp(mp_limb_t a, mp_limb_signed_t
   exp, mp_limb_t n, double npre)
```

Returns `a^exp` modulo n given a precomputed inverse of n computed by `n_precompute_inverse()`. We require $n < 2^{53}$ and $0 \leq a < n$. There are no restrictions on `exp`, i.e. it can be negative.

This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo n at each step.

```
mp_limb_t n_powmod_ui_precomp(mp_limb_t a, mp_limb_t exp,
   mp_limb_t n, double npre)
```

Returns `a^exp` modulo n given a precomputed inverse of n computed by `n_precompute_inverse()`. We require $n < 2^{53}$ and $0 \leq a < n$. The exponent `exp` is unsigned and so can be larger than allowed by `n_powmod_precomp`.

This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo n at each step.

```
mp_limb_t n_powmod(mp_limb_t a, mp_limb_signed_t exp,
   mp_limb_t n)
```

Returns `a^exp` modulo n. We require `n < 2^FLINT_D_BITS` and $0 \leq a < n$. There are no restrictions on `exp`, i.e. it can be negative.

This is implemented by precomputing an inverse and calling the `precomp` version of this function.

```
mp_limb_t n_powmod2_preinv(mp_limb_t a, mp_limb_signed_t
   exp, mp_limb_t n, mp_limb_t ninv)
```

Returns `(a^exp) % n` given a precomputed inverse of n computed by `n_preinvert_limb()`. We require $0 \leq a < n$, but there are no restrictions on n or on `exp`, i.e. it can be negative.

This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo n at each step.

```
mp_limb_t n_powmod2(mp_limb_t a, mp_limb_signed_t exp,
   mp_limb_t n)
```

Returns `(a^exp) % n`. We require $0 \leq a < n$, but there are no restrictions on n or on `exp`, i.e. it can be negative.

This is implemented by precomputing an inverse limb and calling the `preinv` version of this function.

```
mp_limb_t n_powmod2_ui_preinv(mp_limb_t a, mp_limb_t exp,
   mp_limb_t n, mp_limb_t ninv)
```

Returns `(a^exp) % n` given a precomputed inverse of n computed by `n_preinvert_limb()`. We require $0 \leq a < n$, but there are no restrictions on n. The exponent `exp` is unsigned and so can be larger than allowed by `n_powmod2_preinv`.

This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo n at each step.

```
mp_limb_t n_sqrtmod(mp_limb_t a, mp_limb_t p)
```

Computes a square root of a modulo p.

Assumes that p is a prime and that a is reduced modulo p. Returns 0 if a is a quadratic non-residue modulo p.

```
slong n_sqrtmod_2pow(mp_limb_t ** sqrt, mp_limb_t a, slong
   exp)
```

Computes all the square roots of `a` modulo `2^exp`. The roots are stored in an array which is created and whose address is stored in the location pointed to by `sqrt`. The array of roots is allocated by the function but must be cleaned up by the user by calling `flint_free`. The number of roots is returned by the function. If `a` is not a quadratic residue modulo `2^exp` then 0 is returned by the function and the location `sqrt` points to is set to NULL.

```
slong n_sqrtmod_primepow(mp_limb_t ** sqrt, mp_limb_t a,
   mp_limb_t p, slong exp)
```

Computes all the square roots of `a` modulo `p^exp`. The roots are stored in an array which is created and whose address is stored in the location pointed to by `sqrt`. The array of roots is allocated by the function but must be cleaned up by the user by calling `flint_free`. The number of roots is returned by the function. If `a` is not a quadratic residue modulo `p^exp` then 0 is returned by the function and the location `sqrt` points to is set to NULL.

```
slong n_sqrtmodn(mp_limb_t ** sqrt, mp_limb_t a, n_factor_t
   * fac)
```

Computes all the square roots of `a` modulo `m` given the factorisation of `m` in `fac`. The roots are stored in an array which is created and whose address is stored in the location pointed to by `sqrt`. The array of roots is allocated by the function but must be cleaned up by the user by calling `flint_free`. The number of roots is returned by the function. If `a` is not a quadratic residue modulo `m` then 0 is returned by the function and the location `sqrt` points to is set to NULL.

56.10 Prime number generation and counting

```
void n_primes_init(n_primes_t iter)
```

Initialises the prime number iterator `iter` for use.

```
void n_primes_clear(n_primes_t iter)
```

Clears memory allocated by the prime number iterator `iter`.

```
mp_limb_t n_primes_next(n_primes_t iter)
```

Returns the next prime number and advances the state of `iter`. The first call returns 2.

Small primes are looked up from `flint_small_primes`. When this table is exhausted, primes are generated in blocks by calling `n_primes_sieve_range`.

```
void n_primes_jump_after(n_primes_t iter, mp_limb_t n)
```

Changes the state of `iter` to start generating primes after n (excluding n itself).

```
void n_primes_extend_small(n_primes_t iter, mp_limb_t bound)
```

Extends the table of small primes in `iter` to contain at least two primes larger than or equal to `bound`.

```
void n_primes_sieve_range(n_primes_t iter, mp_limb_t a,
    mp_limb_t b)
```

Sets the block endpoints of `iter` to the smallest and largest odd numbers between a and b inclusive, and sieves to mark all odd primes in this range. The iterator state is changed to point to the first number in the sieved range.

```
void n_compute_primes(ulong num_primes)
```

Precomputes at least `num_primes` primes and their `double` precomputed inverses and stores them in an internal cache. Assuming that FLINT has been built with support for thread-local storage, each thread has its own cache.

```
const mp_limb_t * n_primes_arr_readonly(ulong num_primes)
```

Returns a pointer to a read-only array of the first `num_primes` prime numbers. The computed primes are cached for repeated calls. The pointer is valid until the user calls `n_cleanup_primes` in the same thread.

```
const double * n_prime_inverses_arr_readonly(ulong n)
```

Returns a pointer to a read-only array of inverses of the first `num_primes` prime numbers. The computed primes are cached for repeated calls. The pointer is valid until the user calls `n_cleanup_primes` in the same thread.

```
void n_cleanup_primes()
```

Frees the internal cache of prime numbers used by the current thread. This will invalidate any pointers returned by `n_primes_arr_readonly` or `n_prime_inverses_arr_readonly`.

```
mp_limb_t n_nextprime(mp_limb_t n, int proved)
```

Returns the next prime after n. Assumes the result will fit in an `mp_limb_t`. If proved is 0, i.e. false, the prime is not proven prime, otherwise it is.

```
ulong n_prime_pi(mp_limb_t n)
```

Returns the value of the prime counting function $\pi(n)$, i.e. the number of primes less than or equal to n. The invariant `n_prime_pi(n_nth_prime(n))== n`.

Currently, this function simply extends the table of cached primes up to an upper limit and then performs a binary search.

```
void n_prime_pi_bounds(ulong *lo, ulong *hi, mp_limb_t n)
```

Calculates lower and upper bounds for the value of the prime counting function `lo <= pi(n)<= hi`. If `lo` and `hi` point to the same location, the high value will be stored.

The upper approximation is $1.25506n/\ln n$, and the lower is $n/\ln n$. These bounds are due to Rosser and Schoenfeld [26] and valid for $n \geq 17$.

We use the number of bits in n (or one less) to form an approximation to $\ln n$, taking care to use a value too small or too large to maintain the inequality.

```
mp_limb_t n_nth_prime(ulong n)
```

Returns the nth prime number p_n, using the mathematical indexing convention $p_1 = 2, p_2 = 3, \ldots$.

This function simply ensures that the table of cached primes is large enough and then looks up the entry.

```
void n_nth_prime_bounds(mp_limb_t *lo, mp_limb_t *hi, ulong
    n)
```

Calculates lower and upper bounds for the nth prime number p_n, `lo <= p_n <= hi`. If `lo` and `hi` point to the same location, the high value will be stored. Note that this function will overflow for sufficiently large n.

We use the following estimates, valid for $n > 5$:

$$p_n > n(\ln n + \ln\ln n - 1)$$
$$p_n < n(\ln n + \ln\ln n)$$
$$p_n < n(\ln n + \ln\ln n - 0.9427) \quad (n \geq 15985)$$

The first inequality was proved by Dusart [11], and the last is due to Massias and Robin [22]. For a further overview, see http://primes.utm.edu/howmany.shtml.

We bound $\ln n$ using the number of bits in n as in `n_prime_pi_bounds()`, and estimate $\ln\ln n$ to the nearest integer; this function is nearly constant.

56.11 Primality testing

`int n_is_oddprime_small(mp_limb_t n)`

Returns 1 if n is an odd prime smaller than `FLINT_ODDPRIME_SMALL_CUTOFF`. Expects n to be odd and smaller than the cutoff.

This function merely uses a lookup table with one bit allocated for each odd number up to the cutoff.

`int n_is_oddprime_binary(mp_limb_t n)`

This function performs a simple binary search through the table of cached primes for n. If it exists in the array it returns 1, otherwise 0. For the algorithm to operate correctly n should be odd and at least 17.

Lower and upper bounds are computed with `n_prime_pi_bounds()`. Once we have bounds on where to look in the table, we refine our search with a simple binary algorithm, taking the top or bottom of the current interval as necessary.

`int n_is_prime_pocklington(mp_limb_t n, ulong iterations)`

Tests if n is a prime using the Pocklington–Lehmer primality test. If 1 is returned n has been proved prime. If 0 is returned n is composite. However -1 may be returned if nothing was proved either way due to the number of iterations being too small.

The most time consuming part of the algorithm is factoring $n - 1$. For this reason `n_factor_partial()` is used, which uses a combination of trial factoring and Hart's one line factor algorithm [16] to try to quickly factor $n - 1$. Additionally if the cofactor is less than the square root of $n - 1$ the algorithm can still proceed.

One can also specify a number of iterations if less time should be taken. Simply set this to `~WORD(0)` if this is irrelevant. In most cases a greater number of iterations will not significantly affect timings as most of the time is spent factoring.

See http://mathworld.wolfram.com/PocklingtonsTheorem.html for a description of the algorithm.

`int n_is_prime_pseudosquare(mp_limb_t n)`

Tests if n is a prime according to [21, Theorem 2.7].

We first factor N using trial division up to some limit B. In fact, the number of primes used in the trial factoring is at most `FLINT_PSEUDOSQUARES_CUTOFF`.

Next we compute N/B and find the next pseudosquare L_p above this value, using a static table as per http://research.att.com/~njas/sequences/b002189.txt.

As noted in the text, if p is prime then Step 3 will pass. This test rejects many composites, and so by this time we suspect that p is prime. If N is 3 or 7 modulo 8, we are done, and N is prime.

We now run a probable prime test, for which no known counterexamples are known, to reject any composites. We then proceed to prove N prime by executing Step 4. In the case that N is 1 modulo 8, if Step 4 fails, we extend the number of primes p_i at Step 3 and hope to find one which passes Step 4. We take the test one past the largest p for which we have pseudosquares L_p tabulated, as this already corresponds to the next L_p which is bigger than 2^{64} and hence larger than any prime we might be testing.

As explained in the text, Condition 4 cannot fail if N is prime.

The possibility exists that the probable prime test declares a composite prime. However in that case an error is printed, as that would be of independent interest.

```
int n_is_prime(mp_limb_t n)
```

Tests if n is a prime. This first sieves for small prime factors, then simply calls `n_is_probabprime()`. This has been checked against the tables of Feitsma and Galway http://www.cecm.sfu.ca/Pseudoprimes/index-2-to-64.html and thus constitutes a check for primality (rather than just pseudoprimality) up to 2^64.

In future, this test may produce and check a certificate of primality. This is likely to be significantly slower for prime inputs.

```
int n_is_strong_probabprime_precomp(mp_limb_t n, double
    npre, mp_limb_t a, mp_limb_t d)
```

Tests if n is a strong probable prime to the base a. We require that d is set to the largest odd factor of $n-1$ and `npre` is a precomputed inverse of n computed with `n_precompute_inverse()`. We also require that $n < 2^{53}$, a to be reduced modulo n and not 0 and n to be odd.

If we write $n - 1 = 2^s d$ where d is odd then n is a strong probable prime to the base a, i.e. an a-SPRP, if either $a^d = 1 \pmod{n}$ or $(a^d)^{2^r} = -1 \pmod{n}$ for some r less than s.

A description of strong probable primes is given here: http://mathworld.wolfram.com/StrongPseudoprime.html

```
int n_is_strong_probabprime2_preinv(mp_limb_t n, mp_limb_t
    ninv, mp_limb_t a, mp_limb_t d)
```

Tests if n is a strong probable prime to the base a. We require that d is set to the largest odd factor of $n-1$ and `npre` is a precomputed inverse of n computed with `n_preinvert_limb()`. We require a to be reduced modulo n and not 0 and n to be odd.

If we write $n - 1 = 2^s d$ where d is odd then n is a strong probable prime to the base a (an a-SPRP) if either $a^d = 1 \pmod{n}$ or $(a^d)^{2^r} = -1 \pmod{n}$ for some r less than s.

A description of strong probable primes is given here: http://mathworld.wolfram.com/StrongPseudoprime.html

```
int n_is_probabprime_fermat(mp_limb_t n, mp_limb_t i)
```

Returns 1 if n is a base i Fermat probable prime. Requires $1 < i < n$ and that i does not divide n.

By Fermat's Little Theorem if i^{n-1} is not congruent to 1 then n is not prime.

```
int n_is_probabprime_fibonacci(mp_limb_t n)
```

Let F_j be the jth element of the Fibonacci sequence $0, 1, 1, 2, 3, 5, \ldots$, starting at $j = 0$. Then if n is prime we have $F_{n-(n/5)} = 0 \pmod{n}$, where $(n/5)$ is the Jacobi symbol.

For further details, see [9, pp. 142].

We require that n is not divisible by 2 or 5.

`int n_is_probabprime_BPSW(mp_limb_t n)`

Implements a Baillie–Pomerance–Selfridge–Wagstaff probable primality test. This is a variant of the usual BPSW test (which only uses Fermat and Lucas tests, see Baillie and Wagstaff [4]) making use of the comment on page 143 of [9] regarding Fibonacci pseudoprimes.

There are no known counterexamples to this being a primality test.

Up to 2^64 the test we use has been checked against tables of pseudoprimes. Thus it is a primality test up to this limit.

`int n_is_probabprime_lucas(mp_limb_t n)`

For details on Lucas pseudoprimes, see [9, pp. 143].

We implement a variant of the Lucas pseudoprime test similar to that described by Baillie and Wagstaff [4].

`int n_is_probabprime(mp_limb_t n)`

Tests if n is a probable prime. Up to `FLINT_ODDPRIME_SMALL_CUTOFF` this algorithm uses `n_is_oddprime_small()` which uses a lookup table.

Next it calls `n_compute_primes()` with the maximum table size and uses this table to perform a binary search for n up to the table limit.

Then up to 1050535501 it uses a number of strong probable prime tests, `n_is_strong_probabprime_preinv()`, etc., for various bases. The output of the algorithm is guaranteed to be correct up to this bound due to exhaustive tables, described at http://uucode.com/obf/dalbec/alg.html.

Beyond that point the BPSW probabilistic primality test is used, by calling the function `n_is_probabprime_BPSW()`. There are no known counterexamples, and it has been checked against the tables of Feitsma and Galway and up to the accuracy of those tables, this is an exhaustive check up to 2^64, i.e. there are no counterexamples.

56.12 Square root and perfect power testing

`mp_limb_t n_sqrt(mp_limb_t a)`

Computes the integer truncation of the square root of a. The integer itself can be represented exactly as a double and its square root is computed to the nearest place. If a is one below a square, the rounding may be up, whereas if it is one above a square, the rounding will be down. Thus the square root may be one too large in some instances. We also have to be careful when the square of this too large value causes an overflow. The same assumptions hold for a single precision float provided the square root itself can be represented in a single float, i.e. for $a < 281474976710656 = 2^{46}$.

`mp_limb_t n_sqrtrem(mp_limb_t * r, mp_limb_t a)`

Computes the integer truncation of the square root of a. The integer itself can be represented exactly as a double and its square root is computed to the nearest place. If a is one below a square, the rounding may be up, whereas if it is one above a square, the rounding will be down. Thus the square root may be one too large in some instances.

We also have to be careful when the square of this too large value causes an overflow. The same assumptions hold for a single precision float provided the square root itself can be represented in a single float, i.e. for $a < 281474976710656 = 2^{46}$. The remainder is computed by subtracting the square of the computed square root from a.

```
int n_is_square(mp_limb_t x)
```

Returns 1 if x is a square, otherwise 0.

This code first checks if x is a square modulo 64, $63 = 3 \times 3 \times 7$ and $65 = 5 \times 13$, using lookup tables, and if so it then takes a square root and checks that the square of this equals the original value.

```
int n_is_perfect_power235(mp_limb_t n)
```

Returns 1 if n is a perfect square, cube or fifth power.

This function uses a series of modular tests to reject most non 235-powers. Each modular test returns a value from 0 to 7 whose bits respectively indicate whether the value is a square, cube or fifth power modulo the given modulus. When these are logically `AND`ed together, this gives a powerful test which will reject most non-235 powers.

If a bit remains set indicating it may be a square, a standard square root test is performed. Similarly a cube root or fifth root can be taken, if indicated, to determine whether the power of that root is exactly equal to n.

56.13 Factorisation

```
int n_remove(mp_limb_t * n, mp_limb_t p)
```

Removes the highest possible power of p from n, replacing n with the quotient. The return value is that highest power of p that divided n. Assumes n is not 0.

For $p = 2$ trailing zeroes are counted. For other primes p is repeatedly squared and stored in a table of powers with the current highest power of p removed at each step until no higher power can be removed. The algorithm then proceeds down the power tree again removing powers of p until none remain.

```
int n_remove2_precomp(mp_limb_t * n, mp_limb_t p, double
   ppre)
```

Removes the highest possible power of p from n, replacing n with the quotient. The return value is that highest power of p that divided n. Assumes n is not 0. We require `ppre` to be set to a precomputed inverse of p computed with `n_precompute_inverse()`.

For $p = 2$ trailing zeroes are counted. For other primes p we make repeated use of `n_divrem2_precomp()` until division by p is no longer possible.

```
void n_factor_insert(n_factor_t * factors, mp_limb_t p,
   ulong exp)
```

Inserts the given prime power factor `p^exp` into the `n_factor_t factors`. See the documentation for `n_factor_trial()` for a description of the `n_factor_t` type.

The algorithm performs a simple search to see if p already exists as a prime factor in the structure. If so the exponent there is increased by the supplied exponent. Otherwise a new factor `p^exp` is added to the end of the structure.

There is no test code for this function other than its use by the various factoring functions, which have test code.

```
mp_limb_t n_factor_trial_range(n_factor_t * factors,
   mp_limb_t n, ulong start, ulong num_primes)
```

Trial factor n with the first `num_primes` primes, but starting at the prime with index start (counting from zero).

One requires an initialised `n_factor_t` structure, but factors will be added by default to an already used `n_factor_t`. Use the function `n_factor_init()` defined in `ulong_extras` if initialisation has not already been completed on factors.

Once completed, `num` will contain the number of distinct prime factors found. The field p is an array of `mp_limb_t`'s containing the distinct prime factors, `exp` an array containing the corresponding exponents.

The return value is the unfactored cofactor after trial factoring is done.

The function calls `n_compute_primes()` automatically. See the documentation for that function regarding limits.

The algorithm stops when the current prime has a square exceeding n, as no prime factor of n can exceed this unless n is prime.

The precomputed inverses of all the primes computed by `n_compute_primes()` are utilised with the `n_remove2_precomp()` function.

```
mp_limb_t n_factor_trial(n_factor_t * factors, mp_limb_t n,
   ulong num_primes)
```

This function calls `n_factor_trial_range()`, with the value of 0 for `start`. By default this adds factors to an already existing `n_factor_t` or to a newly initialised one.

```
mp_limb_t n_factor_power235(ulong *exp, mp_limb_t n)
```

Returns 0 if n is not a perfect square, cube or fifth power. Otherwise it returns the root and sets `exp` to either 2, 3 or 5 appropriately.

This function uses a series of modular tests to reject most non 235-powers. Each modular test returns a value from 0 to 7 whose bits respectively indicate whether the value is a square, cube or fifth power modulo the given modulus. When these are logically `AND`ed together, this gives a powerful test which will reject most non-235 powers.

If a bit remains set indicating it may be a square, a standard square root test is performed. Similarly a cube root or fifth root can be taken, if indicated, to determine whether the power of that root is exactly equal to n.

```
mp_limb_t n_factor_one_line(mp_limb_t n, ulong iters)
```

This implements Bill Hart's one line factoring algorithm [16]. It is a variant of Fermat's algorithm which cycles through a large number of multipliers instead of incrementing the square root. It is faster than SQUFOF for n less than about 2^{40}.

```
mp_limb_t n_factor_lehman(mp_limb_t n)
```

Lehman's factoring algorithm. Currently works up to 10^{16}, but is not particularly efficient and so is not used in the general factor function. Always returns a factor of n.

```
mp_limb_t n_factor_SQUFOF(mp_limb_t n, ulong iters)
```

Attempts to split n using the given number of iterations of SQUFOF. Simply set `iters` to `~WORD(0)` for maximum persistence.

The version of SQUFOF imlemented here is as described by Gower and Wagstaff [12].

We start by trying SQUFOF directly on n. If that fails we multiply it by each of the primes in `flint_primes_small` in turn. As this multiplication may result in a two limb

value we allow this in our implementation of SQUFOF. As SQUFOF works with values about half the size of n it only needs single limb arithmetic internally.

If SQUFOF fails to factor n we return 0, however with `iters` large enough this should never happen.

```
void n_factor(n_factor_t * factors, mp_limb_t n, int proved)
```

Factors n with no restrictions on n. If the prime factors are required to be checked with a primality test, one may set `proved` to 1, otherwise set it to 0, and they will only be probable primes. N.B: at the present there is no difference because the probable prime tests have been exhaustively tested up to 2^64.

However, in future, this flag may produce and separately check a primality certificate. This may be quite slow (and probably no less reliable in practice).

For details on the `n_factor_t` structure, see `n_factor_trial()`.

This function first tries trial factoring with a number of primes specified by the constant `FLINT_FACTOR_TRIAL_PRIMES`. If the cofactor is 1 or prime the function returns with all the factors.

Otherwise, the cofactor is placed in the array `factor_arr`. Whilst there are factors remaining in there which have not been split, the algorithm continues. At each step each factor is first checked to determine if it is a perfect power. If so it is replaced by the power that has been found. Next if the factor is small enough and composite, in particular, less than `FLINT_FACTOR_ONE_LINE_MAX` then `n_factor_one_line()` is called with `FLINT_FACTOR_ONE_LINE_ITERS` to try and split the factor. If that fails or the factor is too large for `n_factor_one_line()` then `n_factor_SQUFOF()` is called, with `FLINT_FACTOR_SQUFOF_ITERS`. If that fails an error results and the program aborts. However this should not happen in practice.

```
mp_limb_t n_factor_trial_partial(n_factor_t * factors,
   mp_limb_t n, mp_limb_t * prod, ulong num_primes,
   mp_limb_t limit)
```

Attempts trial factoring of n with the first `num_primes primes`, but stops when the product of prime factors so far exceeds `limit`.

One requires an initialised `n_factor_t` structure, but factors will be added by default to an already used `n_factor_t`. Use the function `n_factor_init()` defined in `ulong_extras` if initialisation has not already been completed on `factors`.

Once completed, `num` will contain the number of distinct prime factors found. The field p is an array of `mp_limb_t`'s containing the distinct prime factors, `exp` an array containing the corresponding exponents.

The return value is the unfactored cofactor after trial factoring is done. The value `prod` will be set to the product of the factors found.

The function calls `n_compute_primes()` automatically. See the documentation for that function regarding limits.

The algorithm stops when the current prime has a square exceeding n, as no prime factor of n can exceed this unless n is prime.

The precomputed inverses of all the primes computed by `n_compute_primes()` are utilised with the `n_remove2_precomp()` function.

```
mp_limb_t n_factor_partial(n_factor_t * factors, mp_limb_t
   n, mp_limb_t limit, int proved)
```

Factors n, but stops when the product of prime factors so far exceeds `limit`.

One requires an initialised `n_factor_t` structure, but factors will be added by default to an already used `n_factor_t`. Use the function `n_factor_init()` defined in `ulong_extras` if initialisation has not already been completed on `factors`.

On exit, `num` will contain the number of distinct prime factors found. The field p is an array of `mp_limb_t`'s containing the distinct prime factors, `exp` an array containing the corresponding exponents.

The return value is the unfactored cofactor after factoring is done.

The factors are proved prime if `proved` is 1, otherwise they are merely probably prime.

```
mp_limb_t n_factor_pp1(mp_limb_t n, ulong B1, ulong c)
```

Factors n using Williams' $p + 1$ factoring algorithm, with prime limit set to $B1$. We require c to be set to a random value. Each trial of the algorithm with a different value of c gives another chance to factor n, with roughly exponentially decreasing chance of finding a missing factor. If $p + 1$ (or $p - 1$) is not smooth for any factor p of n, the algorithm will never succeed. The value c should be less than n and greater than 2.

If the algorithm succeeds, it returns the factor, otherwise it returns 0 or 1 (the trivial factors modulo n).

56.14 Arithmetic functions

```
int n_moebius_mu(mp_limb_t n)
```

Computes the Moebius function $\mu(n)$, which is defined as $\mu(n) = 0$ if n has a prime factor of multiplicity greater than 1, $\mu(n) = -1$ if n has an odd number of distinct prime factors, and $\mu(n) = 1$ if n has an even number of distinct prime factors. By convention, $\mu(0) = 0$.

For even numbers, we use the identities $\mu(4n) = 0$ and $\mu(2n) = -\mu(n)$. Odd numbers up to a cutoff are then looked up from a precomputed table storing $\mu(n) + 1$ in groups of two bits.

For larger n, we first check if n is divisible by a small odd square and otherwise call `n_factor()` and count the factors.

```
void n_moebius_mu_vec(int * mu, ulong len)
```

Computes $\mu(n)$ for `n = 0, 1, ..., len - 1`. This is done by sieving over each prime in the range, flipping the sign of $\mu(n)$ for every multiple of a prime p and setting $\mu(n) = 0$ for every multiple of p^2.

```
int n_is_squarefree(mp_limb_t n)
```

Returns 0 if n is divisible by some perfect square, and 1 otherwise. This simply amounts to testing whether $\mu(n) \neq 0$. As special cases, 1 is considered squarefree and 0 is not considered squarefree.

```
mp_limb_t n_euler_phi(mp_limb_t n)
```

Computes the Euler totient function $\phi(n)$, counting the number of positive integers less than or equal to n that are coprime to n.

56.15 Factorials

```
mp_limb_t n_factorial_fast_mod2_preinv(ulong n, mp_limb_t
   p, mp_limb_t pinv)
```

Returns $n!$ mod p given a precomputed inverse of p as computed by `n_preinvert_limb()`. p is not required to be a prime, but no special optimisations are made for composite p. Uses fast multipoint evaluation, running in about $O(n^{1/2})$ time.

```
mp_limb_t n_factorial_mod2_preinv(ulong n, mp_limb_t p,
   mp_limb_t pinv)
```

Returns $n!$ mod p given a precomputed inverse of p as computed by `n_preinvert_limb()`. p is not required to be a prime, but no special optimisations are made for composite p.

Uses a lookup table for small n, otherwise computes the product if n is not too large, and calls the fast algorithm for extremely large n.

56.16 Primitive Roots and Discrete Logarithms

```
mp_limb_t n_primitive_root_prime_prefactor(mp_limb_t p,
   n_factor_t * factors)
```

Returns a primitive root for the multiplicative subgroup of Z/pZ where p is prime given the factorisation ($factors$) of $p - 1$.

```
mp_limb_t n_primitive_root_prime(mp_limb_t p)
```

Returns a primitive root for the multiplicative subgroup of Z/pZ where p is prime.

```
mp_limb_t n_discrete_log_bsgs(mp_limb_t b, mp_limb_t a,
   mp_limb_t n)
```

Returns the discrete logarithm of b with respect to a in the multiplicative subgroup of Z/nZ when Z/nZ is cyclic That is, it returns an number x such that $a^x = b \bmod n$. The multiplicative subgroup is only cyclic when n is 2, 4, p^k, or $2p^k$ where p is an odd prime and k is a positive integer.

§57. long_extras

Signed single limb arithmetic

57.1 Properties

`size_t z_sizeinbase(slong n, int b)`

Returns the number of digits in the base b representation of the absolute value of the integer n.

Assumes that $b \geq 2$.

57.2 Random functions

`mp_limb_signed_t z_randtest(flint_rand_t state)`

Returns a pseudo random number with a random number of bits, from 0 to `FLINT_BITS`. The probability of the special values 0, ± 1, `COEFF_MAX`, `COEFF_MIN`, `WORD_MAX` and `WORD_MIN` is increased.

This random function is mainly used for testing purposes.

`mp_limb_signed_t z_randtest_not_zero(flint_rand_t state)`

As for `z_randtest(state)`, but does not return 0.

`mp_limb_t z_randint(flint_rand_t state, mp_limb_t limit)`

Returns a pseudo random number of absolute value less than `limit`. If `limit` is zero or exceeds `WORD_MAX`, it is interpreted as `WORD_MAX`.

§58. fft

Fast Fourier Transforms

58.1 Split/combine FFT coefficients

```
mp_size_t fft_split_limbs(mp_limb_t ** poly, mp_srcptr
   limbs, mp_size_t total_limbs, mp_size_t coeff_limbs,
   mp_size_t output_limbs)
```

Split an integer `(limbs, total_limbs)` into coefficients of length `coeff_limbs` limbs and store as the coefficients of `poly` which are assumed to have space for `output_limbs + 1` limbs per coefficient. The coefficients of the polynomial do not need to be zeroed before calling this function, however the number of coefficients written is returned by the function and any coefficients beyond this point are not touched.

```
mp_size_t fft_split_bits(mp_limb_t ** poly, mp_srcptr
   limbs, mp_size_t total_limbs, mp_bitcnt_t bits,
   mp_size_t output_limbs)
```

Split an integer `(limbs, total_limbs)` into coefficients of the given number of `bits` and store as the coefficients of `poly` which are assumed to have space for `output_limbs + 1` limbs per coefficient. The coefficients of the polynomial do not need to be zeroed before calling this function, however the number of coefficients written is returned by the function and any coefficients beyond this point are not touched.

```
void fft_combine_limbs(mp_limb_t * res, mp_limb_t ** poly,
   slong length, mp_size_t coeff_limbs, mp_size_t
   output_limbs, mp_size_t total_limbs)
```

Evaluate the polynomial `poly` of the given `length` at `B^coeff_limbs`, where `B = 2^FLINT_BITS`, and add the result to the integer `(res, total_limbs)` throwing away any bits that exceed the given number of limbs. The polynomial coefficients are assumed to have at least `output_limbs` limbs each, however any additional limbs are ignored.

If the integer is initially zero the result will just be the evaluation of the polynomial.

```
void fft_combine_bits(mp_limb_t * res, mp_limb_t ** poly,
   slong length, mp_bitcnt_t bits, mp_size_t output_limbs,
   mp_size_t total_limbs)
```

Evaluate the polynomial `poly` of the given `length` at `2^bits` and add the result to the integer `(res, total_limbs)` throwing away any bits that exceed the given number of

limbs. The polynomial coefficients are assumed to have at least `output_limbs` limbs each, however any additional limbs are ignored. If the integer is initially zero the result will just be the evaluation of the polynomial.

58.2 Test helper functions

```
void fermat_to_mpz(mpz_t m, mp_limb_t * i, mp_size_t limbs)
```

Convert the Fermat number `(i, limbs)` modulo `B^limbs + 1` to an `mpz_t m`. Assumes `m` has been initialised. This function is used only in test code.

58.3 Arithmetic modulo a generalised Fermat number

```
void mpn_addmod_2expp1_1(mp_limb_t * r, mp_size_t limbs,
   mp_limb_signed_t c)
```

Adds the signed limb `c` to the generalised fermat number `r` modulo `B^limbs + 1`. The compiler should be able to inline this for the case that there is no overflow from the first limb.

```
void mpn_normmod_2expp1(mp_limb_t * t, mp_size_t limbs)
```

Given `t` a signed integer of `limbs + 1` limbs in twos complement format, reduce `t` to the corresponding value modulo the generalised Fermat number `B^limbs + 1`, where `B = 2^FLINT_BITS`.

```
void mpn_mul_2expmod_2expp1(mp_limb_t * t, mp_limb_t * i1,
   mp_size_t limbs, mp_bitcnt_t d)
```

Given `i1` a signed integer of `limbs + 1` limbs in twos complement format reduced modulo `B^limbs + 1` up to some overflow, compute `t = i1*2^d` modulo p. The result will not necessarily be fully reduced. The number of bits `d` must be nonnegative and less than `FLINT_BITS`. Aliasing is permitted.

```
void mpn_div_2expmod_2expp1(mp_limb_t * t, mp_limb_t * i1,
   mp_size_t limbs, mp_bitcnt_t d)
```

Given `i1` a signed integer of `limbs + 1` limbs in twos complement format reduced modulo `B^limbs + 1` up to some overflow, compute `t = i1/2^d` modulo p. The result will not necessarily be fully reduced. The number of bits `d` must be nonnegative and less than `FLINT_BITS`. Aliasing is permitted.

58.4 Generic butterflies

```
void fft_adjust(mp_limb_t * r, mp_limb_t * i1, mp_size_t i,
   mp_size_t limbs, mp_bitcnt_t w)
```

Set `r` to `i1` times z^i modulo `B^limbs + 1` where z corresponds to multiplication by 2^w. This can be thought of as part of a butterfly operation. We require $0 \leq i < n$ where $nw =$ `limbs*FLINT_BITS`. Aliasing is not supported.

```
void fft_adjust_sqrt2(mp_limb_t * r, mp_limb_t * i1,
   mp_size_t i, mp_size_t limbs, mp_bitcnt_t w, mp_limb_t *
   temp)
```

Set `r` to `i1` times z^i modulo `B^limbs + 1` where z corresponds to multiplication by $\sqrt{2}^w$. This can be thought of as part of a butterfly operation. We require $0 \leq i < 2 * n$ and odd where $nw =$ `limbs*FLINT_BITS`.

```
void butterfly_lshB(mp_limb_t * t, mp_limb_t * u, mp_limb_t
   * i1, mp_limb_t * i2, mp_size_t limbs, mp_size_t x,
   mp_size_t y)
```

We are given two integers `i1` and `i2` modulo `B^limbs + 1` which are not necessarily normalised. We compute `t = (i1 + i2)*B^x` and `u = (i1 - i2)*B^y` modulo p. Aliasing between inputs and outputs is not permitted. We require `x` and `y` to be less than `limbs` and nonnegative.

```
void butterfly_rshB(mp_limb_t * t, mp_limb_t * u, mp_limb_t
   * i1, mp_limb_t * i2, mp_size_t limbs, mp_size_t x,
   mp_size_t y)
```

We are given two integers `i1` and `i2` modulo `B^limbs + 1` which are not necessarily normalised. We compute `t = (i1 + i2)/B^x` and `u = (i1 - i2)/B^y` modulo p. Aliasing between inputs and outputs is not permitted. We require `x` and `y` to be less than `limbs` and nonnegative.

58.5 Radix 2 transforms

```
void fft_butterfly(mp_limb_t * s, mp_limb_t * t, mp_limb_t
   * i1, mp_limb_t * i2, mp_size_t i, mp_size_t limbs,
   mp_bitcnt_t w)
```

Set `s = i1 + i2`, `t = z1^i*(i1 - i2)` modulo `B^limbs + 1` where `z1 = exp(Pi*I/n)` corresponds to multiplication by 2^w. Requires $0 \leq i < n$ where $nw =$ `limbs*FLINT_BITS`.

```
void ifft_butterfly(mp_limb_t * s, mp_limb_t * t, mp_limb_t
   * i1, mp_limb_t * i2, mp_size_t i, mp_size_t limbs,
   mp_bitcnt_t w)
```

Set `s = i1 + z1^i*i2`, `t = i1 - z1^i*i2` modulo `B^limbs + 1` where `z1 = exp(-Pi*I/n)` corresponds to division by 2^w. Requires $0 \leq i < 2n$ where $nw =$ `limbs*FLINT_BITS`.

```
void fft_radix2(mp_limb_t ** ii, mp_size_t n, mp_bitcnt_t
   w, mp_limb_t ** t1, mp_limb_t ** t2)
```

The radix 2 DIF FFT works as follows:

Input: `[i0, i1, ..., i(m-1)]`, for $m = 2n$ a power of 2.

Output: `[r0, r1, ..., r(m-1)]`
`= FFT[i0, i1, ..., i(m-1)]`.

Algorithm:

• Recursively compute `[r0, r2, r4, ...., r(m-2)]`
`= FFT[i0+i(m/2), i1+i(m/2+1), ..., i(m/2-1)+i(m-1)]`

• Let `[t0, t1, ..., t(m/2-1)]`
`= [i0-i(m/2), i1-i(m/2+1), ..., i(m/2-1)-i(m-1)]`

• Let `[u0, u1, ..., u(m/2-1)]`
`= [z1^0*t0, z1^1*t1, ..., z1^(m/2-1)*t(m/2-1)]` where `z1 = exp(2*Pi*I/m)` corresponds to multiplication by 2^w.

• Recursively compute `[r1, r3, ..., r(m-1)]`
`= FFT[u0, u1, ..., u(m/2-1)]`

The parameters are as follows:

• `2*n` is the length of the input and output arrays

• w is such that 2^w is an $2n$-th root of unity in the ring $\mathbb{Z}/p\mathbb{Z}$ that we are working in, i.e. $p = 2^{wn} + 1$ (here n is divisible by `GMP_LIMB_BITS`)

• `ii` is the array of inputs (each input is an array of limbs of length `wn/GMP_LIMB_BITS + 1` (the extra limbs being a "carry limb"). Outputs are written in-place.

We require nw to be at least 64 and the two temporary space pointers to point to blocks of size `n*w + FLINT_BITS` bits.

```
void fft_truncate(mp_limb_t ** ii,  mp_size_t n,
   mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t ** t2,
   mp_size_t trunc)
```

As for `fft_radix2` except that only the first `trunc` coefficients of the output are computed and the input is regarded as having (implied) zero coefficients from coefficient `trunc` onwards. The coefficients must exist as the algorithm needs to use this extra space, but their value is irrelevant. The value of `trunc` must be divisible by 2.

```
void fft_truncate1(mp_limb_t ** ii, mp_size_t n,
   mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t ** t2,
   mp_size_t trunc)
```

As for `fft_radix2` except that only the first `trunc` coefficients of the output are computed. The transform still needs all $2n$ input coefficients to be specified.

```
void ifft_radix2(mp_limb_t ** ii, mp_size_t n, mp_bitcnt_t
   w, mp_limb_t ** t1, mp_limb_t ** t2)
```

The radix 2 DIF IFFT works as follows:

Input: `[i0, i1, ..., i(m-1)]`, for $m = 2n$ a power of 2.

Output: `[r0, r1, ..., r(m-1)]`
`= IFFT[i0, i1, ..., i(m-1)]`.

Algorithm:

• Recursively compute `[s0, s1, ...., s(m/2-1)]`
`= IFFT[i0, i2, ..., i(m-2)]`

• Recursively compute `[t(m/2), t(m/2+1), ..., t(m-1)]`
`= IFFT[i1, i3, ..., i(m-1)]`

• Let `[r0, r1, ..., r(m/2-1)]`
`= [s0+z1^0*t0, s1+z1^1*t1, ..., s(m/2-1)+z1^(m/2-1)*t(m/2-1)]` where `z1 = exp(-2*Pi*I/m)` corresponds to division by 2^w.

• Let `[r(m/2), r(m/2+1), ..., r(m-1)]`
`= [s0-z1^0*t0, s1-z1^1*t1, ..., s(m/2-1)-z1^(m/2-1)*t(m/2-1)]`

The parameters are as follows:

• `2*n` is the length of the input and output arrays

• w is such that 2^w is an $2n$-th root of unity in the ring $\mathbb{Z}/p\mathbb{Z}$ that we are working in, i.e. $p = 2^{wn} + 1$ (here n is divisible by `GMP_LIMB_BITS`)

• `ii` is the array of inputs (each input is an array of limbs of length `wn/GMP_LIMB_BITS + 1` (the extra limbs being a "carry limb"). Outputs are written in-place.

We require nw to be at least 64 and the two temporary space pointers to point to blocks of size `n*w + FLINT_BITS` bits.

```
void ifft_truncate(mp_limb_t ** ii, mp_size_t n,
   mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t ** t2,
   mp_size_t trunc)
```

As for `ifft_radix2` except that the output is assumed to have zeros from coefficient trunc onwards and only the first trunc coefficients of the input are specified. The remaining coefficients need to exist as the extra space is needed, but their value is irrelevant. The value of `trunc` must be divisible by 2.

Although the implementation does not require it, we assume for simplicity that `trunc` is greater than n. The algorithm begins by computing the inverse transform of the first n coefficients of the input array. The unspecified coefficients of the second half of the array are then written: coefficient `trunc + i` is computed as a twist of coefficient `i` by a root of unity. The values of these coefficients are then equal to what they would have been if the inverse transform of the right hand side of the input array had been computed with full data from the start. The function `ifft_truncate1` is then called on the entire right half of the input array with this auxilliary data filled in. Finally a single layer of the IFFT is completed on all the coefficients up to `trunc` being careful to note that this involves doubling the coefficients from `trunc - n` up to `n`.

```
void ifft_truncate1(mp_limb_t ** ii, mp_size_t n,
   mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t ** t2,
   mp_size_t trunc)
```

Computes the first `trunc` coefficients of the radix 2 inverse transform assuming the first `trunc` coefficients are given and that the remaining coefficients have been set to the value they would have if an inverse transform had already been applied with full data.

The algorithm is the same as for `ifft_truncate` except that the coefficients from `trunc` onwards after the inverse transform are not inferred to be zero but the supplied values.

```
void fft_butterfly_sqrt2(mp_limb_t * s, mp_limb_t * t,
   mp_limb_t * i1, mp_limb_t * i2, mp_size_t i, mp_size_t
   limbs, mp_bitcnt_t w, mp_limb_t * temp)
```

Let $w = 2k+1$, $i = 2j+1$. Set `s = i1 + i2`, `t = z1^i*(i1 - i2)` modulo `B^limbs + 1` where `z1^2 = exp(Pi*I/n)` corresponds to multiplication by 2^w. Requires $0 \leq i < 2n$ where $nw =$ `limbs*FLINT_BITS`.

Here `z1` corresponds to multiplication by 2^k then multiplication by `(2^(3nw/4)- 2^(nw/4))`. We see `z1^i` corresponds to multiplication by `(2^(3nw/4)- 2^(nw/4))*2^(j+ik)`.

We first multiply by `2^(j + ik + wn/4)` then multiply by an additional `2^(nw/2)` and subtract.

```
void ifft_butterfly_sqrt2(mp_limb_t * s, mp_limb_t * t,
   mp_limb_t * i1, mp_limb_t * i2, mp_size_t i, mp_size_t
   limbs, mp_bitcnt_t w, mp_limb_t * temp)
```

Let $w = 2k+1$, $i = 2j+1$. Set `s = i1 + z1^i*i2`, `t = i1 - z1^i*i2` modulo `B^limbs + 1` where `z1^2 = exp(-Pi*I/n)` corresponds to division by 2^w. Requires $0 \leq i < 2n$ where $nw =$ `limbs*FLINT_BITS`.

Here `z1` corresponds to division by 2^k then division by `(2^(3nw/4)- 2^(nw/4))`. We see `z1^i` corresponds to division by `(2^(3nw/4)- 2^(nw/4))*2^(j+ik)` which is the same as division by `2^(j+ik + 1)` then multiplication by `(2^(3nw/4)- 2^(nw/4))`.

Of course, division by `2^(j+ik + 1)` is the same as multiplication by `2^(2*wn - j - ik - 1)`. The exponent is positive as $i \leq 2 * n$, $j < n$, $k < w/2$.

We first multiply by `2^(2*wn - j - ik - 1 + wn/4)` then multiply by an additional `2^(nw/2)` and subtract.

```
void fft_truncate_sqrt2(mp_limb_t ** ii, mp_size_t n,
   mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t ** t2,
   mp_limb_t ** temp, mp_size_t trunc)
```

As per `fft_truncate` except that the transform is twice the usual length, i.e. length $4n$ rather than $2n$. This is achieved by making use of twiddles by powers of a square root of 2, not powers of 2 in the first layer of the transform.

We require nw to be at least 64 and the three temporary space pointers to point to blocks of size `n*w + FLINT_BITS` bits.

```
void ifft_truncate_sqrt2(mp_limb_t ** ii, mp_size_t n,
   mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t ** t2,
   mp_limb_t ** temp, mp_size_t trunc)
```

As per `ifft_truncate` except that the transform is twice the usual length, i.e. length $4n$ instead of $2n$. This is achieved by making use of twiddles by powers of a square root of 2, not powers of 2 in the final layer of the transform.

We require nw to be at least 64 and the three temporary space pointers to point to blocks of size `n*w + FLINT_BITS` bits.

58.6 Matrix Fourier Transforms

```
void fft_butterfly_twiddle(mp_limb_t * u, mp_limb_t * v,
   mp_limb_t * s, mp_limb_t * t, mp_size_t limbs,
   mp_bitcnt_t b1, mp_bitcnt_t b2)
```

Set `u = 2^b1*(s + t)`, `v = 2^b2*(s - t)` modulo `B^limbs + 1`. This is used to compute `u = 2^(ws*tw1)*(s + t)`,
`v = 2^(w+ws*tw2)*(s - t)` in the matrix fourier algorithm, i.e. effectively computing an ordinary butterfly with additional twiddles by `z1^rc` for row r and column c of the matrix of coefficients. Aliasing is not allowed.

```
void ifft_butterfly_twiddle(mp_limb_t * u, mp_limb_t * v,
   mp_limb_t * s, mp_limb_t * t, mp_size_t limbs,
   mp_bitcnt_t b1, mp_bitcnt_t b2)
```

Set `u = s/2^b1 + t/2^b1)`, `v = s/2^b1 - t/2^b1` modulo `B^limbs + 1`. This is used to compute `u = 2^(-ws*tw1)*s + 2^(-ws*tw2)*t)`,
`v = 2^(-ws*tw1)*s + 2^(-ws*tw2)*t)` in the matrix fourier algorithm, i.e. effectively computing an ordinary butterfly with additional twiddles by `z1^(-rc)` for row r and column c of the matrix of coefficients. Aliasing is not allowed.

```
void fft_radix2_twiddle(mp_limb_t ** ii, mp_size_t is,
   mp_size_t n, mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t
   ** t2, mp_size_t ws, mp_size_t r, mp_size_t c, mp_size_t
   rs)
```

As for `fft_radix2` except that the coefficients are spaced by `is` in the array `ii` and an additional twist by `z^c*i` is applied to each coefficient where i starts at r and increases by `rs` as one moves from one coefficient to the next. Here `z` corresponds to multiplication by `2^ws`.

```
void ifft_radix2_twiddle(mp_limb_t ** ii, mp_size_t is,
   mp_size_t n, mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t
   ** t2, mp_size_t ws, mp_size_t r, mp_size_t c, mp_size_t
   rs)
```

As for `ifft_radix2` except that the coefficients are spaced by `is` in the array `ii` and an additional twist by `z^(-c*i)` is applied to each coefficient where i starts at r and increases by `rs` as one moves from one coefficient to the next. Here `z` corresponds to multiplication by `2^ws`.

```
void fft_truncate1_twiddle(mp_limb_t ** ii, mp_size_t is,
   mp_size_t n, mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t
   ** t2, mp_size_t ws, mp_size_t r, mp_size_t c, mp_size_t
   rs, mp_size_t trunc)
```

As per `fft_radix2_twiddle` except that the transform is truncated as per `fft_truncate1`.

```
void ifft_truncate1_twiddle(mp_limb_t ** ii, mp_size_t is,
   mp_size_t n, mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t
   ** t2, mp_size_t ws, mp_size_t r, mp_size_t c, mp_size_t
   rs, mp_size_t trunc)
```

As per `ifft_radix2_twiddle` except that the transform is truncated as per `ifft_truncate1`.

```
void fft_mfa_truncate_sqrt2(mp_limb_t ** ii, mp_size_t n,
   mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t ** t2,
   mp_limb_t ** temp, mp_size_t n1, mp_size_t trunc)
```

This is as per the `fft_truncate_sqrt2` function except that the matrix fourier algorithm is used for the left and right FFTs. The total transform length is $4n$ where `n = 2^depth` so that the left and right transforms are both length $2n$. We require `trunc > 2*n` and that `trunc` is divisible by `2*n1` (explained below).

The matrix fourier algorithm, which is applied to each transform of length $2n$, works as follows. We set `n1` to a power of 2 about the square root of n. The data is then thought of as a set of `n2` rows each with `n1` columns (so that `n1*n2 = 2n`).

The length $2n$ transform is then computed using a whole pile of short transforms. These comprise `n1` column transforms of length `n2` followed by some twiddles by roots of unity (namely `z^rc` where r is the row and c the column within the data) followed by `n2` row transforms of length `n1`. Along the way the data needs to be rearranged due to the fact that the short transforms output the data in binary reversed order compared with what is needed.

The matrix fourier algorithm provides better cache locality by decomposing the long length $2n$ transforms into many transforms of about the square root of the original length.

For better cache locality the sqrt2 layer of the full length $4n$ transform is folded in with the column FFTs performed as part of the first matrix fourier algorithm on the left half of the data.

The second half of the data requires a truncated version of the matrix fourier algorithm. This is achieved by truncating to an exact multiple of the row length so that the row transforms are full length. Moreover, the column transforms will then be truncated transforms and their truncated length needs to be a multiple of 2. This explains the condition on `trunc` given above.

To improve performance, the extra twiddles by roots of unity are combined with the butterflies performed at the last layer of the column transforms.

We require nw to be at least 64 and the three temporary space pointers to point to blocks of size `n*w + FLINT_BITS` bits.

```
void ifft_mfa_truncate_sqrt2(mp_limb_t ** ii, mp_size_t n,
   mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t ** t2,
   mp_limb_t ** temp, mp_size_t n1, mp_size_t trunc)
```

This is as per the `ifft_truncate_sqrt2` function except that the matrix fourier algorithm is used for the left and right IFFTs. The total transform length is $4n$ where `n = 2^depth` so that the left and right transforms are both length $2n$. We require `trunc > 2*n` and that `trunc` is divisible by `2*n1`.

We set `n1` to a power of 2 about the square root of n.

As per the matrix fourier FFT the sqrt2 layer is folded into the the final column IFFTs for better cache locality and the extra twiddles that occur in the matrix fourier algorithm are combined with the butterflied performed at the first layer of the final column transforms.

We require nw to be at least 64 and the three temporary space pointers to point to blocks of size `n*w + FLINT_BITS` bits.

```
void fft_mfa_truncate_sqrt2_outer(mp_limb_t ** ii,
   mp_size_t n, mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t
   ** t2, mp_limb_t ** temp, mp_size_t n1, mp_size_t trunc)
```

Just the outer layers of `fft_mfa_truncate_sqrt2`.

```
void fft_mfa_truncate_sqrt2_inner(mp_limb_t ** ii,
   mp_limb_t ** jj, mp_size_t n, mp_bitcnt_t w, mp_limb_t
   ** t1, mp_limb_t ** t2, mp_limb_t ** temp, mp_size_t n1,
   mp_size_t trunc, mp_limb_t * tt)
```

The inner layers of `fft_mfa_truncate_sqrt2` and `ifft_mfa_truncate_sqrt2` combined with pointwise mults.

```
void ifft_mfa_truncate_sqrt2_outer(mp_limb_t ** ii,
   mp_size_t n, mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t
   ** t2, mp_limb_t ** temp, mp_size_t n1, mp_size_t trunc)
```

The outer layers of `ifft_mfa_truncate_sqrt2` combined with normalisation.

58.7 Negacyclic multiplication

```
void fft_negacyclic(mp_limb_t ** ii, mp_size_t n,
   mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t ** t2,
   mp_limb_t ** temp)
```

As per `fft_radix2` except that it performs a sqrt2 negacyclic transform of length $2n$. This is the same as the radix 2 transform except that the i-th coefficient of the input is first multiplied by $\sqrt{2}^{iw}$.

We require nw to be at least 64 and the two temporary space pointers to point to blocks of size `n*w + FLINT_BITS` bits.

```
void ifft_negacyclic(mp_limb_t ** ii, mp_size_t n,
   mp_bitcnt_t w, mp_limb_t ** t1, mp_limb_t ** t2,
   mp_limb_t ** temp)
```

As per `ifft_radix2` except that it performs a sqrt2 negacyclic inverse transform of length $2n$. This is the same as the radix 2 inverse transform except that the i-th coefficient of the output is finally divided by $\sqrt{2}^{iw}$.

We require nw to be at least 64 and the two temporary space pointers to point to blocks of size `n*w + FLINT_BITS` bits.

```
void fft_naive_convolution_1(mp_limb_t * r, mp_limb_t * ii,
   mp_limb_t * jj, mp_size_t m)
```

Performs a naive negacyclic convolution of `ii` with `jj`, both of length m and sets r to the result. This is essentially multiplication of polynomials modulo $x^m + 1$.

```
void _fft_mulmod_2expp1(mp_limb_t * r1, mp_limb_t * i1,
   mp_limb_t * i2, mp_size_t r_limbs, mp_bitcnt_t depth,
   mp_bitcnt_t w)
```

Multiply `i1` by `i2` modulo `B^r_limbs + 1` where `r_limbs = nw/FLINT_BITS` with `n = 2^depth`. Uses the negacyclic FFT convolution CRT'd with a 1 limb naive convolution. We require that `depth` and `w` have been selected as per the wrapper `fft_mulmod_2expp1` below.

```
slong fft_adjust_limbs(mp_size_t limbs)
```

Given a number of limbs, returns a new number of limbs (no more than the next power of 2) which will work with the Nussbaumer code. It is only necessary to make this adjustment if `limbs > FFT_MULMOD_2EXPP1_CUTOFF`.

```
void fft_mulmod_2expp1(mp_limb_t * r, mp_limb_t * i1,
   mp_limb_t * i2, mp_size_t n, mp_size_t w, mp_limb_t * tt)
```

As per `_fft_mulmod_2expp1` but with a tuned cutoff below which more classical methods are used for the convolution. The temporary space is required to fit `n*w + FLINT_BITS` bits. There are no restrictions on n, but if `limbs = n*w/FLINT_BITS` then if `limbs` exceeds `FFT_MULMOD_2EXPP1_CUTOFF` the function `fft_adjust_limbs` must be called to increase the number of limbs to an appropriate value.

58.8 Integer multiplication

```
void mul_truncate_sqrt2(mp_ptr r1, mp_srcptr i1, mp_size_t
   n1, mp_srcptr i2, mp_size_t n2, mp_bitcnt_t depth,
   mp_bitcnt_t w)
```

Integer multiplication using the radix 2 truncated sqrt2 transforms.

Set `(r1, n1 + n2)` to the product of `(i1, n1)` by `(i2, n2)`. This is achieved through an FFT convolution of length at most `2^(depth + 2)` with coefficients of size nw bits where `n = 2^depth`. We require `depth >= 6`. The input data is broken into chunks of data not exceeding `(nw - (depth + 1))/2` bits. If breaking the first integer into chunks of this size results in `j1` coefficients and breaking the second integer results in `j2` chunks then `j1 + j2 - 1 <= 2^(depth + 2)`.

If `n = 2^depth` then we require nw to be at least 64.

```
void mul_mfa_truncate_sqrt2(mp_ptr r1, mp_srcptr i1,
   mp_size_t n1, mp_srcptr i2, mp_size_t n2, mp_bitcnt_t
   depth, mp_bitcnt_t w)
```

As for `mul_truncate_sqrt2` except that the cache friendly matrix fourier algorithm is used.

If `n = 2^depth` then we require nw to be at least 64. Here we also require w to be 2^i for some $i \geq 0$.

```
void flint_mpn_mul_fft_main(mp_ptr r1, mp_srcptr i1,
   mp_size_t n1, mp_srcptr i2, mp_size_t n2)
```

The main integer multiplication routine. Sets `(r1, n1 + n2)` to `(i1, n1)` times `(i2, n2)`. We require `n1 >= n2 > 0`.

58.9 Convolution

```
void fft_convolution(mp_limb_t ** ii, mp_limb_t ** jj,
   slong depth, slong limbs, slong trunc, mp_limb_t ** t1,
   mp_limb_t ** t2, mp_limb_t ** s1, mp_limb_t * tt)
```

Perform an FFT convolution of `ii` with `jj`, both of length `4*n` where `n = 2^depth`. Assume that all but the first `trunc` coefficients of the output (placed in `ii`) are zero. Each coefficient is taken modulo `B^limbs + 1`. The temporary spaces `t1`, `t2` and `s1` must have `limbs + 1` limbs of space and `tt` must have `2*(limbs + 1)` of free space.

§59. qsieve

59.1 Quadratic sieve

`mp_limb_t qsieve_ll_factor(mp_limb_t hi, mp_limb_t lo)`

Given an integer `n = (hi, lo)` find a factor and return it. If a tiny factor is encountered, this is returned very quickly. Otherwise the quadratic sieve algorithm is employed. The algorithm requires that n not be prime and not be a perfect power. There is also a limit to the size of n. During the algorithm n will be multiplied by a small multiplier k (from 1 to 47). The product kn must fit in two limbs. If not the algorithm will silently fail, returning 0. Otherwise a factor of n which fits in a single limb will be returned.

§60. perm

Permutations

60.1 Memory management

```
slong * _perm_init(slong n)
```

Initialises the permutation for use.

```
void _perm_clear(slong *vec)
```

Clears the permutation.

60.2 Assignment

```
void _perm_set(slong *res, const slong *vec, slong n)
```

Sets the permutation `res` to the same as the permutation `vec`.

```
void _perm_set_one(slong *vec, slong n)
```

Sets the permutation to the identity permutation.

```
void _perm_inv(slong *res, const slong *vec, slong n)
```

Sets `res` to the inverse permutation of `vec`. Allows aliasing of `res` and `vec`.

60.3 Composition

```
void _perm_compose(slong *res, const slong *vec1, const
   slong *vec2, slong n)
```

Forms the composition $\pi_1 \circ \pi_2$ of two permutations π_1 and π_2. Here, π_2 is applied first, that is, $(\pi_1 \circ \pi_2)(i) = \pi_1(\pi_2(i))$.

Allows aliasing of `res`, `vec1` and `vec2`.

60.4 Parity

```
int _perm_parity(const slong *vec, slong n)
```

Returns the parity of `vec`, 0 if the permutation is even and 1 if the permutation is odd.

60.5 Randomisation

```
int _perm_randtest(slong *vec, slong n, flint_rand_t state)
```

Generates a random permutation vector of length n and returns its parity, 0 or 1.

This function uses the Knuth shuffle algorithm to generate a uniformly random permutation without retries.

60.6 Input and output

```
int _perm_print(const slong * vec, slong n)
```

Prints the permutation vector of length n to `stdout`.

§61. longlong.h

64-bit arithmetic

61.1 Auxiliary asm macros

`umul_ppmm(high_prod, low_prod, multipler, multiplicand)`

Multiplies two single limb integers `MULTIPLER` and `MULTIPLICAND`, and generates a two limb product in `HIGH_PROD` and `LOW_PROD`.

`smul_ppmm(high_prod, low_prod, multipler, multiplicand)`

As for `umul_ppmm()` but the numbers are signed.

```
udiv_qrnnd(quotient, remainder, high_numerator,
   low_numerator, denominator)
```

Divides an unsigned integer, composed by the limb integers `HIGH_NUMERATOR` and `LOW_NUMERATOR`, by `DENOMINATOR` and places the quotient in `QUOTIENT` and the remainder in `REMAINDER`. `HIGH_NUMERATOR` must be less than `DENOMINATOR` for correct operation.

```
sdiv_qrnnd(quotient, remainder, high_numerator,
   low_numerator, denominator)
```

As for `udiv_qrnnd()` but the numbers are signed. The quotient is rounded towards 0. Note that as the quotient is signed it must lie in the range $[-2^63, 2^63)$.

`count_leading_zeros(count, x)`

Counts the number of zero-bits from the msb to the first non-zero bit in the limb `x`. This is the number of steps `x` needs to be shifted left to set the msb. If `x` is 0 then count is undefined.

`count_trailing_zeros(count, x)`

As for `count_leading_zeros()`, but counts from the least significant end. If `x` is zero then count is undefined.

```
add_ssaaaa(high_sum, low_sum, high_addend_1, low_addend_1,
   high_addend_2, low_addend_2)
```

Adds two limb integers, composed by `HIGH_ADDEND_1` and `LOW_ADDEND_1`, and `HIGH_ADDEND_2` and `LOW_ADDEND_2`, respectively. The result is placed in `HIGH_SUM` and `LOW_SUM`. Overflow, i.e. carry out, is not stored anywhere, and is lost.

```
add_sssaaaaaa(high_sum, mid_sum, low_sum, high_addend_1,
   mid_addend_1, low_addend_1, high_addend_2, mid_addend_2,
   low_addend_2)
```

Adds two three limb integers. Carry out is lost.

```
sub_ddmmss(high_difference, low_difference, high_minuend,
   low_minuend, high_subtrahend, low_subtrahend)
```

Subtracts two limb integers, composed by `HIGH_MINUEND_1` and `LOW_MINUEND_1`, and `HIGH_SUBTRAHEND_2` and `LOW_SUBTRAHEND_2`, respectively. The result is placed in `HIGH_DIFFERENCE` and `LOW_DIFFERENCE`. Overflow, i.e. carry out is not stored anywhere, and is lost.

```
invert_limb(invxl, xl)
```

Computes an approximate inverse `invxl` of the limb `xl`, with an implicit leading 1. More formally it computes

```
invxl = (B^2 - B*x - 1)/x = (B^2 - 1)/x - B
```

Note that x must be normalised, i.e. with msb set. This inverse makes use of the following theorem of Torbjorn Granlund and Peter Montgomery [14, Lemma 8.1]:

Let d be normalised, $d < B$, i.e. it fits in a word, and suppose that $md < B^2 \leq (m+1)d$. Let $0 \leq n \leq Bd - 1$. Write $n = n_2B + n_1B/2 + n_0$ with $n_1 = 0$ or 1 and $n_0 < B/2$. Suppose $q_1B + q_0 = n_2B + (n_2 + n_1)(m - B) + n_1(d - B/2) + n_0$ and $0 \leq q_0 < B$. Then $0 \leq q_1 < B$ and $0 \leq n - q_1d < 2d$.

In the theorem, m is the inverse of d. If we let `m = invxl + B` and $d = x$ we have $md = B^2 - 1 < B^2$ and $(m + 1)x = B^2 + d - 1 \geq B^2$.

The theorem is often applied as follows: note that n_0 and $n_1(d - B/2)$ are both less than $B/2$. Also note that $n_1(m - B) < B$. Thus the sum of all these terms contributes at most 1 to q_1. We are left with $n_2B + n_2(m - B)$. But note that $(m - B)$ is precisely our precomputed inverse `invxl`. If we write $q_1B + q_0 = n_2B + n_2(m - B)$, then from the theorem, we have $0 \leq n - q_1d < 3d$, i.e. the quotient is out by at most 2 and is always either correct or too small.

```
udiv_qrnnd_preinv(q, r, nh, nl, d, di)
```

As for `udiv_qrnnd()` but takes a precomputed inverse `di` as computed by `invert_limb()`. The algorithm, in terms of the theorem above, is:

```
nadj = n1*(d-B/2) + n0
xh, xl = (n2+n1)*(m-B)
xh, xl += nadj + n2*B ( xh, xl = n2*B + (n2+n1)*(m-B) +
   n1*(d-B/2) + n0 )
_q1 = B - xh - 1
xh, xl = _q1*d + nh, nl - B*d = nh, nl - q1*d - d so
   that xh = 0 or -1
r = xl + xh*d where xh is 0 if q1 is off by 1,
   otherwise -1
q = xh - _q1 = xh + 1 + n2
```

§62. mpn_extras

62.1 Macros

`MACRO MPN_NORM(a, an)`

Normalise `(a, an)` so that either `an` is zero or `a[an - 1]` is nonzero.

`MACRO MPN_SWAP(a, an, b, bn)`

Swap `(a, an)` and `(b, bn)`, i.e. swap pointers and sizes.

62.2 Utility functions

`void flint_mpn_debug(mp_srcptr x, mp_size_t xsize)`

Prints debug information about `(x, xsize)` to `stdout`. In particular, this will print binary representations of all the limbs.

`int flint_mpn_zero_p(mp_srcptr x, mp_size_t xsize)`

Returns 1 if all limbs of `(x, xsize)` are zero, otherwise 0.

62.3 Divisibility

`int flint_mpn_divisible_1_p(x, xsize, d) (macro)`

Expression determining whether `(x, xsize)` is divisible by the `mp_limb_t d` which is assumed to be odd-valued and at least 3.

This function is implemented as a macro.

```
mp_size_t flint_mpn_divexact_1(mp_ptr x, mp_size_t xsize,
   mp_limb_t d)
```

Divides x once by a known single-limb divisor, returns the new size.

```
mp_size_t flint_mpn_remove_2exp(mp_ptr x, mp_size_t xsize,
   mp_bitcnt_t *bits)
```

Divides `(x, xsize)` by 2^n where n is the number of trailing zero bits in x. The new size of x is returned, and n is stored in the bits argument. x may not be zero.

```
mp_size_t flint_mpn_remove_power_ascending(mp_ptr x,
   mp_size_t xsize, mp_ptr p, mp_size_t psize, ulong *exp)
```

Divides `(x, xsize)` by the largest power n of `(p, psize)` that is an exact divisor of x. The new size of x is returned, and n is stored in the `exp` argument. x may not be zero, and p must be greater than 2.

This function works by testing divisibility by ascending squares $p, p^2, p^4, p^8, \ldots$, making it efficient for removing potentially large powers. Because of its high overhead, it should not be used as the first stage of trial division.

```
int flint_mpn_factor_trial(mp_srcptr x, mp_size_t xsize,
    slong start, slong stop)
```

Searches for a factor of `(x, xsize)` among the primes in positions `start, ..., stop-1` of `flint_primes`. Returns i if `flint_primes[i]` is a factor, otherwise returns 0 if no factor is found. It is assumed that `start >= 1`.

62.4 Division

```
int flint_mpn_divides(mp_ptr q, mp_srcptr array1, mp_size_t
    limbs1, mp_srcptr arrayg, mp_size_t limbsg, mp_ptr temp)
```

If `(arrayg, limbsg)` divides `(array1, limbs1)` then `(q, limbs1 - limbsg + 1)` is set to the quotient and 1 is returned, otherwise 0 is returned. The temporary space `temp` must have space for `limbsg` limbs.

Assumes limbs1 `limbs1 >= limbsg > 0`.

```
mp_limb_t flint_mpn_preinv1(mp_limb_t d, mp_limb_t d2)
```

Computes a precomputed inverse from the leading two limbs of the divisor `b, n` to be used with the `preinv1` functions. We require the most significant bit of `b, n` to be 1.

```
mp_limb_t flint_mpn_divrem_preinv1(mp_ptr q, mp_ptr a,
    mp_size_t m, mp_srcptr b, mp_size_t n, mp_limb_t dinv)
```

Divide `a, m` by `b, n`, returning the high limb of the quotient (which will either be 0 or 1), storing the remainder in-place in `a, n` and the rest of the quotient in `q, m - n`. We require the most significant bit of `b, n` to be 1. dinv must be computed from `b[n - 1]`, `b[n - 2]` by `flint_mpn_preinv1`. We also require `m >= n >= 2`.

```
void flint_mpn_mulmod_preinv1(mp_ptr r, mp_srcptr a,
    mp_srcptr b, mp_size_t n, mp_srcptr d, mp_limb_t dinv,
    ulong norm)
```

Given a normalised integer d with precomputed inverse `dinv` provided by `flint_mpn_preinv1`, computes $ab \pmod d$ and stores the result in r. Each of a, b and r is expected to have n limbs of space, with zero padding if necessary.

The value `norm` is provided for convenience. If a, b and d have been shifted left by `norm` bits so that d is normalised, then r will be shifted right by `norm` bits so that it has the same shift as all the inputs.

We require a and b to be reduced modulo n before calling the function.

```
void flint_mpn_preinvn(mp_ptr dinv, mp_srcptr d, mp_size_t
    n)
```

Compute an n limb precomputed inverse `dinv` of the n limb integer d.

We require that d is normalised, i.e. with the most significant bit of the most significant limb set.

```
void flint_mpn_mod_preinvn(mp_ptr r, mp_srcptr a, mp_size_t
   m, mp_srcptr d, mp_size_t n, mp_srcptr dinv)
```

Given a normalised integer d of n limbs, with precomputed inverse `dinv` provided by `flint_mpn_preinvn` and integer a of m limbs, computes $a \pmod d$ and stores the result in-place in the lower n limbs of a. The remaining limbs of a are destroyed.

We require $m \geq n$. No aliasing of a with any of the other operands is permitted.

Note that this function is not always as fast as ordinary division.

```
mp_limb_t flint_mpn_divrem_preinvn(mp_ptr q, mp_ptr r,
   mp_srcptr a, mp_size_t m, mp_srcptr d, mp_size_t n,
   mp_srcptr dinv)
```

Given a normalised integer d with precomputed inverse `dinv` provided by `flint_mpn_preinvn`, computes the quotient of a by d and stores the result in q and the remainder in the lower n limbs of a. The remaining limbs of a are destroyed.

The value q is expected to have space for $m - n$ limbs and we require $m >= n$. No aliasing is permitted between q and a or between these and any of the other operands.

Note that this function is not always as fast as ordinary division.

```
void flint_mpn_mulmod_preinvn(mp_ptr r, mp_srcptr a,
   mp_srcptr b, mp_size_t n, mp_srcptr d, mp_srcptr dinv,
   ulong norm)
```

Given a normalised integer d with precomputed inverse `dinv` provided by `flint_mpn_preinvn`, computes $ab \pmod d$ and stores the result in r. Each of a, b and r is expected to have n limbs of space, with zero padding if necessary.

The value `norm` is provided for convenience. If a, b and d have been shifted left by `norm` bits so that d is normalised, then r will be shifted right by `norm` bits so that it has the same shift as all the inputs.

We require a and b to be reduced modulo n before calling the function.

Note that this function is not always as fast as ordinary division.

62.5 GCD

```
mp_size_t flint_mpn_gcd_full(mp_ptr arrayg, mp_ptr array1,
   mp_size_t limbs1, mp_ptr array2, mp_size_t limbs2)
```

Sets `(arrayg, retvalue)` to the gcd of `(array1, limbs1)` and `(array2, limbs2)`.

The only assumption is that neither `limbs1` or `limbs2` is zero.

62.6 Special numbers

```
void flint_mpn_harmonic_odd_balanced(mp_ptr t, mp_size_t *
   tsize, mp_ptr v, mp_size_t * vsize, slong a, slong b,
   slong n, int d)
```

Computes `(t,tsize)` and `(v,vsize)` such that $t/v = H_n = 1 + 1/2 + \cdots + 1/n$. The computation is performed using recursive balanced summation over the odd terms. The resulting fraction will not generally be normalized. At the top level, this function should be called with $n > 0$, $a = 1$, $b = n$, and $d = 1$.

Enough space should be allocated for t and v to fit the entire sum $1 + 1/2 + \cdots + 1/n$ computed without normalization; i.e. t and v should have room to fit $n!$ plus one extra

limb.

62.7 Random Number Generation

```
void flint_mpn_rrandom(mp_limb_t *rp, gmp_randstate_t
   state, mp_size_t n)
```

Generates a random number with `n` limbs and stores it on `rp`. The number it generates will tend to have long strings of zeros and ones in the binary representation.

Useful for testing functions and algorithms, since this kind of random numbers have proven to be more likely to trigger corner-case bugs.

```
void flint_mpn_urandomb(mp_limb_t *rp, gmp_randstate_t
   state, mp_bitcnt_t n)
```

Generates a uniform random number `n` bits and stores it on `rp`.

§63. flintxx

C++ wrapper library for FLINT

63.1 Introduction

FLINT provides fast algorithms for number theoretic computations. For many reasons, it is written in C. Nonetheless, some users prefer object oriented syntax. FLINT ships with a set of wrapper C++ classes, together termed flintxx, which provide such an object oriented syntax.

In this chapter, we describe how to *use* flintxx. The FLINT developer wishing to extend the wrapper library should consult appendix A.

In general, flintxx strives to behave just like the underlying FLINT C interface, except that we use classes and C++ operators to make the client code look more pleasant. The simple example from the section on `fmpz` can be transcribed into C++ as follows:

```
#include "fmpzxx.h"
...
using namespace flint;
fmpzxx x, y;
x = 7u;
y = x*x;
std::cout << x << "^2 = " << y << std::endl;
```

As can be seen here, and in general, if a FLINT C interface is called `foo` and resides in `foo.h`, then the C++ version is called `fooxx` and resides in `fooxx.h`. All flintxx classes live inside `namespace flint`.

Functions which operate on wrapper classes are usually implemented as overloaded stand-alone functions, with the type prefix dropped. So for example a call to `flint::gcd(f1, f2)` yields an expression template evaluating via `fmpz_gcd`, provided `f1` and `f2` are (evaluate to) instances of `fmpzxx`. Sometimes we felt that dropping the type prefix would yield incomprehensible names, as for example in `fmpq_next_minimal`, or `fmpq_reconstruct`. In these cases the type prefix is swapped for the flintxx equivalent, so the flintxx version would be called `fmpqxx_reconstruct`. In this situation, usually the same functionality is also exposed as a (possibly static) member function, and this is the preferred way of accessing the functionality. Thus one should write `fmpqxx::reconstruct(a, m)` or `fmpqxx(0, 1u).next_minimal()`.

63.2 Overview

Expression templates The implementation of flintxx tries very hard not to incur any overhead over using the native C interface. For this reason, we use *expression templates* for lazily evaluating expressions. This allows us to avoid creating excessively many temporaries, for example. This means that even if `x` and `y` are of type `fmpzxx` (say), then `x + y` will *not* be of type `fmpzxx`. Instead it will be an object which for most purposes behaves just like `fmpzxx`, but really only expresses the fact that it represents the sum of `x` and `y`.

This distinction almost never matters, since expression templates are evaluated automatically in most cases. Thus `cout << x + y` or `x + y == 7` will work just as one might expect. There are ways to request explicit evaluation of an expression template, most notably `(x + y).evaluate()` and `fmpzxx(x + y)`.

One caveat of the expression template approach is that compiler error messages can be long and hard to understand. In flintxx we strive to type-check template parameters as early as possible in order to keep error messages short and close to the actual user error. Excessively long error messages are often indicative of a bug in flintxx.

Tuples Many FLINT functions naturally return two (or more) arguments. A typical example is `divrem`. The underlying C function is (for example) `void fmpz_poly_divrem(fmpz_poly_t Q, fmpz_poly_t R, const fmpz_poly_t A, const fmpz_poly_t B)`. Mapping this directly to C++ would yield something like `void divrem(fmpz_polyxx& Q, fmpz_polyxx& R, const fmpz_polyxx& A, const fmpz_polyxx& B)`. While functional, this is not particularly nice; the syntax `divrem(Q, R, A, B)`, where the first two arguments are modified, is just very reminiscent of C. We would prefer an expression closer to the python analogue `(Q, R) = divrem(A, B)`.

This is where *ltuples* enter.[1] In fact, the following is a valid flintxx expression: `ltupleref(Q, R)= divrem(A, B)`.

In generality, the implementation of ltuples is fairly involved template metaprogramming. For the purpose of this documentation, ltuple types are denoted as follows: `Ltuple<Type1, Type2, ..., Typer>`. So `divrem` would return an object of type `Ltuple<fmpz_polyxx, fmpz_polyxx>`. The user should never try to construct such types by hand; instead use the function `ltupleref` (and perhaps occasionally `ltuple`; both documented later).

One thing to observe is that ltuples are typed fairly weakly. Thus assignments and equality comparisons can be performed as long as both sides have the same length, and the operation can be performed on all components (whether or not the component types match).

Another interesting feature of ltuples is the type `flint::detail::IGNORED_TYPE`. In an ltuple assignment, where the left hand side contains a reference to this type, the relevant entry is just discarded. Since the `ltuple.h` header automatically creates a static instance `_` of this type, in the following listing, the lines marked `(1)` and `(2)` have the same effect (but the second is potentially more efficient).

```
#include "fmpz_polyxx.h"
...
using namespace flint;

fmpz_polyxx f, g;
// ...
```

[1] The 'l' in `ltuple` stands for "lazy", i.e. the fact that they can be used in expression templates. The reason for this name is that flintxx internally uses a non-lazy tuple class just called `tuple`, on which `ltuple` in fact is built.

```
fmpz_polyxx R;
ltupleref(_, R) = divrem(f, g); // (1)
R = f % g;                      // (2)
```

Note finally that using `ltuple` intermediates often results in more copies than necessary. For example the expression `ltupleref(num, _)= divrem(a, b)` assigns the quotient to `num`, creating just a temporary `fmpzxx` to hold the remainder. In contrast, `num = divrem(a, b).get<0>()` creates two temporary instances of `fmpzxx`.

Reference types One subtlety in wrapping a C library is that references do not work as easily as one might expect. For example, consider the class `fmpqxx`, wrapping `fmpq_t`, i.e. rational numbers. As such, an instance of `fmpqxx` has a numerator and denominator. In C, these are accessible via macros `fmpq_numref` and `fmpq_denref`, which yield `fmpz*`, which can be used essentially interchangeably with `fmpz_t`. In particular, any library function which operates on `fmpz_t` can operate on the numerator or denominator of an `fmpq_t`. In C++, we would like to have a member function `den` (and `num`) which returns an object of type `fmpzxx&` (i.e. a reference to `fmpzxx`).

However, this is not possible, since `fmpqxx` is *not* implemented as a pair of `fmpzxx`, and instead simply contains an `fmpq_t`.

For this reason, for every C interface `foo`, flintxx provides two additional types, called `fooxx_ref` and `fooxx_srcref`, acting as replacements for `fooxx&` and `const foox&`, respectively, in situations where no underlying C++ object exists. Instances of `fooxx_ref` or `fooxx_srcref` behave exactly like instances `fooxx`, in fact the user should never notice a difference. Any flintxx operation or expression which works on objects of type `foo` also works on objects of type `fooxx_ref` and `fooxx_srcref`. Moreover, instances of `foo` can be converted implicitly to `fooxx_ref` or `fooxx_srcref`, and `fooxx_ref` can be converted implicitly to `fooxx_srcref`. It is also possible to explicitly convert reference types `fooxx_*ref` to `fooxx` (since this entails copying, we provide no implicit conversion).

In summary, the class `fooxx_ref` behaves like a reference to an object of type `fooxx`. As such it can be used both as a right hand side and as a left hand side, just like `fooxx`. The class `fooxx_srcref` behaves like a reference to a constant object of type `fooxx`, and so cannot be used as a left hand side. These objects are created by flintxx automatically, for example upon calling `fmpqxx::num()`.

Unified coefficient access Consider again the `x.num()` method of `fmpqxx`. In various situations, this can have different return types. Namely, if `x` is a writable expression, then `x.num()` returns an `fmpzxx_ref`. In particular the return value behaves just like `fmpzxx`, no evaluation is necessary to obtain it, there are no copies, and it is possible to change the return value (and thus change `x`). If on the other hand `x` is a readonly immediate, then the return value of `x.num()` has type `fmpzxx_srcref`. This again behaves just like `fmpzxx` and no evaluations or copies are necessary, but this time it is not possible to change the return value (and so it is not possible to change `x`, either). Finally, if `x` is a lazy expression, then the return value is actually a lazy expression template. Thus to obtain the "actual" value of `x.num()`, evaluations are necessary, and potentially so are copies.

Thus in any case the return value behaves just like `fmpqxx`, but apart from that the behaviour of `x.num()` varies quite drastically in the different situations. We call this "unified coefficient access" (the coefficients of a `fmpqxx` being `num(), den()`), and the same behaviour occurs in many other flintxx types, e.g. in `fmpz_polyxx.coeff()`, etc.

Type conversion As a rule, flintxx does not perform automatic type conversions (except when related to the promotion to reference types, c/f earlier discussion). In

expression templates, operands can be automatically promoted if the underlying C interface provides this facility. Beyond that, types have to be converted explicitly.

There are two ways of doing this. The preferred one is using static constructor functions. Typical examples are `fmpz_polyxx::from_ground(fmpzarg)` and `nmod_polyxx::reduce(mplimbarg, nmodctxarg)`. The former takes an (expression template evaluating to) `fmpzxx` and returns an `fmpz_polyxx` representing the constant polynomial with value the `fmpzxx`. The latter takes an argument of type `mp_limb_t` and one of type `nmodxx_ctx_srcref` (essentially a word-sized modulus) and returns an `nmod_polyxx` representing the constant polynomial obtained by reducing `mplimbarg`.

The general format for this is `totype::constructorname(arg1, arg2, ...)`. We prefer this because it makes explicit the type that is being converted to, and the way the arguments are to be interpreted.

This format only works if the target type is part of flintxx. In other cases, we use a `.to<totype>()` syntax, as in `fmpzexpr.to<slong>()`.

Input and output In C++ it is customary to provide input and output via iostreams, and overloading the operators `<<` and `>>`. When wrapping a C library which works on the `FILE` interface, this is rather hard to accomplish. For this reason, flintxx only provides streaming output (i.e. `<<`), and only when there is a `to_string` method. Unfortunately this applies to only a small subset of the FLINT types.

For output in other cases, and input in all cases, we provide C-like functions. Namely, the functions `print`, `print_pretty`, `read` and `read_pretty` can be used similarly to the C `flint_printf` and `scanf`. For example, `print(x)` where `x` is an `fmpz` has the same effect as `std::cout << x`.

Extending flintxx Explanations on the inner workings of the flintxx expression template library and how they pertain to wrapping new C interfaces can be found in appendix A. Here we just want to remark that the flintxx classes are not designed for inheritance. If you want to modify behaviour, you should wrap flintxx types into your own classes (extension by aggregation, not inheritance). The header `flintxx/forwarding.h` was meant to facilitate this, but has not been finished.

63.3 Notations and conventions for the C++ interface documentation

As explained above, the flintxx classes and functions perform quite a number of operations which should be invisible to the user. Some template types implement methods which only make sense for some template arguments, etc. For example, every expression template built from `fmpq_polyxx` (polynomials with rational coefficients) has a method `set_coeff`. However, this method only makes sense for objects of type `fmpq_polyxx` or `fmpq_polyxx_ref` (calling it on other types will result in a compilation error), and its existence in objects of other types should be considered an implementation detail.

In what follows, we document a "virtual" set of classes and functions, which explain how the user should expect its objects to behave, and which we guarantee to maintain. Other interfaces should be considered implementation details and subject to change.

Consider the interface `fmpzxx`, and more concretely an instance `a`. As in the above discussion, we see that from `a` we can build a lot of different objects: expression templates like `a+a`, constant objects like `const fmpzxx& b = a;`, reference objects like `fmpzxx_ref c(a)`, etc. These by nature behave somewhat differently. For our purposes, we classify types into "targets" (things which can be assigned to), "sources" (things which contain actual computed data, or references thereto, as opposed to lazy expression templates)

and "expressions" (sources or expression templates). Note that every target is a source, and every source is an expression.

We denote any type which can act as a target for `fmpzxx` as `Fmpz_target` (note the initial capital letter!), any `fmpzxx` source as `Fmpz_source` and any `fmpzxx` expression as `Fmpz_expr`. Such "made up" type names (always with initial capital letter) are referred to as "virtual types" in the sequel; these are used for all flint classes (e.g. `Fmpq_expr` or `Fmpz_polyxx_src`). Some examples of virtual types for `fmpzxx` are given in table 63.1.

Fmpz_target	Fmpz_src	Fmpz_expr
`fmpzxx a;`	`const fmpzxx a;`	`a + b`
`fmpzxx& b = a;`	`const fmpzxx& b(a);`	`a * b`
`fmpzxx_ref c(a);`	`fmpzxx_srcref c(a);`	`gcd(a, b)`

Table 63.1: Examples of virtual types for `fmpzxx`.

When using virtual types, we will suppress reference notation. No flintxx types are ever copied automatically, unless the documentation explicitly says so. This is a general philosophy of flintxx: the library does as many things automatically as it can, without introducing additional calls to underlying flint C functions. So for example, it is not possible to implicitly convert `int` to `fmpzxx` (since doing so requires a C call). Of course explicit conversions (or assignments) work completely fine.

It is also often the case that flintxx functions are conditionally enabled templates. A notation such as `void foo(T:is_signed_integer)` denotes a template function which is enabled whenever the template parameter `T` satisfies the *type trait* `is_signed_integer`. These type traits should be self-explanatory, and in any case are documented in TODO.

In what follows, we will never document copy constructors, or implicit conversion constructors pertaining to reference types. We will also not document assignment operators for expressions of the same type. (Thus if `x` is an `fmpzxx` and `y` is an `fmpqxx`, then `x = x` and `y = x` are both valid, but only the second assignment operator is documented explicitly.)

Most flintxx functions and methods wrap underlying C functions in a way which is evident from the signature of the flintxx function/method. If this is the case, no further documentation is provided. For example, the function `double dlog(Fmpz_expr x)` simply wraps `double fmpz_dlog(const fmpz_t)`. As is evident from the return type, `dlog` immediately evaluates its argument, and then computes the logarithm. In contrast, a function like `Fmpz_expr gcd(Fmpz_expr, Fmpz_expr)` returns a lazily evaluated expression template (and wraps `void fmpz_gcd(fmpz_t, const fmpz_t, const fmpz_t)`).

In case the C function has more than one return value in the form of arguments passed in by reference, the C++ wrapper returns an `ltuple`. In this case, the order of the `ltuple` arguments is the same as the order of the function arguments; so for example `ltupleref(Q, R)= divrem(A, B)` has the same effect as `fmpz_poly_divrem(q, r, a, b)`, provided `Q, R, A, B` are `fmpz_polyxx` and `q, r, a, b` are the underlying `fmpz_poly_t`. If such a convention is followed, the documentation below may not further explain anything. In all other cases, further explanation is provided (this applies in particular if the C function has return type different from `void`).

Global functions or member functions? Often it is not clear if functionality should be exposed as a global function, such as `gcd(a, b)`, or as a member function, such as `a.gcd(b)`. In flintxx, we strive to make both available when feasible. In the following documentation, the global versions are documented in detail (explaining the allowed types etc), whereas the member function versions are summarised more briefly under e.g. `Fmpz_expr::unary operation()const`, `Fmpz_expr::binary operation(??)const` etc.

63.4 flint_exception

This is the main exception type used by the flintxx library. It derives from `std::domain_error`. As such its main method is `what()`, yielding an English language description of the problem encountered.

63.5 frandxx

The type `frandxx` wraps `flint_rand_t` and takes care of initialising and clearing random states. It is defined in `flintxx/frandxx.h`. Note that this type is not copyable.

```
frandxx::frandxx()
```

Initialize random state.

```
flint_rand_t& frandxx::_data()

const flint_rand_t& frandxx::_data() const
```

Obtain a reference to the underlying C random state.

63.6 ltuple

Lazy tuples are implemented in `flintxx/ltuple.h`. They are used throughout flintxx to emulate functions with several return values.

This header automatically creates a static instance of `flint::detail::IGNORED_TYPE`. It is accessible in namespace flint, under the name `FLINT_LTUPLE_PLACEHOLDER_NAME`, which defaults to `_`. See `ltupleref` documentation for how to use this.

```
Ltuple<T1&, ..., Tn&> ltupleref(T1& t1, ..., Tn& tn)
```

Construct an ltuple of references, binding to the arguments `t1`, ..., `tn`. Instances of `flint::detail::IGNORED_TYPE` can be used as placeholders. Currently $n \leq 4$.

```
Ltuple<T1, ..., Tn> ltuple(const T1& t1, ..., const Tn& tn)
```

Construct an ltuple containing copies of `t1`, ..., `tn`. Currently `n \le 4`.

```
Tk_expr Ltuple<T1, ..., Tn>_expr::get<k>() const
```

If `Tk` is an expression template type, then the `get<k>()` method returns a lazy expression evaluating to the kth element of the (potentially lazy) ltuple.

If `Tk` is not an expression template type, this method evaluates the ltuple, and returns the kth entry.

On ltuple immediates, reference versions are also available, which can be used to manipulate the entries.

63.7 permxx

Permutations are mostly used by row reduction algorithms. Even though we support limited arithmetic on them (e.g. composition), permutations are not implemented as expression templates.

`permxx` wraps the C interface `perm` operating on `slong*`.

```
permxx::permxx(slong n)
```

```
static permxx permxx::one(slong n)
```

Initialize an identity permutation on the set $[n] = \{0, 1, \ldots, n-1\}$.

```
static permxx permxx::randtest(slong n)
```

Generate a random permutation on $[n]$. See `_perm_randtest`.

```
bool permxx::operator==(const permxx&)
```

```
bool permxx::operator!=(const permxx&)
```

```
slong permxx::size() const
```

Return the size of the set being permuted (n in the constructors).

```
slong& operator[](slong i)
```

```
slong operator[](slong i) const
```

Return the image of i under the permutation.

```
permxx permxx::operator*(const permxx&)
```

```
permxx compose(const permxx& p1, const permxx& p2)
```

Compute the composition of two permutations. See `_perm_compose`.

```
void permxx::set_inv(const permxx& o)
```

Set self to the inverse permutation of `o`.

```
permxx permxx::inv() const
```

```
permxx inv(const permxx&)
```

Return the inverse permutation.

```
int print(const permxx&)
```

63.8 fmpzxx

63.8.1 C++ particulars

```
Fmpz_expr::unary operation() const
```

The following unary functions are made available as member functions: `sqrt`, `abs`.

```
Fmpz_expr::binary operation(??) const
```

The following binary functions are made available as member functions: `cdiv_q`, `divexact`, `fdiv_qr`, `fdiv_r`, `fdiv_r_2exp`, `gcd`, `gcdinv`, `invmod`, `lcm`, `negmod`, `pow`, `rfac`, `root`, `sqrtmod`, `tdiv_q`, `tdiv_q_2exp`, `tdiv_qr`, `xgcd`.

```
Fmpz_expr::ternary operation(??, ??) const
```

The following ternary functions are made available as member functions: `divexact2`, `mul2`, `mul_tdiv_q_2exp`, `powm`.

63.8.2 Memory management

```
fmpzxx::fmpzxx()
```

Initialize to zero.

```
fmpzxx::fmpzxx(const char*)
```

```
fmpzxx::fmpzxx(T:is_integer)
```

Initialize from a primitive data type. See `fmpz_set_str`, `fmpz_set_si` and `fmpz_set_ui`.

63.8.3 Random generation

```
static fmpzxx fmpzxx::randbits(frandxx& state)
```

```
static fmpzxx fmpzxx::randtest(frandxx& state)
```

```
static fmpzxx fmpzxx::randtest_unsigned(frandxx& state)
```

```
static fmpzxx fmpzxx::randtest_not_zero(frandxx& state)
```

```
static fmpzxx fmpzxx::randm(frandxx& state, Fmpz_expr m)
```

```
static fmpzxx fmpzxx::randtest_mod(frandxx& state,
   Fmpz_expr m)
```

```
static fmpzxx fmpzxx::randtest_mod_signed(frandxx& state,
   Fmpz_expr m)
```

63.8.4 Conversion

```
std::string Fmpz_expr::to_string(int base = 10) const
```

Convert self into a `string`. See `fmpz_get_str`.

```
slong Fmpz_expr::to<slong>() const
```

Convert self to `slong`. See `fmpz_get_si`.

```
slong Fmpz_expr::to<ulong>() const
```

Convert self to `ulong`. See `fmpz_get_si`.

```
double Fmpz_expr::to<double>() const
```

Convert self to `double`. See `fmpz_get_d`.

```
double Fmpz_expr::get_d_2exp(long& exp) const
```

```
Fmpz_target Fmpz_target::operator=(const char*)
```

```
Fmpz_target Fmpz_target::operator=(T:is_integer)
```

See `fmpz_set_str`, `fmpz_set_ui` and `fmpz_set_si`.

```
void Fmpz_target::set_ui_smod(mp_limb_t x, mv_limb_t m)
```

```
void Fmpz_target::set_uiui(mp_limb_t x, mv_limb_t m)

void Fmpz_target::neg_uiui(mp_limb_t x, mv_limb_t m)
```

63.8.5 Input and output

```
int print(Fmpz_expr)

int print(FILE*, Fmpz_expr)

int read(Fmpz_target)

int read(FILE*, Fmpz_target)
```

63.8.6 Basic properties and manipulation

```
size_t Fmpz_expr::sizeinbase(int) const

size_t sizeinbase(Fmpz_expr, int)

mp_bitcnt_t Fmpz_expr::bits() const

mp_bitcnt_t bits(Fmpz_expr)

mp_bitcnt_t Fmpz_expr::size() const

mp_bitcnt_t size(Fmpz_expr)

mp_bitcnt_t Fmpz_expr::val2() const

mp_bitcnt_t val2(Fmpz_expr)

int Fmpz_expr::sign() const

int sign(Fmpz_expr)

void Fmpz_target::set_zero()

void Fmpz_target::set_one()

bool Fmpz_expr::abs_fits_ui() const

bool Fmpz_expr::fits_si() const

void Fmpz_target::setbit(ulong)

bool Fmpz_expr::tstbit(ulong) const
```

63.8.7 Comparision

Relational operators `<=`, `>` etc are overloaded, where `e1` and `e2` can be any combination of `Fmpz_expr` and `T:is_integer`. See `fmpz_cmp`, `fmpz_cmp_si` and `fmpz_cmp_ui`.

```
bool Fmpz_expr::is_zero() const
```

Return if this expression evaluates to zero.

```
bool Fmpz_expr::is_one() const
```

Return if this expression evaluates to one.

```
bool Fmpz_expr::is_pm1() const
```

Return if this expression evaluates to ± 1.

```
bool Fmpz_expr::is_even() const
```

Return if this expression evaluates to an even integer.

```
bool Fmpz_expr::is_odd() const
```

Return if the expression evaluates to an odd integer.

63.8.8 Basic arithmetic

Arithmetic operators `+`, `-`, `*`, `/`, `%`, `<<` and `>>` are overloaded. See the `fmpz` documentation for which argument types are allowed. Symmetric operators with asymmetric type arguments can be used in either order, even if this is not exposed in the C interface.

The shift operators wrap `fmpz_fdiv_q_2exp` and `fmpz_mul_2exp`. The division operators use `fmpz_fdiv`.

```
Fmpz_expr abs(Fmpz_expr)

Fmpz_expr mul2_uiui(Fmpz_expr g, ulong x, ulong y)

Fmpz_expr cdiv_q(Fmpz_expr, Fmpz_expr)

Fmpz_expr cdiv_q(Fmpz_expr, T:is_integer)

Fmpz_expr tdiv_q(Fmpz_expr, Fmpz_expr)

Fmpz_expr tdiv_q(Fmpz_expr, T:is_integer)

Fmpz_expr divexact(Fmpz_expr, Fmpz_expr)

Fmpz_expr divexact(Fmpz_expr, T:is_integer)

Fmpz_expr fdiv_r(Fmpz_expr, Fmpz_expr)

Fmpz_expr tdiv_q_2exp(Fmpz_expr, T:is_unsigned_integer)

Fmpz_expr fdiv_r_2exp(Fmpz_expr, T:is_unsigned_integer)

Fmpz_expr divexact2(Fmpz_expr g, ulong x, ulong y)

Fmpz_expr mul_tdiv_q_2exp(Fmpz_expr g, Fmpz_expr x, ulong
   exp)

Fmpz_expr mul_tdiv_q_2exp(Fmpz_expr g, long x, ulong exp)

Ltuple<fmpzxx, fmpzxx>_expr fdiv_qr(Fmpz_expr g, Fmpz_expr
   h)

Ltuple<fmpzxx, fmpzxx>_expr tdiv_qr(Fmpz_expr g, Fmpz_expr
   h)
```

```
bool Fmpz_expr::divisible(Fmpz_expr) const

bool Fmpz_expr::divisible(T:fits_into_slong) const

bool divisible(Fmpz_expr n, Fmpz_expr d)

bool divisible(Fmpz_expr n, T:fits_into_slong d)
```

Return if d divides n. See `fmpz_divisible`.

```
Fmpz_expr powm(Fmpz_expr g, ulong e, Fmpz_expr m)

Fmpz_expr powm(Fmpz_expr g, Fmpz_expr e, Fmpz_expr m)

Fmpz_expr pow(Fmpz_expr, T:is_unsigned_integer)

long clog(Fmpz_expr x, Fmpz_expr b)

long clog(Fmpz_expr x, ulong b)

long flog(Fmpz_expr x, Fmpz_expr b)

long flog(Fmpz_expr x, ulong b)

double dlog(Fmpz_expr x)

long Fmpz_expr::clog(Fmpz_expr) const

long Fmpz_expr::clog(T:is_unsigned_integer) const

long Fmpz_expr::flog(Fmpz_expr) const

long Fmpz_expr::flog(T:is_unsigned_integer) const

double Fmpz_expr::dlog() const

Ltuple<bool, fmpzxx>_expr sqrtmod(Fmpz_expr a, Fmpz_expr b)
```

`ltupleref(b, N)= sqrtmod(A, B)` has the same effect as `b = fmpz_sqrtmod(n, a, b)`, where `n, a, b` are the underlying `fmpz_t` of `N, A, B`.

```
Ltuple<fmpzxx, fmpzxx>_expr sqrtrem(Fmpz_expr g)

Fmpz_expr sqrt(Fmpz_expr)

bool Fmpz_expr::is_square() const
```

Return if this expression evaluates to a square integer.

```
Fmpz_expr root(Fmpz_expr, T:fits_into_slong)

Fmpz_expr rfac(Fmpz_expr, T:is_unsigned_integer)

Fmpz_expr fac(T:is_unsigned_integer)

Fmpz_expr fib(T:is_unsigned_integer)

Fmpz_expr bin(T:is_unsigned_integer, U:is_unsigned_integer)
```

63.8.9 Greatest common divisor

```
Ltuple<fmpzxx, fmpzxx>_expr gcdinv(Fmpz_expr f, Fmpz_expr g)

Ltuple<fmpzxx, fmpzxx, fmpzxx>_expr xgcd(Fmpz_expr f,
   Fmpz_expr g)

Fmpz_expr gcd(Fmpz_expr, Fmpz_expr)

Fmpz_expr lcm(Fmpz_expr, Fmpz_expr)
```

63.8.10 Modular arithmetic

```
Ltuple<slong, fmpzxx>_expr remove(Fmpzxx a, Fmpzxx b)

int jacobi(Fmpz_expr a, Fmpz_expr p)

int Fmpz_expr::jacobi(Fmpz_expr) const

Fmpz_expr invmod(Fmpz_expr, Fmpz_expr)

Fmpz_expr negmod(Fmpz_expr, Fmpz_expr)
```

63.8.11 Bit packing and unpacking

Beware that argument orders are different relative to the C interface, to facilitate default arguments.

```
static Fmpz_expr fmpzxx::bit_unpack(const
   vector<mp_limb_t>& v, mp_bitcnt_t bits, mp_bitcnt_t
   shift = 0, int negate = 0, bool borrow = false)

static Fmpz_expr fmpzxx::bit_unpack_unsigned(const
   vector<mp_limb_t>& v, mp_bitcnt_t bits, mp_bitcnt_t
   shift = 0)
```

Unpack an `fmpzxx` from `v`.

```
bool bit_pack(std::vector<mp_limb_t>& v, mp_bitcnt_t bits,
   Fmpz_expr, mp_bitcnt_t shift = 0, int negate = 0, bool
   borrow = false)
```

Pack an `fmpzxx` to `v`. The vector `v` is required to be of sufficient size.

63.8.12 Logic operations

Binary logic operators `& | ^` (and, or, xor) are also overloaded (implemented when both arguments are `Fmpz_expr`).

```
void Fmpz_target::clrbit(ulong i)

void Fmpz_target::combit(ulong i)

int Fmpz_expr::popcnt() const
```

63.8.13 Chinese remaindering

```
Fmpz_expr Fmpz_expr::CRT(Fmpz_expr, T:is_unsigned_integer,
    T:is_unsigned_integer, bool) const
```

```
Fmpz_expr CRT(Fmpz_expr, Fmpz_expr, T:is_unsigned_integer,
    T:is_unsigned_integer, bool)
```

See `fmpz_CRT_ui`.

```
fmpz_combxx::fmpz_combxx(const std::vector<mp_limb_t>&
    primes)
```

The class `fmpz_combxx` wraps both `fmpz_comb_t` and `fmpz_comb_temp_t`. The argument `primes` is the vector of moduli to use, and must not be deallocated before the newly constructed `fmpz_combxx`. Note that the internal `fmpz_comb_temp_t` structure may be modified even on constant instances of `fmpz_combxx`.

```
void multi_mod(std::vector<mp_limb_t>& out, Fmpz_expr in,
    const fmpz_combxx& comb)
```

Reduce `in` modulo the primes stored in `comb`, and store the results in `out`. The vector `out` must have sufficient size, and its size will not be changed.

```
Fmpz_expr multi_CRT(const std::vector<mp_limb_t>& residues,
    const fmpz_combxx comb, bool sign)
```

Reconstruct an integer from its residues. See `fmpz_multi_CRT_ui`.

63.8.14 Primality testing

```
bool Fmpz_expr::is_probabprime() const
```

```
bool Fmpz_expr::is_prime_pseudosquare() const
```

63.9 fmpz_factorxx

```
fmpz_factorxx::fmpz_factorxx()
```

Initialise an empty factorisation.

```
fmpz_factorxx::fmpz_factorxx(const fmpz_factorxx& o)
```

Copy a factorisation.

```
bool fmpz_factorxx::operator==(const fmpz_factorxx&)
```

Compare two factorisations.

```
ulong fmpz_factorxx::size() const
```

Return the number of stored factors.

```
ulong fmpz_factorxx::exp(slong i) const
```

```
ulong& fmpz_factorxx::exp(slong i)
```

Obtain the exponent of the ith factor.

```
fmpzxx_srcref fmpz_factorxx::p(slong i) const
```

```
fmpzxx_ref fmpz_factorxx::p(slong i)
```

Obtain the ith factor.

```
int fmpz_factorxx::sign() const

int& fmpz_factorxx::sign()
```

Obtain the sign of the factored expression.

```
void fmpz_factorxx::set_factor(Fmpz_expr)

void fmpz_factorxx::set_factor(T:fits_into_slong)

bool fmpz_factorxx::set_factor_trial_range(Fmpz_expr,
   ulong, ulong)

bool fmpz_factorxx::set_factor_pp1(Fmpz_expr, ulong, ulong,
   ulong)
```

Factorise an integer and store its factors. See `fmpz_factor` etc.

```
Fmpz_expr fmpz_factorxx::expand() const

Fmpz_expr fmpz_factorxx::expand_iterative() const

Fmpz_expr fmpz_factorxx::expand_multiexp() const

fmpz_factorxx factor(Fmpz_expr)

fmpz_factorxx factor(T:fits_into_slong)

Ltuple<bool, fmpz_factorxx>_expr
   factor_trial_range(Fmpz_expr)

fmpz_factorxx factor_pp1(Fmpz_expr)

void print(const fmpz_factorxx&)
```

63.10 fmpz_matxx

The class `fmpz_matxx` wraps `fmpz_mat_t`, and so represents matrices with coefficients in $\mathbb{Z}$.

Owing to the design of `fmpz_mat_t`, the use of `fmpz_matxx` has a number of peculiarities.

- Matrix assignment does not automatically resize. This also includes assigning (and thus evaluating) a lazy expression to an ordinary matrix. As a consequence, the evaluation code cannot use temporary merging, and may thus create more temporaries than a similar expression involving non-matrices.
- Several functions operating on `fmpz_mat_t` do not allow aliasing. The flintxx layer just passes expressions on to the C layer, so it is the responsibility of the user to avoid aliasing where it is disallowed. Note that since no temporary merging is used with matrices, aliases are never introduced by the evaluation code.

63.10.1 Not yet split into subsections

```
slong Fmpz_mat_expr::rank() const
```

```
Fmpz_expr Fmpz_mat_expr::det_modular_given_divisor(
   Fmpz_mat_expr, Fmpz_expr) const
```

See `fmpz_mat_det_modular_given_divisor`.

```
Fmpz_mat_target
   Fmpz_mat_target::operator=(T:fits_into_slong)
```

```
Fmpz_mat_target Fmpz_mat_target::operator=(const char*)
```

63.10.2 C++ particulars

```
slong Fmpz_mat_expr::rows() const
```

```
slong Fmpz_mat_expr::cols() const
```

Obtain the number of rows/columns in this matrix. These functions never cause evaluation (the matrix size is computed from the operations in the expression template and the size of the input matrices).

```
Fmpz_mat_expr::unary operation() const
```

The following unary functions are made available as member functions: `sqr`, `charpoly`, `det`, `det_bareiss`, `det_bound`, `det_cofactor`, `det_divisor`, `trace`, `transpose`.

```
Fmpz_mat_expr::binary operation(??) const
```

The following binary functions are made available as member functions: `det_modular`, `det_modular_accelerated`, `divexact`, `mul_classical`, `mul_multi_mod`, `pow`, codesolve, `solve_bound`, `solve_cramer`, `solve_dixon`, `solve_fflu`.

63.10.3 Memory management

```
fmpz_matxx::fmpz_matxx(slong i, slong j)
```

Allocate a matrix of size $i \times j$.

63.10.4 Basic assignment and manipulation

```
?? Fmpz_mat_expr::at(T:fits_into_slong, U:fits_into_slong)
   const
```

Unified coefficient access to the matrix entries.

```
void Fmpq_mat_target::set_zero()
```

```
void Fmpq_mat_target::set_one()
```

```
static fmpq_matxx fmpq_matxx::zero(slong rows, slong cols)
```

```
static fmpq_matxx fmpq_matxx::one(slong rows, slong cols)
```

63.10.5 Input and output

```
print(Fmpz_mat_expr)
```

```
print(FILE*, Fmpz_mat_expr)

print_pretty(Fmpz_mat_expr)

print_pretty(FILE*, Fmpz_mat_expr)

read(Fmpz_mat_target)

read(FILE*, Fmpz_mat_target)
```

63.10.6 Comparison

The overloaded operator == can be used for equality testing. Additionally, we have the following functions.

```
bool Fmpz_mat_expr::is_zero() const

bool Fmpz_mat_expr::is_empty() const

bool Fmpz_mat_expr::is_quare() const
```

63.10.7 Conversion

```
static fmpz_matxx fmpz_matxx::lift(Nmod_mat_expr)

static fmpz_matxx fmpz_matxx::lift_unsigned(Nmod_mat_expr)
```

See `fmpz_mat_set_nmod_mat` and `fmpz_mat_set_nmod_mat_unsigned`.

```
static fmpz_matxx fmpz_matxx::reduce(Fmpq_mat_expr,
   Fmz_expr)
```

See `fmpq_mat_get_fmpz_mat_mod_fmpz`.

```
static fmpz_matxx
   fmpz_matxx::from_integral_fraction(Fmpq_mat_expr)

void Fmpz_mat_target::set_integral_fraction(Fmpq_mat_expr)
```

See `fmpq_mat_get_fmpz_mat`. Raises `flint_exception` if the argument has non-integer entries.

63.10.8 Randomisation

```
void Fmpz_mat_target::set_randbits(frandxx& state,
   mp_bitcnt_t bits)

void Fmpz_mat_target::set_randtest(frandxx& state,
   mp_bitcnt_t bits)

void Fmpz_mat_target::set_randintrel(frandxx& state,
   mp_bitcnt_t bits)

void Fmpz_mat_target::set_randsimdioph(frandxx& state,
   mp_bitcnt_t bits, mp_bitcount_t bits2)
```

```
void Fmpz_mat_target::set_randtrulike(frandxx& state,
    mp_bitcnt_t bits, ulong q)

void Fmpz_mat_target::set_randtrulike2(frandxx& state,
    mp_bitcnt_t bits, ulong q)

void Fmpz_mat_target::set_randajtai(frandxx& state,
    mp_bitcnt_t bits, double alpha)

void Fmpz_mat_target::set_randrank(frandxx& state, slong
    rank, mp_bitcnt_t bits)

void Fmpz_mat_target::set_randdet(frandxx& state, Fmpz_expr
    d)
```

See `fmpz_mat_randbits` etc.

```
static fmpz_matxx fmpz_matxx::randbits(slong r, slong c,
    frandxx& state, mp_bitcnt_t bits)

static fmpz_matxx fmpz_matxx::randtest(slong r, slong c,
    frandxx& state, mp_bitcnt_t bits)

static fmpz_matxx fmpz_matxx::randintrel(slong r, slong c,
    frandxx& state, mp_bitcnt_t bits)

static fmpz_matxx fmpz_matxx::randsimdioph(slong r, slong
    c, frandxx& state, mp_bitcnt_t bits, mp_bitcount_t bits2)

static fmpz_matxx fmpz_matxx::randtrulike(slong r, slong c,
    frandxx& state, mp_bitcnt_t bits, ulong q)

static fmpz_matxx fmpz_matxx::randtrulike2(slong r, slong
    c, frandxx& state, mp_bitcnt_t bits, ulong q)

static fmpz_matxx fmpz_matxx::randajtai(slong r, slong c,
    frandxx& state, mp_bitcnt_t bits, double alpha)

static fmpz_matxx fmpz_matxx::randrank(slong r, slong c,
    frandxx& state, slong rank, mp_bitcnt_t bits)

static fmpz_matxx fmpz_matxx::randdet(slong r, slong c,
    frandxx& state, Fmpz_expr d)
```

Static versions of the above, where the first two arguments specify the dimensions of the matrix.

```
int Fmpz_mat_target::set_randpermdiag(frandxx& state, Vec v)
```

See `fmpz_mat_randpermdiag`. The type `vec` must have methods `_array()` and `size()` similar to `fmpz_vecxx`.

```
void Fmpz_mat_target::apply_randops(frandxx& state, slong
    count)
```

See `fmpz_mat_randops`.

63.10.9 Transpose

```
Fmpz_expr transpose(Fmpz_mat_expr)
```

63.10.10 Modular reduction and reconstruction

To reduce a single matrix modulo a word-sized modulus, see `nmod_matxx::reduce`.

We use a special class `nmod_mat_vector` to represent a vector of matrices reduced with respect to differing moduli.

```
Fmpz_mat_expr Fmpz_mat_expr::CRT(Fmpz_expr, Nmod_mat_expr,
    bool)

Fmpz_mat_expr CRT(Fmpz_mat_expr, Fmpz_expr, Nmod_mat_expr,
    bool)
```

See `fmpz_mat_CRT_ui`.

```
nmod_mat_vector::nmod_mat_vector(slong rows, slong cols,
    const std::vector<mp_limb_t>& primes)
```

Initialize a vector of matrices with dimensions given by `rows`, `cols` and moduli given by `primes`.

```
nmod_matxx_ref nmod_mat_vector::operator[](std::size_t idx)

nmod_matxx_srcref nmod_mat_vector::operator[](std::size_t
    idx) const
```

Obtain a reference to one of the stored matrices.

```
std::size_t nmod_mat_vector::size() const
```

Obtain the number of stored matrices.

```
void nmod_mat_vector::set_multi_mod(Fmpz_mat_expr m)
```

Reduce `m` modulo each of the primes stored in this vector, and store the results. See `fmpz_mat_multi_mod_ui`.

```
void nmod_mat_vector::set_multi_mod_precomp(Fmpz_mat_expr
    m, const fmpz_combxx& comb)
```

Reduce `m` modulo each of the primes stored in this vector, and store the results. Use precomputed data in `comp`. See `fmpz_mat_multi_mod_ui_precomp`.

```
nmod_mat_vector multi_mod(Fmpz_mat_expr m, const
    std::vector<mp_limb_t>& primes)

nmod_mat_vector multi_mod_precomp(Fmpz_mat_expr m, const
    std::vector<mp_limb_t>& primes, const fmpz_combxx& comb)
```

Convenience functions combining the allocation of memory and modular reduction.

63.10.11 Arithmetic

The overloaded operators `+ - *` can be used for ordinary matrix-matrix and matrix-scalar arithmetic. Additionally, we provide the following functions.

```
Fmpz_mat_expr divexact(Fmpz_mat_expr, Fmpz_expr)

Fmpz_mat_expr divexact(Fmpz_mat_expr, T:is_integer)
```

```
Fmpz_mat_expr mul_classical(Fmpz_mat_expr, Fmpz_mat_expr)

Fmpz_mat_expr mul_multi_mod(Fmpz_mat_expr, Fmpz_mat_expr)

Fmpz_expr sqr(Fmpz_mat_expr)

Fmpz_mat_expr pow(Fmpz_mat_expr, T:is_unsigned_integer)
```

63.10.12 Inverse

```
Ltuple<bool, fmpz_matxx, fmpzxx>_expr inv(Fmpz_mat_expr)
```

`ltupleref(b, M, D)= inv(A)` has the same effect as `b = fmpz_mat_inv(m, d, a)`, where `m, d, a` are the underlying C objects corresponding to `M, D, A`.

63.10.13 Trace

```
Fmpz_mat_expr trace(Fmpz_mat_expr)
```

63.10.14 Determinant

```
Fmpz_expr det(Fmpz_mat_expr)

Fmpz_expr det_cofactor(Fmpz_mat_expr)

Fmpz_expr det_bareiss(Fmpz_mat_expr)

Fmpz_expr det_divisor(Fmpz_mat_expr)

Fmpz_expr det_bound(Fmpz_mat_expr)

Fmpz_expr det_modular(Fmpz_mat_expr, bool proved)

Fmpz_expr det_modular_accelerated(Fmpz_mat_expr, bool
   proved)

Fmpz_expr det_modular_given_divisor(Fmpz_mat_expr,
   Fmpz_expr, bool proved)
```

63.10.15 Characteristic polynomial

```
Fmpz_poly_expr charpoly(Fmpz_mat_expr)
```

63.10.16 Rank

```
slong rank(Fmpz_mat_expr)
```

63.10.17 Non-singular solving

```
Ltuple<bool, fmpz_matxx, fmpzxx>_expr solve( Fmpz_mat_expr
   B, Fmpz_mat_expr X)

Ltuple<bool, fmpz_matxx, fmpzxx>_expr solve_dixon(
   Fmpz_mat_expr B, Fmpz_mat_expr X)
```

```
Ltuple<bool, fmpz_matxx, fmpzxx>_expr solve_cramer(
    Fmpz_mat_expr B, Fmpz_mat_expr X)

Ltuple<bool, fmpz_matxx, fmpzxx>_expr solve_fflu(
    Fmpz_mat_expr B, Fmpz_mat_expr X)
```

`ltupleref(w, M, D)= solve(B, X)` has the same effect as `w = fmpz_mat_solve(m, d, b, x)`, where `m, d, b, x` are the underlying C objects corresponding to `M, D, B, X`. Similarly for the other functions.

```
Ltuple<fmpzxx, fmpzxx>_expr solve_bound( Fmpz_mat_expr B,
    Fmpz_mat_expr X)
```

63.10.18 Row reduction

Beware that compared to the C interface, the flintxx row reduction interface changes some argument orders. This is to facilitate default arguments.

```
slong find_pivot_any(Fmpz_mat_expr, slong, slong, slong)
```

See `fmpz_mat_find_pivot_any`.

```
Ltuple<slong, fmpz_matxx, fmpzxx>_expr fflu(Fmpz_mat_expr
    A, permxx* perm = 0, bool rankcheck = false)
```

See `fmpz_mat_fflu`.

```
Ltuple<slong, fmpz_matxx, fmpzxx>_expr rref(Fmpz_mat_expr A)
```

See `fmpz_mat_rref`.

63.10.19 Modular gaussian elimination

```
slong Fmpz_mat_target::set_rref_mod(Fmpz_expr n, permxx*
    perm = 0)
```

See `fmpz_mat_rref_mod`.

63.10.20 Nullspace

```
Ltuple<slong, fmpz_matxx>_expr nullspace(Fmpz_mat_expr A)
```

`ltupleref(n, B)= nullspace(A)` has the same effect as `n = fmpz_mat_nullspace(b, a)`, where `b, a` are the underlying `fmpz_mat_t` corresponding to `B, A`.

63.11 fmpz_polyxx

63.11.1 C++ particulars

```
Fmpz_poly_expr::unary operation() const
```

The following unary functions are made available as member functions: `derivative`, `primitive_part`, `sqr`, `sqr_classical`, `sqr_karatsuba`, `sqr_KS`, `sqrt`, `sqrt_classical`, `content`, `height`, `bound_roots`, `twonorm`.

```
Fmpz_poly_expr::binary operation(??) const
```

The following binary functions are made available as member functions: `compose_divconquer`, `compose_horner`, `div_basecase`, `div_divconquer`, `divexact`, `divrem`, `divrem_basecase`, `divrem_divconquer`, `div_root`, `evaluate_divconquer`, `evaluate_horner`, `fdiv_2exp`, `gcd`, `gcd_heuristic`, `gcd_modular`, `gcd_subresultant`, `inv_series`, `inv_series_newton`, `lcm`, `mul_2exp`, `mul_classical`, `mul_karatsuba`, `mul_KS`, `mulmid_classical`, `mul_SS`, `shift_left`, `shift_right`, `pow`, `pow_addchains`, `pow_binexp`, `pow_binomial`, `pow_multinomial`, `pseudo_div`, `pseudo_divrem`, `pseudo_divrem_basecase`, `pseudo_divrem_cohen`, `pseudo_divrem_divconquer`, `pseudo_rem`, `pseudo_rem_cohen`, `resultant`, `reverse`, `revert_series`, `revert_series_lagrange`, `revert_series_lagrange_fast`, `revert_series_newton`, `smod`, `sqrlow`, `sqrlow_classical`, `sqrlow_karatsuba_n`, `sqrlow_KS`, `taylor_shift`, `taylor_shift_horner`, `taylor_shift_divconquer`, `tdiv`, `tdiv_2exp`, `xgcd`, `xgcd_modular`, `divides`.

```
Fmpz_poly_expr::ternary operation(??, ??) const
```

The following ternary functions are made available as member functions: `compose_series`, `compose_series_brent_kung`, `compose_horner`, `div_series`, `mulhigh_classical`, `mulhigh_karatsuba_n`, `mulhigh_n`, `mullow`, `mullow_classical`, `mullow_karatsuba_n`, `mullow_KS`, `mullow_SS`, `pow_trunc`.

```
Fmpz_poly_expr Fmpz_poly_expr::operator()(Fmpz_poly_expr)
   const

Fmpz_poly_expr Fmpz_poly_expr::operator()(Fmpz_expr) const
```

Overloaded `operator()` for evaluation or composition.

63.11.2 Memory management

```
fmpz_polyxx::fmpz_polyxx()

fmpz_polyxx::fmpz_polyxx(slong alloc)
```

See `fmpz_poly_init2`.

```
fmpz_polyxx::fmpz_polyxx(const char* str)
```

See `fmpz_poly_set_str`.

```
void Fmpz_poly_target realloc(slong alloc)

void Fmpz_poly_target::fit_length(slong len)

void Fmpz_poly_target::_normalise()

void Fmpz_poly_target::_set_length(slong len)
```

63.11.3 Polynomial parameters

```
slong Fmpz_poly_expr::length() const

slong Fmpz_poly_expr::degree() const
```

63.11.4 Assignment and basic manipulation

```
Fmpz_poly_target Fmpz_poly_target::operator=(T:is_integer)

Fmpz_poly_target Fmpz_poly_target::operator=(Fmpz_expr)
```

```
Fmpz_poly_target Fmpz_poly_target::operator=(const char*)

std::string Fmpz_poly_expr::to_string() const

std::string Fmpz_poly_expr::pretty(const char* x) const
```

See `fmpz_poly_get_str_pretty`.

```
void Fmpz_poly_target::set_zero()

void Fmpz_poly_target::set_one()

static fmpz_polyxx fmpz_polyxx::zero()

static fmpz_polyxx fmpz_polyxx::one()

void Fmpz_poly_target::zero_coeffs(slong i, slong j)

Fmpz_poly_expr reverse(Fmpz_poly_expr, T:fits_into_slong)

void Fmpz_poly_target::truncate(slong)
```

63.11.5 Randomisation

```
static fmpz_polyxx fmpz_polyxx::randtest( frandxx& state,
   slong len, mp_bitcnt_t bits)

static fmpz_polyxx fmpz_polyxx::randtest_unsigned( frandxx&
   state, slong len, mp_bitcnt_t bits)

static fmpz_polyxx fmpz_polyxx::randtest_not_zero( frandxx&
   state, slong len, mp_bitcnt_t bits)
```

See `fmpz_poly_randtest` etc.

63.11.6 Getting and setting coefficients

```
Fmpz_expr Fmpz_poly_expr::get_coeff(slong n)
```

Obtain coefficient n of the polynomial. It is valid to call this with n greater than the degree, in which case zero is returned.

```
void Fmpz_poly_target::set_coeff(slong n, Fmpz_expr)

void Fmpz_poly_target::set_coeff(slong n, T:is_integer)

?? Fmpz_poly_expr::coeff(slong n) const
```

Unified coefficient access for coefficient n. The result is undefined if n is greater than the degree of the polynomial (or negative).

If the leading coefficient of the polynomial is set to zero in this way, a call to `_normalise` is necessary.

```
?? Fmpz_poly_expr::lead() const
```

Unified coefficient access for the leading coefficient. The result is undefined if the length of the polynomial is zero.

If this is used to set the leading coefficient to zero, call to `_normalise` is necessary.

63.11.7 Comparison

As usual, `fmpz_polyxx` can be compared using `operator==`. Additionally, the following functions are provided.

```
bool Fmpz_poly_expr::is_one() const

bool Fmpz_poly_expr::is_zero() const

bool Fmpz_poly_expr::is_unit() const
```

63.11.8 Addition and subtraction

The overloaded operators `+ -` can be used for addition, subtraction and negation.

63.11.9 Scalar multiplication and division

The overloaded operators `* /` can be used for scalar multiplication and division, and the operator `\%` for remaindering. For finer control, the following functions are provided.

```
Fmpz_poly_expr mul_2exp(Fmpz_poly_expr,
   T:is_unsigned_integer)

Fmpz_poly_expr fdiv_2exp(Fmpz_poly_expr,
   T:is_unsigned_integer)

Fmpz_poly_expr tdiv(Fmpz_poly_expr, Fmpz_expr)

Fmpz_poly_expr tdiv(Fmpz_poly_expr, T:is_integer)

Fmpz_poly_expr divexact(Fmpz_poly_expr, Fmpz_expr)

Fmpz_poly_expr divexact(Fmpz_poly_expr, T:is_integer)

Fmpz_poly_expr smod(Fmpz_poly_expr, Fmpz_expr)
```

See `fmpz_poly_scalar_smod_fmpz`.

63.11.10 Bit packing

```
Fmpz_expr bit_pack(Fmpz_poly_expr, T:fits_into_mp_bitcnt_t)

static Fmpz_poly_expr fmpz_polyxx::bit_unpack(Fmpz_expr,
   T:fits_into_mp_bitcnt_t)

static Fmpz_poly_expr fmpz_polyxx::bit_unpack_unsigned(
   Fmpz_expr, traits::fits_into_mp_bitcnt_t)
```

63.11.11 Multiplication

The overloaded operator * can also be used for poly-poly multiplication. Additionally, the following functions are provided.

```
Fmpz_poly_expr mul_classical(Fmpz_poly_expr, Fmpz_poly_expr)

Fmpz_poly_expr mulmid_classical(Fmpz_poly_expr,
   Fmpz_poly_expr)

Fmpz_poly_expr mul_karatsuba(Fmpz_poly_expr, Fmpz_poly_expr)

Fmpz_poly_expr mul_SS(Fmpz_poly_expr, Fmpz_poly_expr)

Fmpz_poly_expr mul_KS(Fmpz_poly_expr, Fmpz_poly_expr)

Fmpz_poly_expr mullow(Fmpz_poly_expr, Fmpz_poly_expr, slong)

Fmpz_poly_expr mullow_classical(Fmpz_poly_expr,
   Fmpz_poly_expr, slong)

Fmpz_poly_expr mullow_karatsuba_n(Fmpz_poly_expr,
   Fmpz_poly_expr, slong)

Fmpz_poly_expr mullow_KS(Fmpz_poly_expr, Fmpz_poly_expr,
   slong)

Fmpz_poly_expr mullow_SS(Fmpz_poly_expr, Fmpz_poly_expr,
   slong)

Fmpz_poly_expr mulhigh_n(Fmpz_poly_expr, Fmpz_poly_expr,
   slong)

Fmpz_poly_expr mulhigh_classical(Fmpz_poly_expr,
   Fmpz_poly_expr, slong)

Fmpz_poly_expr mulhigh_karatsuba_n(Fmpz_poly_expr,
   Fmpz_poly_expr, slong)
```

63.11.12 Squaring

```
Fmpz_poly_expr sqr(Fmpz_poly_expr)

Fmpz_poly_expr sqr_KS(Fmpz_poly_expr)

Fmpz_poly_expr sqr_karatsuba(Fmpz_poly_expr)

Fmpz_poly_expr sqr_classical(Fmpz_poly_expr)

Fmpz_poly_expr sqrlow(Fmpz_poly_expr, T:fits_into_slong n)

Fmpz_poly_expr sqrlow_classical(Fmpz_poly_expr,
   T:fits_into_slong n)
```

```
Fmpz_poly_expr sqrlow_KS(Fmpz_poly_expr, T:fits_into_slong
   n)

Fmpz_poly_expr sqrlow_karatsuba_n(Fmpz_poly_expr,
   T:fits_into_slong n)
```

63.11.13 Powering

```
Fmpz_poly_expr pow(Fmpz_poly_expr, T:is_unsigned_integer)

Fmpz_poly_expr pow_multinomial(Fmpz_poly_expr,
   T:is_unsigned_integer)

Fmpz_poly_expr pow_binomial(Fmpz_poly_expr,
   T:is_unsigned_integer)

Fmpz_poly_expr pow_binexp(Fmpz_poly_expr,
   T:is_unsigned_integer)

Fmpz_poly_expr pow_addchains(Fmpz_poly_expr,
   T:is_unsigned_integer)

Fmpz_poly_expr pow_trunc(Fmpz_poly_expr, ulong e, slong n)
```

63.11.14 Shifting

```
Fmpz_poly_expr shift_left(Fmpz_poly_expr, T:fits_into_slong)

Fmpz_poly_expr shift_right(Fmpz_poly_expr,
   T:fits_into_slong)
```

63.11.15 Bit sizes and norms

```
Fmpz_expr height(Fmpz_poly_expr)

Fmpz_expr twonorm(Fmpz_poly_expr)

ulong Fmpz_poly_expr::max_limbs() const

slong Fmpz_poly_expr::max_bits() const
```

63.11.16 Greatest common divisor

```
Fmpz_poly_expr gcd(Fmpz_poly_expr, Fmpz_poly_expr)

Fmpz_poly_expr gcd_subresultant(Fmpz_poly_expr,
   Fmpz_poly_expr)

Fmpz_poly_expr gcd_heuristic(Fmpz_poly_expr, Fmpz_poly_expr)

Fmpz_poly_expr gcd_modular(Fmpz_poly_expr, Fmpz_poly_expr)

Fmpz_poly_expr lcm(Fmpz_poly_expr, Fmpz_poly_expr)

Ltuple<fmpzxx, fmpz_polyxx, fmpz_polyxx>_expr
   xgcd(Fmpz_poly_expr f, Fmpz_poly_expr g)
```

```
Ltuple<fmpzxx, fmpz_polyxx, fmpz_polyxx>_expr
   xgcd_modular(Fmpz_poly_expr f, Fmpz_poly_expr g)
```

`ltupleref(N, Q, R)= xgcd(F, G)` has the same effect as `fmpz_poly_xgcd(n, q, r, f, g)` where `n, q, r, f, g` are the underlying C objects.

```
Fmpz_expr resultant(Fmpz_poly_expr)
```

63.11.17 Gaussian content

```
Fmpz_expr content(Fmpz_poly_expr)
```

```
Fmpz_poly_expr primitive_part(Fmpz_poly_expr)
```

63.11.18 Square-free

```
bool Fmpz_poly_expr::is_squarefree() const
```

63.11.19 Euclidean division

The overloaded operators `/ %` can be used for euclidean division and remainder. Additionally, the following functions are provided.

```
Fmpz_poly_expr div_basecase(Fmpz_poly_expr, Fmpz_poly_expr)
```

```
Fmpz_poly_expr div_divconquer(Fmpz_poly_expr,
   Fmpz_poly_expr)
```

```
Fmpz_poly_expr rem_basecase(Fmpz_poly_expr, Fmpz_poly_expr)
```

```
Ltuple<fmpz_polyxx, fmpz_polyxx>_expr divrem(Fmpz_poly_expr
   A, Fmpz_poly_expr B)
```

```
Ltuple<fmpz_polyxx, fmpz_polyxx>_expr
   divrem_basecase(Fmpz_poly_expr A, Fmpz_poly_expr B)
```

```
Ltuple<fmpz_polyxx, fmpz_polyxx>_expr divrem_divconquer(
   Fmpz_poly_expr A, Fmpz_poly_expr B)
```

`ltupleref(Q, R)= divrem(A, B)` has the same effect as `fmpz_poly_divrem(q, r, a, b)`, where `q, r, a, b` are the underlying `fmpz_poly_t` corresponding to `Q, R, A, B`.

```
Fmpz_poly_expr div_root(Fmpz_poly_expr, Fmpz_expr)
```

63.11.20 Divisibility testing

```
Ltuple<bool, fmpz_polyxx>_expr divides(Fmpz_poly_expr A,
   Fmpz_poly_expr B)
```

`ltupleref(d, Q)= divides(A, B)` sets `d` to `true` and `Q` to `B/A` if `A` divides `B`, and else sets `d` to `false`. See `fmpz_poly_divides`.

63.11.21 Power series division

```
Fmpz_poly_expr inv_series_newton(Fmpz_poly_expr,
   T:fits_into_slong)
```

```
Fmpz_poly_expr inv_series(Fmpz_poly_expr, T:fits_into_slong)

Fmpz_poly_expr div_series(Fmpz_poly_expr, Fmpz_poly_expr,
   slong n)
```

63.11.22 Pseudo division

```
Ltuple<fmpz_polyxx, fmpz_polyxx, ulong>_expr pseudo_divrem(
   Fmpz_poly_expr A, Fmpz_poly_expr B)

Ltuple<fmpz_polyxx, fmpz_polyxx, ulong>_expr
   pseudo_divrem_basecase( Fmpz_poly_expr A, Fmpz_poly_expr
   B)

Ltuple<fmpz_polyxx, fmpz_polyxx, ulong>_expr
   pseudo_divrem_divconquer( Fmpz_poly_expr A,
   Fmpz_poly_expr B)
```

`ltupleref(Q, R, d)= pseudo_divrem(A, B)` has the same effect as `fmpz_poly_pseudo_divrem(q, r, &d, a, b)`, where `q, r, a, b` are the underlying `fmpz_poly_t` corresponding to `Q, R, A, B`.

```
Ltuple<fmpz_polyxx, fmpz_polyxx>_expr
   pseudo_divrem_cohen(Fmpz_poly_expr A, Fmpz_poly_expr B)
```

`ltupleref(Q, R)= pseudo_divrem_cohen(A, B)` has the same effect as `fmpz_poly_pseudo_divrem_cohen(q, r, a, b)`, where `q, r, a, b` are the underlying `fmpz_poly_t` corresponding to `Q, R, A, B`.

```
Ltuple<fmpz_polyxx, ulong>_expr pseudo_div(Fmpz_poly_expr
   A, Fmpz_poly_expr B)

Ltuple<fmpz_polyxx, ulong>_expr pseudo_rem(Fmpz_poly_expr
   A, Fmpz_poly_expr B)
```

`ltupleref(Q, d)= pseudo_div(A, B)` has the same effect as `fmpz_poly_pseudo_div(q, &d, a, b)`, where `q, a, b` are the underlying `fmpz_poly_t` corresponding to `Q, A, B`.

```
Fmpz_poly_expr pseudorem_cohen(Fmpz_poly_expr,
   Fmpz_poly_expr)
```

63.11.23 Derivative

```
Fmpz_poly_expr derivative(Fmpz_poly_expr)
```

63.11.24 Evaluation

The overloaded `operator()` can be used for evaluation. Additionally, the following functions are provided.

```
Fmpz_expr evaluate(Fmpz_poly_expr, Fmpz_expr)

Fmpz_vec_expr evaluate(Fmpz_poly_expr, Fmpz_vec_expr)

Fmpz_expr evaluate_horner(Fmpz_poly_expr, Fmpz_expr)

Fmpz_expr evaluate_divconquer(Fmpz_poly_expr, Fmpz_expr)

mp_limb_t evaluate_mod(Fmpz_poly_expr p, mp_limb_t x,
   mp_limb_t n)
```

63.11.25 Interpolation

```
static Fmpz_poly_expr fmpz_polyxx::interpolate(
   Fmpz_vec_expr xs, Fmpz_vec_expr ys)
```

See `fmpz_poly_interpolate_fmpz_vec`.

63.11.26 Composition.

The overloaded `operator()` can be used for composition. Additionally, the following functions are provided.

```
Fmpz_poly_expr compose(Fmpz_poly_expr, Fmpz_poly_expr)

Fmpz_poly_expr compose_horner(Fmpz_poly_expr,
   Fmpz_poly_expr)

Fmpz_poly_expr compose_divconquer(Fmpz_poly_expr,
   Fmpz_poly_expr)
```

63.11.27 Taylor shift

```
Fmpz_poly_expr taylor_shift(Fmpz_poly_expr, Fmpz_expr)

Fmpz_poly_expr taylor_shift_horner(Fmpz_poly_expr,
   Fmpz_expr)

Fmpz_poly_expr taylor_shift_divconquer(Fmpz_poly_expr,
   Fmpz_expr)
```

63.11.28 Power series composition

```
Fmpz_poly_expr compose_series(Fmpz_poly_expr,
   Fmpz_poly_expr, slong)

Fmpz_poly_expr compose_series_horner(Fmpz_poly_expr,
   Fmpz_poly_expr, slong)

Fmpz_poly_expr compose_series_brent_kung(Fmpz_poly_expr,
   Fmpz_poly_expr, slong)
```

63.11.29 Power series reversion

```
Fmpz_poly_expr revert_series(Fmpz_poly_expr,
   T:fits_into_slong)

Fmpz_poly_expr revert_series_newton(Fmpz_poly_expr,
   T:fits_into_slong)

Fmpz_poly_expr revert_series_lagrange(Fmpz_poly_expr,
   T:fits_into_slong)

Fmpz_poly_expr revert_series_lagrange_fast(Fmpz_poly_expr,
   T:fits_into_slong)
```

63.11.30 Square root

```
Fmpz_poly_expr sqrt(Fmpz_poly_expr p)

Fmpz_poly_expr sqrt_classical(Fmpz_poly_expr p)
```

Compute the square root of `p`, provided `p` is a perfect square. Else raise `flint_exception`. See `fmpz_poly_sqrt`.

63.11.31 Signature

```
void Fmpz_poly_expr::signature(slong& r1, slong& r2) const
```

See `fmpz_poly_signature`.

63.11.32 Hensel lifting

```
Ltuple<fmpz_polyxx, fmpz_polyxx, fmpz_polyxx,
   fmpz_polyxx>_expr hensel_lift( Fmpz_poly_expr f,
   Fmpz_poly_expr g, Fmpz_poly_expr h, Fmpz_poly_expr a,
   Fmpz_poly_expr b, Fmpz_expr p, Fmpz_expr p1)

Ltuple<fmpz_polyxx, fmpz_polyxx>_expr
   hensel_lift_without_inverse( Fmpz_poly_expr f,
   Fmpz_poly_expr g, Fmpz_poly_expr h, Fmpz_poly_expr a,
   Fmpz_poly_expr b, Fmpz_expr p, Fmpz_expr p1)

Ltuple<fmpz_polyxx, fmpz_polyxx>_expr
   hensel_lift_only_inverse( Fmpz_poly_expr G,
   Fmpz_poly_expr H, Fmpz_poly_expr a, Fmpz_poly_expr b,
   Fmpz_expr p, Fmpz_expr p1)
```

See `fmpz_poly_hensel_lift` etc.

```
fmpz_poly_factorxx::set_hensel_lift_once(Fmpz_poly_expr,
   const nmod_poly_factorxx&, slong)

fmpz_poly_factorxx hensel_lift_once(Fmpz_poly_expr, const
   nmod_poly_factorxx&, slong)
```

See `fmpz_poly_hensel_lift_once`. Note that these two functions are defined in the `fmpz_factorxx` module.

63.11.33 Input and output

```
print(Fmpz_poly_expr)

print(FILE*, Fmpz_poly_expr)

print_pretty(Fmpz_poly_expr, const char* var)

print_pretty(FILE*, Fmpz_poly_expr, const char* var)

read(Fmpz_poly_target)

read(FILE*, Fmpz_poly_target)
```

```
read_pretty(Fmpz_poly_target, const char* var)

read_pretty(FILE*, Fmpz_poly_target, const char* var)
```

63.11.34 Modular reduction and reconstruction

For modular reduction, see `nmod_polyxx::reduce`.

```
Fmpz_poly_expr Fmpz_poly_expr::CRT(Fmpz_expr,
   Nmod_poly_expr, bool)

Fmpz_poly_expr CRT(Fmpz_poly_expr, Fmpz_expr,
   Nmod_poly_expr, bool)
```

See `fmpz_poly_CRT_ui`.

63.11.35 Products

```
static Fmpz_poly_expr
   fmpz_polyxx::product_roots(Fmpz_vec_expr xs)
```

See `fmpz_poly_product_roots_fmpz_vec`.

63.11.36 Roots

```
Fmpz_expr bound_roots(Fmpz_poly_expr p)
```

63.12 fmpz_poly_factorxx

```
bool fmpz_poly_factorxx::operator==(const
   fmpz_poly_factorxx&)
```

Compare two factorisations.

```
ulong fmpz_poly_factorxx::size() const
```

Return the number of stored factors.

```
slong fmpz_poly_factorxx::exp(slong i) const

slong& fmpz_poly_factorxx::exp(slong i)
```

Obtain the exponent of the ith factor.

```
fmpz_polyxx_srcref fmpz_poly_factorxx::p(slong i) const

fmpz_polyxx_ref fmpz_poly_factorxx::p(slong i)
```

Obtain the ith factor.

```
fmpzxx_srcref fmpz_poly_factorxx::content() const

fmpzxx_ref fmpz_poly_factorxx::content()
```

Obtain the content of the factorised polynomial.

63.12.1 Memory management

```
fmpz_poly_factorxx::fmpz_poly_factorxx()
```

```
explicit fmpz_poly_factorxx::fmpz_poly_factorxx(slong alloc)
```

Initialise an empty factorisation.

```
fmpz_poly_factorxx::fmpz_poly_factorxx(const
   fmpz_poly_factorxx& o)
```

Copy a factorisation.

```
void fmpz_poly_factorxx::realloc(slong a)

void fmpz_poly_factorxx::fit_length(slong a)
```

63.12.2 Manipulating factors

```
void fmpz_poly_factorxx::insert(Fmpz_poly_expr p, slong e)

void fmpz_poly_factorxx::concat(const fmpz_poly_factorxx&)
```

63.12.3 Factoring algorithms

```
void
   fmpz_poly_factorxx::set_factor_squarefree(Fmpz_poly_expr
   p)

void
   fmpz_poly_factorxx::set_factor_zassenhaus(Fmpz_poly_expr
   p)

void
   fmpz_poly_factorxx::set_factor_zassenhaus_recombination(
   const fmpz_poly_factorxx& lifted_fac, Fmpz_poly_expr F,
   Fmpz_expr P, slong exp)

fmpz_poly_factorxx::factor_squarefree(Fmpz_poly_expr)

fmpz_poly_factorxx::factor_zassenhaus(Fmpz_poly_expr)
```

63.13 fmpqxx

63.13.1 C++ particulars

```
?? Fmpq_expr::num() const

?? Fmpq_expr::den() const
```

Unified coefficient access to numerator and denominator. If this is used to modify the object, a call to `canonicalise()` may be necessary.

63.13.2 Memory management

```
fmpqxx::fmqxx()
```

Initialize to zero.

```
fmpqxx::fmpqxx(Fmpz_src num, Fmpz_src den)
```

```
fmpqxx::fmpqxx(T:fits_into_slong num, U:is_unsigned_integer
   den)
```

Initialize from numerator `num` and denominator `den`.

63.13.3 Canonicalisation

```
void Fmpq_target::canonicalise()

bool Fmpq_src::is_canonical() const
```

63.13.4 Basic assignment

```
Fmpq_expr Fmpq_expr::abs() const

Fmpq_expr abs(Fmpq_expr)

void Fmpq_target::set_zero()

void Fmpq_target::set_one()

static fmpqxx fmpqxx::zero()

static fmpqxx fmpqxx::one()
```

63.13.5 Comparison

The overloaded relational operators can be used for comparison. Additionally, we have the following functions.

```
bool Fmpq_expr::is_zero() const

bool Fmpq_expr::is_one() const

int Fmpq_expr::sgn() const

mp_bitcnt_t Fmpq_expr::height_bits() const

Fmpz_expr Fmpq_expr::height() const

mp_bitcnt_t height_bits(Fmpq_expr)

Fmpq_expr height(Fmpq_expr)
```

63.13.6 Conversion

Conversion can be done using the assignment operator, and through the following functions.

```
static fmpqxx fmpqxx::frac(const T& t, const U& u)
```

Same as `fmpqxx res;res.set_frac(t, u)`.

```
static fmpqxx fmpqxx::integer(const T& t)
```

Same as `fmpqxx res;res.set_integer(t)`.

```
void Fmpq_target::set_frac(const T& t, const U& u)
```

`f.set_frac(t, u)` has the same effect as `f.num()= t;f.den()= u;f.canonicalise()`.

```
void Fmpq_target::set_integer(const T& t)
```

`f.set_integer(t)` has the same effect as `f.num()= t;f.den()= 1u;`

```
std::string Fmpq_expr::to_string(int base = 10) const
```

63.13.7 Input and output

```
int print(Fmpq_expr)
```

```
int print(FILE*, Fmpq_expr)
```

63.13.8 Random number generation

```
static fmpqxx fmpqxx::randbits(frandxx& state)
```

```
static fmpqxx fmpqxx::randtest(frandxx& state)
```

```
static fmpqxx fmpqxx::randtest_not_zero(frandxx& state)
```

63.13.9 Arithmetic

The overloaded operators `+ - * /` can be used for arithmetic. Additionally, we provide the following functions.

```
Fmpq_expr Fmpq_expr::inv() const
```

```
Fmpq_expr Fmpq_expr::pow(T:fits_into_slong) const
```

```
Fmpq_expr inv(Fmpq_expr)
```

```
Fmpq_expr pow(Fmpq_expr, T:fits_into_slong)
```

```
Fmpq_expr operator<<(Fmpq_expr, T:is_integer)
```

```
Fmpq_expr operator>>(Fmpq_expr, T:is_integer)
```

Shift operators are overloaded. See `fmpq_div_2exp` and `fmpq_mul_2exp`.

63.13.10 Modular reduction and rational reconstruction

```
Fmpq_expr operator%(Fmpq_expr, Fmpz_expr)
```

See `fmpq_mod_fmpz`. The modular reduction operator may raise a `flint_exception` if modular inversion is not possible.

```
static Fmpq_expr fmpqxx::reconstruct(Fmpz_expr a, Fmpz_expr
    m, Fmpz_expr N, Fmpz_expr D)
```

```
static Fmpq_expr fmpqxx::reconstruct(Fmpz_expr a, Fmpz_expr
    m)
```

Rational reconstruction. May raise a `flint_exception` if reconstruction is not possible. See `fmpq_reconstruct_fmpz` and `fmpq_reconstruct_fmpz2`.

63.13.11 Rational enumeration

```
Fmpq_expr Fmpq_expr::next_minimal() const
```

```
Fmpq_expr Fmpq_expr::next_signed_minimal() const
```

```
Fmpq_expr Fmpq_expr::next_calkin_wilf() const
```

```
Fmpq_expr Fmpq_expr::next_signed_calkin_wilf() const
```

63.13.12 Continued fractions

```
slong Fmpq_expr::cfrac_bound() const
```

```
template<class Vec> void Fmpq_target::set_cfrac(const Vec&
   v, slong n)
```

Set value to a partial fraction expansion. The same conventions apply to `v` as in the constructor.

```
template<class Vec> static fmpqxx fmpqxx::from_cfrac(const
   Vec& v, slong n)
```

Initialize from a partial fraction expansion. `v` must be an instance of a class which provides a method `_array()` that returns (a pointer to) an array of `fmpz`. One such class is `fmpz_vecxx`. The array must have size (at least) `n`.

63.14 fmpq_matxx

The class `fmpq_matxx` wraps `fmpq_mat_t`. Like `fmpz_matxx`, many operations on `fmpq_matxx` do not support aliasing. The details can be found in the documentation of `fmpq_mat_t`. Since `fmpq_matxx` does not use temporary merging, evaluation of subexpressions never creates new aliases.

```
Fmpq_mat_expr::unary operation() const
```

The following unary functions are made available as member functions: `inv`, `transpose`, `det`, `trace`, `numden_entrywise`, `numden_rowwise`, `numden_colwise`, `numden_matwise`, `num_rowwise`.

```
Fmpq_mat_expr::binary operation(??) const
```

The following binary functions are made available as member functions: `solve_dixon`, `solve_fraction_free`, `mul_cleared`, `mul_direct`.

```
Fmpq_mat_expr operator?(??, ??)
```

Arithmetic operators `+ - * /` are overloaded when provided by `fmpq_mat_t`.

```
Fmpq_mat_expr operator-(Fmpq_mat_expr)
```

The unary negation operator is overloaded.

```
Fmpq_mat_target Fmpq_mat_target::operator=(Fmpz_mat_expr)
```

See `fmpq_mat_set_fmpz_mat`.

63.14.1 Memory management

```
fmpq_matxx::fmpq_matxx(slong m, slong n)
```

See `fmpq_mat_init`.

63.14.2 Input and output

```
int print(Fmpq_mat_expr)
```

63.14.3 Entry access

```
?? Fmpq_mat_expr::at(slong, slong) const
```

Unified coefficient access to the entries of the matrix.

63.14.4 Basic assignment

```
Fmpq_mat_expr transpose(Fmpq_poly_mat_expr)

void Fmpq_mat_target::set_zero()

void Fmpq_mat_target::set_one()

static fmpq_matxx fmpq_matxx::zero(slong rows, slong cols)

static fmpq_matxx fmpq_matxx::one(slong rows, slong cols)
```

63.14.5 Random matrix generation

```
void Fmpq_mat_target::set_randtest(frandxx& state, slong
   len, mp_bitcnt_t)

static fmpq_matxx fmpq_matxx::randtest(slong rows, slong
   cols, frandxx& state, slong len, mp_bitcnt_t)

void Fmpq_mat_target::set_randtest_unsigned(frandxx& state,
   slong len, mp_bitcnt_t)

static fmpq_matxx fmpq_matxx::randtest_unsigned(slong rows,
   slong cols, frandxx& state, slong len, mp_bitcnt_t)
```

63.14.6 Special matrices

```
void Fmpq_target::set_hilbert_matrix()

Fmpq_mat_expr hilbert_matrix(slong m, slong n)
```

63.14.7 Basic properties

```
bool Fmpq_mat_expr::is_zero() const

bool Fmpq_mat_expr::is_empty() const

bool Fmpq_mat_expr::is_square() const

bool Fmpq_mat_expr::is_integral() const
```

63.14.8 Integer matrix conversion

```
static fmpq_matxx fmpq_matxx::frac(Fmpz_mat_expr, Fmpz_expr)
```

```
void Fmpq_mat_target::set_frac(Fmpz_mat_expr, Fmpz_expr)
```

See `fmpq_mat_set_fmpz_mat_div_fmpz`.

```
static fmpq_matxx fmpq_matxx::integer_matrix(Fmpz_mat_expr)
```

See `fmpq_mat_set_fmpz_mat`.

```
Fmpz_mat_expr num_rowwise(Fmpq_mat_expr)
```

This has the effect of calling `fmpq_mat_get_fmpz_mat_rowwise` with second argument `NULL`.

```
Ltuple<fmpz_matxx, fmpz_matxx>_expr
   numden_entrywise(Fmpq_mat_expr)
```

See `fmpq_mat_get_fmpz_mat_entrywise`.

```
Ltuple<fmpz_matxx, fmpzxx>_expr
   numden_matwise(Fmpq_mat_expr)
```

See `fmpq_mat_get_fmpz_mat_matwise`.

```
Ltuple<fmpz_matxx, fmpz_vecxx>_expr
   numden_rowwise(Fmpq_mat_expr)
```

See `fmpq_mat_get_fmpz_mat_rowwise`.

```
Ltuple<fmpz_matxx, fmpz_vecxx>_expr
   numden_colwise(Fmpq_mat_expr)
```

See `fmpq_mat_get_fmpz_mat_colwise`.

63.14.9 Modular reduction and rational reconstruction

To reduce an `fmpq_matxx` modulo an `fmpzxx` to get an `fmpz_matxx`, see `fmpz_matxx::reduce`.

```
static fmpq_matxx fmpq_matxx::reconstruct(Fmpz_mat_expr,
   Fmpz_expr)
```

See `fmpq_mat_set_fmpz_mat_mod_fmpz`.

63.14.10 Matrix multiplication

The overloaded `operator*` can be used for matrix multiplication. Finer control can be obtained using the following functions.

```
Fmpq_mat_expr mul_direct(Fmpq_mat_expr, Fmpq_mat_expr)
```

```
Fmpq_mat_expr mul_cleared(Fmpq_mat_expr, Fmpq_mat_expr)
```

63.14.11 Trace

```
Fmpq_expr trace(Fmpq_mat_expr)
```

63.14.12 Determinant

```
Fmpq_expr det(Fmpq_mat_expr)
```

63.14.13 Nonsingular solving

```
Fmpq_mat_expr solve_dixon(Fmpq_mat_expr B, Fmpq_mat_expr X)

Fmpq_mat_expr solve_fraction_free(Fmpq_mat_expr B,
   Fmpq_mat_expr X)
```

See `fmpq_mat_solve_dixon` and `fmpq_mat_solve_fraction_free`. Raises `flint_exception` if B is singular.

63.14.14 Inverse

```
Fmpq_mat_expr inv(Fmpq_mat_expr A)
```

Compute the inverse of the square matrix A. Raises `flint_exception` if A is singular. The modulus is required to be prime.

63.14.15 Echelon form

```
bool Fmpq_mat_target::pivot(slong r, slong c, permxx* perm
   = 0)
```

See `fmpq_mat_pivot`.

```
Ltuple<slong, fmpq_matxx>_expr rref(Fmpq_mat_expr)

Ltuple<slong, fmpq_matxx>_expr rref_classical(Fmpq_mat_expr)

Ltuple<slong, fmpq_matxx>_expr
   rref_fraction_free(Fmpq_mat_expr)
```

See `fmpq_mat_rref` etc.

63.15 fmpq_polyxx

63.15.1 C++ particulars

```
Fmpq_poly_expr Fmpq_poly_expr::operator()(Fmpq_poly_expr)
   const

Fmpq_poly_expr Fmpq_poly_expr::operator()(Fmpq_expr) const
```

Overloaded `operator()` for evaluation or composition.

```
Fmpq_poly_expr::unary operation() const
```

The following unary functions are made available as member functions: `derivative`, `integral`, `inv`, `make_monic`, `primitive_part`, `content`.

```
Fmpq_poly_expr::binary operation(??) const
```

The following binary functions are made available as member functions: `asinh_series`, `asin_series`, `atanh_series`, `atan_series`, `cosh_series`, `cos_series`, `divrem`, `exp_series`, `gcd`, `inv_series`, `inv_series_newton`, `lcm`, `log_series`, `pow`, `resultant`, `reverse`, `revert_series`, `revert_series_lagrange`, `revert_series_lagrange_fast`, `revert_series_newton`, `sinh_series`, `tanh_series`, `tan_series`, `xgcd`, `rescale`, `shift_left`, `shift_right`.

```
Fmpq_poly_expr::ternary operation(??, ??) const
```

The following ternary functions are made available as member functions: `compose_series`, `compose_series_brent_kung`, `compose_series_horner`, `div_series`, `mullow`.

63.15.2 Memory management

```
fmpq_polyxx::fmpq_polyxx()
```

```
fmpq_polyxx::fmpq_polyxx(slong alloc)
```

See `fmpq_poly_init2`.

```
fmpq_polyxx::fmpq_polyxx(const char* str)
```

See `fmpq_poly_set_str`.

```
void Fmpq_poly_target realloc(slong alloc)
```

```
void Fmpq_poly_target::fit_length(slong len)
```

```
void Fmpq_poly_target::_normalise()
```

```
void Fmpq_poly_target::_set_length(slong len)
```

```
void Fmpq_poly_target::canonicalise()
```

```
bool Fmpq_poly_src::is_canonical() const
```

63.15.3 Polynomial parameters

```
slong Fmpq_poly_expr::length() const
```

```
slong Fmpq_poly_expr::degree() const
```

63.15.4 Accessing the numerator and denominator

```
fmzqxx_ref Fmpq_poly_target::get_coeff_numref(slong n)
```

```
fmzqxx_srcref Fmpq_poly_src::get_coeff_numref(slong n) const
```

Obtain a reference to the numerator of coefficient n. The result is undefined if n is greater than the degree of the polynomial (or negative). If this is used to modify the object, a call to `canonicalise()` may be necessary. (No unified access, see `get_coeff`.)

```
?? Fmpq_poly_expr::den() const
```

Unified coefficient access to the denominator of the polynomial. If this is used to modify the object, a call to `canonicalise()` may be necessary.

63.15.5 Random testing

```
static fmpq_polyxx fmpq_polyxx::randtest( frandxx& state,
   slong len, mp_bitcnt_t bits)

static fmpq_polyxx fmpq_polyxx::randtest_unsigned( frandxx&
   state, slong len, mp_bitcnt_t bits)

static fmpq_polyxx fmpq_polyxx::randtest_not_zero( frandxx&
   state, slong len, mp_bitcnt_t bits)
```

See `fmpq_poly_randtest` etc.

63.15.6 Assignment

```
Fmpq_poly_target Fmpq_poly_target::operator=(T:is_integer)

Fmpq_poly_target Fmpq_poly_target::operator=(Fmpq_expr)

Fmpq_poly_target Fmpq_poly_target::operator=(Fmpz_expr)

Fmpq_poly_target Fmpq_poly_target::operator=(Fmpz_poly_expr)

Fmpq_poly_target Fmpq_poly_target::operator=(const char*)

void Fmpq_poly_target::set_zero()

void Fmpq_poly_target::set_one()

static fmpq_polyxx fmpq_polyxx::zero()

static fmpq_polyxx fmpq_polyxx::one()

Fmpq_poly_expr inv(Fmpq_poly_expr)

static fmpq_polyxx fmpq_polyxx::get_slice(Fmpq_poly_expr,
   slong i, slong j)

void Fmpq_poly_target::truncate(slong)

Fmpq_poly_expr reverse(Fmpq_poly_expr, T:fits_into_slong)

std::string Fmpq_poly_expr::pretty(const char* x) const
```

See `fmpq_poly_get_str_pretty`.

```
std::string Fmpq_poly_expr::to_string() const
```

63.15.7 Getting and setting coefficients

```
Fmpqxx_expr Fmpq_poly_expr::get_coeff(slong n) const

void Fmpq_poly_target::set_coeff(slong n, Fmpz_expr)

void Fmpq_poly_target::set_coeff(slong n, Fmpq_expr)

void Fmpq_poly_target::set_coeff(slong n, T:is_integer)
```

63.15.8 Comparison

The overloaded operators `==` `!=` `>=` `>` etc. can be used for comparison. Additionally, we have the following functions.

```
bool Fmpq_poly_expr::is_one() const

bool Fmpq_poly_expr::is_zero() const
```

63.15.9 Arithmetic

The overloaded operators `* / + -` can be used for both polynomial-polynomial and polynomial-scalar arithmetic. Additionally, we provide the following functions.

```
Fmpq_poly_expr mullow(Fmpq_poly_expr, Fmpq_poly_expr, slong)
```

63.15.10 Powering

```
Fmpq_poly_expr pow(Fmpq_poly_expr, T:is_unsigned_integer)
```

63.15.11 Shifting

```
Fmpq_poly_expr shift_left(Fmpq_poly_expr, T:fits_into_slong)

Fmpq_poly_expr shift_right(Fmpq_poly_expr,
   T:fits_into_slong)
```

63.15.12 Euclidean division

The overloaded operators `/ %` can be used for euclidean division and remainder. Additionally, we have the following functions.

```
Ltuple<fmpq_polyxx, fmpq_polyxx>_expr divrem(Fmpq_poly_expr
   A, Fmpq_poly_expr B)
```

`ltupleref(Q, R)= divrem(A, B)` has the same effect as `fmpq_poly_divrem(q, r, a, b)` where `q, r, a, b` are the underlying `fmpq_poly_t` corresponding to `Q, R, A, B`.

63.15.13 Power series division

```
Fmpq_poly_expr inv_series_newton(Fmpq_poly_expr,
   T:fits_into_slong)

Fmpq_poly_expr inv_series(Fmpq_poly_expr, T:fits_into_slong)

Fmpq_poly_expr div_series(Fmpq_poly_expr, Fmpq_poly_expr,
   slong n)
```

63.15.14 Greatest common divisor

```
Fmpq_poly_expr gcd(Fmpq_poly_expr, Fmpq_poly_expr)

Fmpq_poly_expr lcm(Fmpq_poly_expr, Fmpq_poly_expr)

Ltuple<fmpq_polyxx, fmpq_polyxx, fmpq_polyxx>_expr xgcd(
   Fmpq_poly_expr f, Fmpq_poly_expr g)
```

`ltupleref(G, S, T)= xgcd(A, B)` has the same effect as `fmpq_poly_xgcd(g, s, t, a, b)`, where `g, s, t, a, b` denote the underlying `fmpq_poly_t` corresponding to `G, S, T, A, B`.

```
Fmpq_expr resultant(Fmpq_poly_expr)
```

63.15.15 Derivative and integral

```
Fmpq_poly_expr derivative(Fmpq_poly_expr)

Fmpq_poly_expr integral(Fmpq_poly_expr)
```

63.15.16 Square roots

```
Fmpq_poly_expr sqrt_series(Fmpq_poly_expr,
   T:fits_into_slong)

Fmpq_poly_expr invsqrt_series(Fmpq_poly_expr,
   T:fits_into_slong)
```

63.15.17 Transcendental functions

```
Fmpq_poly_expr exp_series(Fmpq_poly_expr, T:fits_into_slong)

Fmpq_poly_expr log_series(Fmpq_poly_expr, T:fits_into_slong)

Fmpq_poly_expr atan_series(Fmpq_poly_expr,
   T:fits_into_slong)

Fmpq_poly_expr atanh_series(Fmpq_poly_expr,
   T:fits_into_slong)

Fmpq_poly_expr asin_series(Fmpq_poly_expr,
   T:fits_into_slong)

Fmpq_poly_expr asinh_series(Fmpq_poly_expr,
   T:fits_into_slong)

Fmpq_poly_expr tan_series(Fmpq_poly_expr, T:fits_into_slong)

Fmpq_poly_expr sin_series(Fmpq_poly_expr, T:fits_into_slong)

Fmpq_poly_expr cos_series(Fmpq_poly_expr, T:fits_into_slong)

Fmpq_poly_expr sinh_series(Fmpq_poly_expr,
   T:fits_into_slong)

Fmpq_poly_expr cosh_series(Fmpq_poly_expr,
   T:fits_into_slong)

Fmpq_poly_expr tanh_series(Fmpq_poly_expr,
   T:fits_into_slong)
```

63.15.18 Evaluation

The overloaded `operator()` can be used for evaluation. Additionally we have the following.

```
Fmpq_expr evaluate(Fmpq_poly_expr, Fmpq_expr)

Fmpq_expr evaluate(Fmpq_poly_expr, Fmpz_expr)
```

63.15.19 Interpolation

```
static Fmpq_poly_expr fmpq_polyxx::interpolate(
   Fmpz_vec_expr xs, Fmpz_vec_expr ys)
```

See `fmpq_poly_interpolate_fmpq_vec`.

63.15.20 Composition

```
Fmpq_poly_expr compose(Fmpq_poly_expr, Fmpq_poly_expr)

Fmpq_poly_expr rescale(Fmpq_poly_expr, Fmpq_expr)
```

63.15.21 Power series composition

```
Fmpq_poly_expr compose_series(Fmpq_poly_expr,
   Fmpq_poly_expr, slong)

Fmpq_poly_expr compose_series_horner(Fmpq_poly_expr,
   Fmpq_poly_expr, slong)

Fmpq_poly_expr compose_series_brent_kung(Fmpq_poly_expr,
   Fmpq_poly_expr, slong)
```

63.15.22 Power series reversion

```
Fmpq_poly_expr revert_series(Fmpq_poly_expr,
   T:fits_into_slong)

Fmpq_poly_expr revert_series_newton(Fmpq_poly_expr,
   T:fits_into_slong)

Fmpq_poly_expr revert_series_lagrange(Fmpq_poly_expr,
   T:fits_into_slong)

Fmpq_poly_expr revert_series_lagrange_fast(Fmpq_poly_expr,
   T:fits_into_slong)
```

63.15.23 Gaussian content

```
Fmpq_expr content(Fmpq_poly_expr)

Fmpq_poly_expr primitive_part(Fmpq_poly_expr)

bool Fmpq_poly_expr::is_monic() const

Fmpq_poly_expr make_monic(Fmpq_poly_expr)
```

63.15.24 Square-free

```
bool Fmpq_poly_expr::is_squarefree() const
```

63.15.25 Input and output

```
print(Fmpq_poly_expr)
```

```
print(FILE*, Fmpq_poly_expr)

print_pretty(Fmpq_poly_expr, const char* var)

print_pretty(FILE*, Fmpq_poly_expr, const char* var)

read(Fmpq_poly_target)

read(FILE*, Fmpq_poly_target)
```

63.16 fmpz_poly_qxx

63.16.1 Memory management

```
fmpz_poly_qxx::fmpz_poly_qxx()

fmpz_poly_qxx::fmpz_poly_qxx(const char*)
```

See `fmpz_poly_q_set_str`.

```
void Fmpz_poly_q_target::canonicalise()

bool Fmpz_poly_q_src::is_canonical() const

?? Fmpz_poly_q_expr::num() const

?? Fmpz_poly_q_expr::den() const
```

Unified coefficient access to the numerator or denominator of the rational function. If this is used for modification, a call to `canonicalise()` may be necessary.

63.16.2 Randomisation

```
static fmpz_poly_qxx fmpz_poly_qxx::randtest(frandxx&
   state, slong len1, mp_bitcnt_t bits1, slong len2,
   mp_bitcnt_t bits2)

static fmpz_poly_qxx
   fmpz_poly_qxx::randtest_not_zero(frandxx& state, slong
   len1, mp_bitcnt_t bits1, slong len2, mp_bitcnt_t bits2)
```

See `fmpz_poly_q_randtest` etc.

63.16.3 Assignment

```
Fmpz_poly_q_target
   Fmpz_poly_q_target::operator=(T:fits_into_slong)

void Fmpz_poly_q_target::set_zero()

void Fmpz_poly_q_target::set_one()

static fmpz_poly_qxx fmpz_poly_qxx::zero()

static fmpz_poly_qxx fmpz_poly_qxx::one()

Fmpz_poly_q_expr inv(Fmpz_poly_q_expr)
```

```
Fmpz_poly_q_expr Fmpz_poly_q_expr::inv() const
```

63.16.4 Comparison

The overloaded operator == can be used for comparison. Additionally, we have the following functions.

```
bool Fmpz_poly_q_expr::is_one() const

bool Fmpz_poly_q_expr::is_zero() const
```

63.16.5 Powering

```
Fmpz_poly_q_expr pow(Fmpz_poly_q_expr,
   T:is_unsigned_integer)

Fmpz_poly_q_expr
   Fmpz_poly_q_expr::pow(T:is_unsigned_integer) const
```

63.16.6 Derivative

```
Fmpz_poly_q_expr Fmpz_poly_q_expr::derivative() const

Fmpz_poly_q_expr derivative(Fmpz_poly_q_expr)
```

63.16.7 Input and output

```
Fmpz_poly_q_target Fmpz_poly_q_target::operator=(const
   char*)
```

See `fmpz_poly_q_set_str`.

```
std::string Fmpz_poly_q_expr::to_string() const
```

See `fmpz_poly_q_get_str`.

```
std::string Fmpz_poly_q_expr::pretty(const char* x) const
```

See `fmpz_poly_q_get_str_pretty`.

```
int print(Fmpz_poly_q_expr)

int print_pretty(Fmpz_poly_q_expr, const char* var)
```

63.17 fmpz_poly_matxx

The class `fmpz_poly_matxx` wraps `fmpz_poly_mat_t`, and so represents matrices with coefficients in $\mathbb{Z}[X]$. Its usage is similar to `fmpz_matxx` in most regards.

```
Fmpz_poly_mat_expr::unary operation() const
```

The following unary functions are made available as member functions: `det`, `det_fflu`, `det_interpolate`, `trace`, `sqr`, `sqr_classical`, `sqr_KS`, `transpose`.

```
Fmpz_poly_mat_expr::binary operation(??) const
```

The following binary functions are made available as member functions: `solve`, `solve_fflu`, `mul_classical`, `mul_interpolate`, `mul_KS`, `pow`, `sqrlow`.

```
Fmpz_poly_mat_expr::three operation(??) const
```

The following threeary functions are made available as member functions: `mullow`, `pow_trunc`.

```
Fmpz_mat_expr Fmpz_poly_mat_expr::operator()(Fmpz_expr)
    const
```

`operator()` is overloaded for matrix evaluation.

```
Fmpz_poly_mat_expr operator?(??, ??)
```

Arithmetic operators `+ - *` are overloaded when provided by `fmpz_poly_mat_t`.

```
Fmpz_poly_mat_expr operator-(Fmpz_poly_mat_expr)
```

The unary negation operator is overloaded.

63.17.1 Input and output

```
int print_pretty(Fmpz_poly_mat_expr, const char* x)
```

63.17.2 Basic properties

```
slong Fmpz_poly_mat_expr::rows() const

slong Fmpz_poly_mat_expr::cols() const
```

Obtain the number of rows/columns in this matrix. These functions never cause evaluation (the matrix size is computed from the operations in the expression template and the size of the input matrices).

63.17.3 Basic assignment and manipulation

```
?? Fmpz_poly_mat_expr::at(T:fits_into_slong,
    U:fits_into_slong) const
```

Unified coefficient access to the matrix entries.

63.17.4 Standard matrices

```
void Fmpz_poly_mat_target::set_zero()

void Fmpz_poly_mat_target::set_one()

static fmpz_poly_matxx fmpz_poly_matxx::zero(slong rows,
    slong cols)

static fmpz_poly_matxx fmpz_poly_matxx::one(slong rows,
    slong cols)
```

63.17.5 Random matrix generation

```
void Fmpz_poly_mat_target::set_randtest(frandxx& state,
    slong len, mp_bitcnt_t)

void Fmpz_poly_mat_target::set_randtest_unsigned(frandxx&
    state, slong len, mp_bitcnt_t)
```

```
void Fmpz_poly_mat_target::set_randtest_sparse(frandxx&
   state, slong len, mp_bitcnt_t, float)

static fmpz_poly_matxx fmpz_poly_matxx::randtest(slong
   rows, slong cols, frandxx&, slong len, mp_bitcnt_t)

static fmpz_poly_matxx
   fmpz_poly_matxx::randtest_unsigned(slong rows, slong
   cols, frandxx&, slong len, mp_bitcnt_t)

static fmpz_poly_matxx
   fmpz_poly_matxx::randtest_sparse(slong rows, slong cols,
   frandxx&, slong len, mp_bitcnt_t, float density)
```

See `fmpz_poly_mat_randtest` etc.

63.17.6 Basic comparison and properties

```
bool Fmpz_poly_mat_expr::is_zero() const

bool Fmpz_poly_mat_expr::is_one() const

bool Fmpz_poly_mat_expr::is_empty() const

bool Fmpz_poly_mat_expr::is_square() const
```

63.17.7 Norms

```
slong Fmpz_poly_mat_expr::max_length() const

slong Fmpz_poly_mat_expr::max_bits() const
```

63.17.8 Transpose

```
Fmpz_poly_mat_expr transpose(Fmpz_poly_mat_expr)
```

63.17.9 Arithmetic

Basic arithmetic is most easily done using the overloaded operators `+ * -`. Finer control can be obtained using the following functions.

```
Fmpz_mat_expr mul_classical(Fmpz_mat_expr, Fmpz_mat_expr)

Fmpz_mat_expr mul_KS(Fmpz_mat_expr, Fmpz_mat_expr)

Fmpz_poly_mat_expr mullow(Fmpz_poly_mat_expr,
   Fmpz_poly_mat_expr, slong)

Fmpz_poly_mat_expr sqr(Fmpz_poly_mat_expr)

Fmpz_poly_mat_expr sqr_KS(Fmpz_poly_mat_expr)

Fmpz_poly_mat_expr sqr_classical(Fmpz_poly_mat_expr)

Fmpz_poly_mat_expr sqrlow(Fmpz_poly_mat_expr,
   T:fits_into_slong n)
```

```
Fmpz_poly_mat_expr pow(Fmpz_poly_mat_expr,
   T:is_unsigned_integer)

Fmpz_poly_mat_expr pow_trunc(Fmpz_poly_mat_expr,
   T:is_unsigned_integer, T:fits_into_slong)

Fmpz_poly_mat_expr prod(Fmpz_poly_mat_vec_expr)
```

63.17.10 Row reduction

Beware that compared to the C interface, the flintxx row reduction interface changes some argument orders. This is to facilitate default arguments.

```
slong find_pivot_any(Fmpz_poly_mat_expr, slong, slong,
   slong)
```

See `fmpz_poly_mat_find_pivot_any`.

```
slong find_pivot_partial(Fmpz_poly_mat_expr, slong, slong,
   slong)
```

See `fmpz_poly_mat_find_pivot_partial`.

```
Ltuple<slong, fmpz_poly_matxx, fmpzxx>_expr
   fflu(Fmpz_poly_mat_expr A, permxx* perm = 0, bool
   rankcheck = false)
```

See `fmpz_poly_mat_fflu`.

```
Ltuple<slong, fmpz_poly_matxx, fmpzxx>_expr
   rref(Fmpz_poly_mat_expr A)
```

See `fmpz_poly_mat_rref`.

63.17.11 Trace

```
Fmpz_poly_expr trace(Fmpz_poly_mat_expr)
```

63.17.12 Determinant and rank

```
Fmpz_poly_expr det(Fmpz_poly_mat_expr)

Fmpz_poly_expr det_fflu(Fmpz_poly_mat_expr)

Fmpz_poly_expr det_interpolate(Fmpz_poly_mat_expr)

slong rank(Fmpz_poly_mat_expr)
```

63.17.13 Inverse

```
Ltuple<bool, fmpz_poly_matxx, fmpz_polyxx>_expr
   inv(Fmpz_poly_mat_expr)
```

`ltupleref(b, M, D)= inv(A)` has the same effect as `b = fmpz_poly_mat_inv(m, d, a)`, where `m, d, a` are the underlying C objects corresponding to `M, D, A`.

63.17.14 Nullspace

```
Ltuple<slong, fmpz_poly_matxx>_expr
   nullspace(Fmpz_poly_mat_expr A)
```

`ltupleref(n, B)= nullspace(A)` has the same effect as
`n = fmpz_poly_mat_nullspace(b, a)`, where `b, a` are the underlying `fmpz_poly_mat_t` corresponding to `B, A`.

63.17.15 Solving

```
Ltuple<bool, fmpz_poly_matxx, fmpz_polyxx>_expr solve(
   Fmpz_poly_mat_expr B, Fmpz_poly_mat_expr X)

Ltuple<bool, fmpz_poly_matxx, fmpz_polyxx>_expr solve_fflu(
   Fmpz_poly_mat_expr B, Fmpz_poly_mat_expr X)

Ltuple<bool, fmpz_poly_matxx, fmpz_polyxx>_expr
   solve_fflu_precomp( const permxx&, Fmpz_poly_mat_expr B,
   Fmpz_poly_mat_expr FFLU, Fmpz_poly_mat_expr X)
```

`ltupleref(w, M, D)= solve(B, X)` has the same effect as
`w = fmpz_poly_mat_solve(m, d, b, x)`, where `m, d, b, x` are the underlying C objects corresponding to `M, D, B, X`. Similarly for the other functions.

63.18 nmodxx

The class `nmodxx` encapsulates the use of `mp_limb_t` together with `nmod_t` for doing arithmetic modulo a word-sized integer. It is defined in `nmod_vecxx.h`.

The C++ equivalent to `nmod_t` is `nmodxx_ctx`. There is a reference version `nmodxx_ctx_srcref`.

The C++ equivalent to `mp_limb_t` in this context is `nmodxx`. Immediate `nmodxx` expressions store both an `mp_limb_t` and an `nmodxx_ctx_srcref`.

The most common ways to construct `nmodxx` are using the static member functions `nmodxx::red` and `nmodxx::make_nored`. For convenience, `operator%` is overloaded with right hand side `nmodxx_ctx` (or `nmodxx_ctx_srcref`) to call `nmodxx::red`.

Just like when `mp_limb_t` is passed to `nmod_t` operations, the limb stored in `nmodxx` is assumed to be reduced, and under this assumption, all computations yield reduced data.

It is assumed that any expression of `nmodxx` involves only one modulus, so that all contexts are interchangeable.

```
explicit nmodxx_ctx::nmodxx_ctx(mp_limb_t n)
```

Initialise a new context for operations modulo n.

```
nmodxx_ctx_srcref::nmodxx_ctx_srcref(const nmodxx_ctx&)
```

Initialise a reference to an `nmodxx_ctx`.

```
static nmodxx_ctx_srcref::make(const nmod_t& nm)
```

Initialise a reference pointing to an `nmod_t`.

```
const nmod_t& nmodxx_ctx::_nmod() const

const nmod_t& nmodxx_ctx_srcref::_nmod() const
```

Obtain a reference to the underlying `nmod_t`.

```
mp_limb_t nmodxx_ctx::n() const
```

```
mp_limb_t nmodxx_ctx_srcref::n() const
```

Obtain the modulus stored in this context.

```
nmodxx::nmodxx(nmodxx_ctx_srcref ctx)
```

Initialise an `nmodxx` to zero.

```
static nmodxx nmodxx::make_nored(mp_limb_t n,
    nmodxx_ctx_srcref ctx)
```

Initialise an `nmodxx` to n, performing no reductions.

```
static nmodxx nmodxx::red(mp_limb_t n, nmodxx_ctx_srcref
    ctx)
```

```
static nmodxx nmodxx::red(Fmpz_expr n, nmodxx_ctx_srcref
    ctx)
```

```
static nmodxx nmodxx::red(Fmpq_expr n, nmodxx_ctx_srcref
    ctx)
```

Initialise an `nmodxx` to the reduction of n.

```
static nmodxx_ref nmodxx_ref::make(mp_limb_t& l,
    nmodxx_ctx_srcref c)
```

```
static nmodxx_srcref nmodxx_srcref::make(const mp_limb_t&,
    nmodxx_ctx_srcref)
```

Obtain a flintxx reference object pointing to `l`, which is interpreted as a limb reduced modulo `c`.

```
void Nmod_target::reduce()
```

Reduce the stored limb.

```
void Nmod_target::set_nored(mp_limb_t n)
```

Set the stored limb to n.

```
std::string Nmod_expr::to_string() const
```

Convert self into a string of the form "a mod b".

```
mp_limb_t Nmod_expr::to<mp_limb_t>() const
```

Obtain the stored limb.

```
nmodxx_ctx_srcref Nmod_expr::estimate_ctx() const
```

Obtain the context of any immediate subexpression. (By our homogeneity assumptions, the result of this operation does not depend on the subexpression chosen.)

```
Nmod_expr Nmod_expr::inv() const
```

```
Nmod_expr Nmod_expr::pow(T:is_unsigned_integer) const
```

```
Nmod_expr operator??(Nmod_expr, Nmod_expr)
```

Arithmetic operators `+ - * /` are overloaded for nmod expressions.

```
Nmod_expr operator-(Nmod_expr)
```

```
Nmod_expr pow(Nmod_expr, T:is_unsigned_integer)
```

```
Nmod_expr inv(Nmod_expr)
```

63.19 nmod_polyxx

The class `nmod_polyxx` wraps `nmod_poly_t`. Like `nmodxx`, instances of `nmod_polyxx` always have an associated `nmodxx_ctx` storing the operating modulus. No expression may involve more than one modulus at a time.

In order to reduce convert a `fmpz_polyxx` or `fmpq_polyxx` to `nmod_polyxx`, see the `reduce` method of `fmpz_polyxx` or `fmpq_polyxx`, respectively.

`nmodxx_ctx_srcref Nmod_poly_expr::estimate_ctx() const`

Obtain the relevant context. This never causes evaluation.

`Nmod_poly_expr::unary operation() const`

The following unary functions are made available as member functions: `derivative`, `integral`, `make_monic`, `sqrt`.

`Nmod_poly_expr::binary operation() const`

The following binary functions are made available as member functions:
`compose_divconquer`, `compose_horner`, `div_basecase`,
`div_divconquer`, `div_newton`, `divrem`,
`divrem_basecase`, `divrem_divconquer`,
`divrem_newton`, `div_root`, `evaluate_fast`,
`evaluate_iter`, `gcd`, `gcd_euclidean`, `gcd_hgcd`,
`inv_series`, `inv_series_basecase`, `inv_series_newton`,
`invsqrt_series`, `mul_classical`, `mul_KS`,
`shift_left`, `shift_right`, `pow`,
`pow_binexp`, `rem_basecase`, `resultant`,
`resultant_euclidean`, `reverse`, `revert_series`,
`revert_series_lagrange`, `revert_series_lagrange_fast`,
`revert_series_newton`, `sqrt_series`, `taylor_shift`,
`taylor_shift_convolution`, `taylor_shift_horner`, `xgcd`,
`xgcd_euclidean`, `xgcd_hgcd`, `log_series`,
`exp_series`, `exp_series_basecase`, `atan_series`,
`atanh_series`, `asin_series`, `asinh_series`,
`sin_series`, `cos_series`, `tan_series`,
`sinh_series`, `cosh_series`, `tanh_series`.

```
Nmod_poly_expr
    Nmod_poly_expr::inflate(T:is_unsigned_integer) const
```

See `inflate`.

```
Nmod_poly_expr
    Nmod_poly_expr::deflate(T:is_unsigned_integer) const
```

See `deflate`.

`Nmod_poly_expr::ternary operation(??, ??) const`

The following ternary functions are made available as member functions:
`compose_mod`, `compose_mod_horner`,
`compose_series_brent_kung`, `compose_series`,
`compose_series_brent_kung`, `compose_series_divconquer`,
`compose_series_horner`, `div_newton_n_preinv`,
`divrem_newton_n_preinv`, `div_series`, `mulhigh`,
`mulhigh_classical`, `mullow`, `mullow_classical`,
`mullow_KS`, `mulmod`, `powmod_binexp`, `pow_trunc`,
`pow_trunc_binexp`.

```
Nmod_poly_expr::fourary operation(??, ??, ??) const
```

The following functions of four arguments are made available as member functions: `compose_mod_brent_kung_preinv`, `mulmod_preinv`, `powmod_binexp_preinv`.

```
Nmod_poly_expr Nmod_poly_expr::operator()(Nmod_poly_expr)
    const

Nmod_expr Nmod_poly_expr::operator()(Nmod_expr) const

Nmod_vec_expr Nmod_poly_expr::operator()(Nmod_vec_expr)
    const
```

The `operator()` is overloaded for evaluation or composition, depending on the argument.

```
Nmod_poly_expr operator?(??, ??)
```

Arithmetic operators `+ - * / %` are overloaded when provided by `nmod_poly_t`.

```
Nmod_poly_expr operator-(Nmod_poly_expr)
```

The unary negation operator is overloaded.

```
Nmod_poly_target Nmod_poly_target::operator=(const char*)
```

See `nmod_poly_set_str`. Raises `flint_exception` if the string is malformed.

63.19.1 Conversion

```
static nmod_polyxx nmod_polyxx::reduce(Fmpz_mod_poly_expr,
    nmodxx_ctx_srcref)

static nmod_polyxx nmod_polyxx::reduce(Fmpq_mod_poly_expr,
    nmodxx_ctx_srcref)

static nmod_polyxx nmod_polyxx::reduce(Fmpz_mod_poly_expr,
    mp_limb_t)

static nmod_polyxx nmod_polyxx::reduce(Fmpq_mod_poly_expr,
    mp_limb_t)
```

See `fmpz_poly_get_nmod_poly`.

```
static nmod_polyxx nmod_polyxx::from_ground(Nmod_expr e)

static nmod_polyxx nmod_polyxx::from_ground(mp_limb_t e,
    nmodxx_ctx_srcref c)
```

Consider $e \in \mathbb{Z}/n\mathbb{Z}$ as an element of $\mathbb{Z}/n\mathbb{Z}[X]$.

63.19.2 Input and output

```
print(Nmod_poly_expr)

print(FILE*, Nmod_poly_expr)

read(Nmod_poly_target)
```

```
read(FILE*, Nmod_poly_target)
```

63.19.3 Memory management

```
nmod_polyxx::nmod_polyxx(mp_limb_t modulus)

nmod_polyxx::nmod_polyxx(mp_limb_t modulus, slong alloc)

nmod_polyxx::nmod_polyxx(nmodxx_ctx_srcref ctx)

nmod_polyxx::nmod_polyxx(nmodxx_ctx_srcref ctx, slong alloc)
```

Instantiate `nmod_polyxx` relative to some modulus. If the second argument is provided, space is allocated for `alloc` coefficients.

```
nmod_polyxx::nmod_polyxx(const char* str)
```

Instantiate `nmod_polyxx` from a string representation. The modulus is parsed (second integer in the string) and the polynomial is initialised with this modulus, then `nmod_poly_set_str` is called. Raises `flint_exception` if the string is malformed.

```
static nmod_polyxx nmod_polyxx::zero(mp_limb_t n)

static nmod_polyxx nmod_polyxx::one(mp_limb_t n)

void Nmod_poly_target realloc(slong alloc)

void Nmod_poly_target::fit_length(slong len)

void Nmod_poly_target::_normalise()
```

63.19.4 Polynomial properties

```
sslong Nmod_poly_expr::length() const

sslong Nmod_poly_expr::degree() const

sslong Nmod_poly_expr::max_bits() const

mp_limb_t Nmod_poly_expr::modulus() const
```

63.19.5 Assignment and basic manipulation

```
void Nmod_poly_target::truncate(slong)

void Nmod_poly_target::set_zero()

void Nmod_poly_target::set_one()

Nmod_poly_expr reverse(Nmod_poly_expr, T:fits_into_slong)
```

63.19.6 Randomisation

```
void Nmod_target::set_randtest(frandxx& state, slong len)

void Nmod_target::set_randtest_irreducible(frandxx& state,
   slong len)
```

```
static nmod_polyxx nmod_polyxx::randtest(mp_limb_t n,
   frandxx& state, slong len)

static nmod_polyxx
   nmod_polyxx::randtest_irreducible(mp_limb_t n, frandxx&
   state, slong len)
```

63.19.7 Getting and setting coefficients

```
Nmodxx_expr Nmod_poly_expr::get_coeff(slong n) const

void Nmod_target::set_coeff(slong i, Nmodxx_expr)

void Nmod_target::set_coeff(slong i, mp_limb_t)
```

63.19.8 Input and output

```
std::string Nmod_poly_expr::to_string() const

std::ostream& operator<<(std::ostream&, Nmod_poly_expr)
```

Output to streams is done by first converting to string.

63.19.9 Comparison

```
bool Nmod_poly_expr::is_one() const

bool Nmod_poly_expr::is_zero() const

bool operator==(Nmod_poly_expr, Nmod_poly_expr)
```

63.19.10 Scalar multiplication and division

Scalar multiplication is provided via overloaded `operator*`. Additionally, the following functions are implemented:

```
Nmod_poly_expr make_monic(Nmod_poly_expr)
```

63.19.11 Bit packing and unpacking

```
Fmpz_expr Nmod_poly_expr::bit_pack(T:fits_into_mp_bitcnt_t)
   const

static nmod_polyxx nmod_polyxx::bit_unpack( Fmpz_expr,
   T:fits_into_mp_bitcnt_t) const
```

63.19.12 Multiplication

Basic multiplication is provided via overloaded `operator*`. Finer control can be obtained using the following functions.

```
Nmod_poly_expr mul_classical(Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr mul_KS(Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr mullow(Nmod_poly_expr, Nmod_poly_expr, slong)
```

```
Nmod_poly_expr mullow_classical(Nmod_poly_expr,
   Nmod_poly_expr, slong)

Nmod_poly_expr mullow_KS(Nmod_poly_expr, Nmod_poly_expr,
   slong)

Nmod_poly_expr mulhigh(Nmod_poly_expr, Nmod_poly_expr,
   slong)

Nmod_poly_expr mulhigh_classical(Nmod_poly_expr,
   Nmod_poly_expr, slong)

Nmod_poly_expr mulmod(Nmod_poly_expr, Nmod_poly_expr,
   Nmod_poly_expr)

Nmod_poly_expr mulmod_preinv(Nmod_poly_expr,
   Nmod_poly_expr, Nmod_poly_expr, Nmod_poly_expr)
```

63.19.13 Powering

```
Nmod_poly_expr pow(Nmod_poly_expr, T:is_unsigned_integer)

Nmod_poly_expr pow_binexp(Nmod_poly_expr,
   T:is_unsigned_integer)

Nmod_poly_expr pow_trunc(Nmod_poly_expr,
   T:is_unsigned_integer, T:fits_into_slong)

Nmod_poly_expr pow_trunc_binexp(Nmod_poly_expr,
   T:is_unsigned_integer, T:fits_into_slong)

Nmod_poly_expr powmod_binexp(Nmod_poly_expr,
   T:is_unsigned_integer, Nmod_poly_expr)

Nmod_poly_expr powmod_binexp_preinv(Nmod_poly_expr,
   T:is_unsigned_integer, Nmod_poly_expr, Nmod_poly_expr)
```

63.19.14 Division

Basic division and remainder is provided by overloaded `operator/` and `operator%`. Finer control can be obtained using the following functions.

```
Ltuple<nmod_polyxx, nmod_polyxx>_expr divrem(Nmod_poly_expr
   A, Nmod_poly_expr B)

Ltuple<nmod_polyxx, nmod_polyxx>_expr
   divrem_basecase(Nmod_poly_expr A, Nmod_poly_expr B)

Ltuple<nmod_polyxx, nmod_polyxx>_expr divrem_divconquer(
   Nmod_poly_expr A, Nmod_poly_expr B)

Ltuple<nmod_polyxx, nmod_polyxx>_expr divrem_newton(
   Nmod_poly_expr A, Nmod_poly_expr B)
```

```
Ltuple<nmod_polyxx, nmod_polyxx>_expr
   divrem_newton_n_preinv( Nmod_poly_expr A, Nmod_poly_expr
   B)

Nmod_poly_expr div_basecase(Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr div_divconquer(Nmod_poly_expr,
   Nmod_poly_expr)

Nmod_poly_expr div_newton(Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr div_newton_n_preinv(Nmod_poly_expr,
   Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr rem_basecase(Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr inv_series(Nmod_poly_expr, T:fits_into_slong)

Nmod_poly_expr inv_series_basecase(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr inv_series_newton(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr div_series(Nmod_poly_expr, Nmod_poly_expr,
   slong n)

Nmod_poly_expr div_root(Nmod_poly_expr, Nmod_expr)
```

63.19.15 Derivative and integral

```
Nmod_poly_expr derivative(Nmod_poly_expr)

Nmod_poly_expr integral(Nmod_poly_expr)
```

63.19.16 Evaluation

Basic evaluation and multipoint evaluation can be achieved using the overloaded `operator()`. Finer control can be obtained using the following functions.

```
Nmod_expr evaluate(Nmod_poly_expr, Nmod_expr)

Nmod_vec_expr evaluate(Nmod_poly_expr, Nmod_vec_expr)

Nmod_vec_expr evaluate_fast(Nmod_poly_expr, Nmod_vec_expr)

Nmod_vec_expr evaluate_iter(Nmod_poly_expr, Nmod_vec_expr)
```

63.19.17 Interpolation

```
static Nmod_poly_expr fmpz_polyxx::interpolate(
   Nmod_vec_expr xs, Nmod_vec_expr ys)

static Nmod_poly_expr fmpz_polyxx::interpolate_barycentric(
   Nmod_vec_expr xs, Nmod_vec_expr ys)

static Nmod_poly_expr fmpz_polyxx::interpolate_fast(
   Nmod_vec_expr xs, Nmod_vec_expr ys)

static Nmod_poly_expr fmpz_polyxx::interpolate_newton(
   Nmod_vec_expr xs, Nmod_vec_expr ys)
```

63.19.18 Composition

Basic composition can be achieved with the overloaded `operator()`. Finer control can be obtained using the following functions.

```
Nmod_poly_expr compose(Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr compose_horner(Nmod_poly_expr,
   Nmod_poly_expr)

Nmod_poly_expr compose_divconquer(Nmod_poly_expr,
   Nmod_poly_expr)
```

63.19.19 Taylor Shift

```
Nmod_poly_expr taylor_shift(Nmod_poly_expr, Nmod_expr)

Nmod_poly_expr taylor_shift_horner(Nmod_poly_expr,
   Nmod_expr)

Nmod_poly_expr taylor_shift_convolution(Nmod_poly_expr,
   Nmod_expr)
```

63.19.20 Modular composition

```
Nmod_poly_expr compose_mod(Nmod_poly_expr, Nmod_poly_expr,
   Nmod_poly_expr)

Nmod_poly_expr compose_mod_horner(Nmod_poly_expr,
   Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr compose_mod_divconquer(Nmod_poly_expr,
   Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr compose_mod_brent_kung(Nmod_poly_expr,
   Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr
   compose_mod_brent_kung_preinv(Nmod_poly_expr,
   Nmod_poly_expr, Nmod_poly_expr, Nmod_poly_expr)
```

63.19.21 Greatest common divisor

```
Nmod_poly_expr gcd(Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr gcd_euclidean(Nmod_poly_expr, Nmod_poly_expr)

Nmod_poly_expr gcd_hgcd(Nmod_poly_expr, Nmod_poly_expr)

Ltuple<nmod_polyxx, nmod_polyxx, nmod_polyxx>_expr xgcd(
   Nmod_poly_expr, Nmod_poly_expr)

Ltuple<nmod_polyxx, nmod_polyxx, nmod_polyxx>_expr
   xgcd_euclidean( Nmod_poly_expr, Nmod_poly_expr)

Ltuple<nmod_polyxx, nmod_polyxx, nmod_polyxx>_expr
   xgcd_hgcd( Nmod_poly_expr, Nmod_poly_expr)

Nmod_expr resultant(Nmod_poly_expr, Nmod_poly_expr)

Nmod_expr resultant_euclidean(Nmod_poly_expr,
   Nmod_poly_expr)
```

63.19.22 Power series composition

```
Nmod_poly_expr compose_series(Nmod_poly_expr,
   Nmod_poly_expr, slong)

Nmod_poly_expr compose_series_horner(Nmod_poly_expr,
   Nmod_poly_expr, slong)

Nmod_poly_expr compose_series_brent_kung(Nmod_poly_expr,
   Nmod_poly_expr, slong)

Nmod_poly_expr compose_series_divconquer(Nmod_poly_expr,
   Nmod_poly_expr, slong)
```

63.19.23 Power series reversion

```
Nmod_poly_expr revert_series(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr revert_series_newton(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr revert_series_lagrange(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr revert_series_lagrange_fast(Nmod_poly_expr,
   T:fits_into_slong)
```

63.19.24 Square roots

```
Nmod_poly_expr sqrt_series(Nmod_poly_expr,
   T:fits_into_slong)
```

```
Nmod_poly_expr invsqrt_series(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr sqrt_series(Nmod_poly_expr p)
```

Compute the square root of p. Raises `flint_exception` if p is not a perfect square.

63.19.25 Transcendental functions

```
Nmod_poly_expr exp_series(Nmod_poly_expr, T:fits_into_slong)

Nmod_poly_expr log_series(Nmod_poly_expr, T:fits_into_slong)

Nmod_poly_expr atan_series(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr atanh_series(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr asin_series(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr asinh_series(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr tan_series(Nmod_poly_expr, T:fits_into_slong)

Nmod_poly_expr sin_series(Nmod_poly_expr, T:fits_into_slong)

Nmod_poly_expr cos_series(Nmod_poly_expr, T:fits_into_slong)

Nmod_poly_expr sinh_series(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr cosh_series(Nmod_poly_expr,
   T:fits_into_slong)

Nmod_poly_expr tanh_series(Nmod_poly_expr,
   T:fits_into_slong)
```

63.19.26 Products

```
static Nmod_poly_expr
   fmpz_polyxx::product_roots(Nmod_vec_expr xs)
```

63.19.27 Inflation and deflation

```
Nmod_poly_expr inflate(Nmod_poly_expr,
   T:is_unsigned_integer)

Nmod_poly_expr deflate(Nmod_poly_expr,
   T:is_unsigned_integer)

uslong Nmod_poly_expr::deflation() const
```

63.19.28 Factorisation

```
bool Nmod_poly_expr::is_squarefree() const
```

```
bool Nmod_poly_expr::is_irreducible() const
```

```
sslong Nmod_poly_target::remove(Nmod_poly_expr)
```

```
nmod_poly_factorxx::nmod_poly_factorxx()
```

Initialise an empty factorisation.

```
nmod_poly_factorxx::nmod_poly_factorxx(const
   nmod_poly_factorxx& o)
```

Copy a factorisation.

```
bool nmod_poly_factorxx::operator==(const
   nmod_poly_factorxx&)
```

Compare two factorisations.

```
ulong nmod_poly_factorxx::size() const
```

Return the number of stored factors.

```
slong nmod_poly_factorxx::exp(slong i) const
```

```
slong& nmod_poly_factorxx::exp(slong i)
```

Obtain the exponent of the ith factor.

```
nmod_polyxx_srcref nmod_poly_factorxx::p(slong i) const
```

```
nmod_polyxx_ref nmod_poly_factorxx::p(slong i)
```

Obtain the ith factor.

```
void nmod_poly_factorxx::realloc(slong a)
```

```
void nmod_poly_factorxx::fit_length(slong a)
```

```
void nmod_poly_factorxx::insert(Nmod_poly_expr p, slong e)
```

```
void nmod_poly_factorxx::concat(const nmod_poly_factorxx&)
```

```
void nmod_poly_factorxx::set_factor(Nmod_poly_expr)
```

```
void
   nmod_poly_factorxx::set_factor_cantor_zassenhaus(Nmod_poly_expr)
```

```
void
   nmod_poly_factorxx::set_factor_berlekamp(Nmod_poly_expr)
```

```
void
   nmod_poly_factorxx::set_factor_kaltofen_shoup(Nmod_poly_expr)
```

```
void
   nmod_poly_factorxx::set_factor_with_cantor_zassenhaus(Nmod_poly_expr)
```

```
void
   nmod_poly_factorxx::set_factor_with_berlekamp(Nmod_poly_expr)
```

```
void
   nmod_poly_factorxx::set_factor_with_kaltofen_shoup(Nmod_poly_expr)

void
   nmod_poly_factorxx::set_factor_squarefree(Nmod_poly_expr)
```

Factorise a polynomial and store its factors. See `nmod_poly_factor` etc.

```
void
   nmod_poly_factorxx::set_factor_equal_deg_probab(frandxx&,
   Nmod_poly_expr, slong)

void
   nmod_poly_factorxx::set_factor_equal_deg(Nmod_poly_expr,
   slong)
```

See `nmod_poly_factor_equal_deg_prob` and `nmod_poly_factor_equal_deg`.

```
void
   nmod_poly_factorxx::set_factor_distinct_deg(Nmod_poly_expr
   p, std::vector<slong>& degs)
```

See `nmod_poly_factor_distinct_deg`. Note that `degs` must have sufficient size to hold all factors. The size of `degs` is not modified.

```
nmod_poly_factorxx factor(Nmod_poly_expr)

nmod_poly_factorxx factor_cantor_zassenhaus(Nmod_poly_expr)

nmod_poly_factorxx factor_berlekamp(Nmod_poly_expr)

nmod_poly_factorxx factor_kaltofen_shoup(Nmod_poly_expr)

nmod_poly_factorxx
   factor_with_cantor_zassenhaus(Nmod_poly_expr)

nmod_poly_factorxx factor_with_berlekamp(Nmod_poly_expr)

nmod_poly_factorxx
   factor_with_kaltofen_shoup(Nmod_poly_expr)

nmod_poly_factorxx factor_squarefree(Nmod_poly_expr)
```

63.20 nmod_matxx

The class `nmod_matxx` wraps `nmod_mat_t`. Like `nmodxx`, instances of `nmod_matxx` always have an associated `nmodxx_ctx` storing the operating modulus. No expression may involve more than one modulus at a time.

Like `fmpz_matxx`, many operations on `nmod_matxx` do not support aliasing. The details can be found in the documentation of `nmod_mat_t`. Since `nmod_matxx` does not use temporary merging, evaluation of subexpressions never creates new aliases.

```
nmodxx_ctx_srcref Nmod_mat_expr::estimate_ctx() const
```

Obtain the relevant context. This never causes evaluation.

```
Nmod_mat_expr::unary operation() const
```

The following unary functions are made available as member functions: `inv`, `transpose`, `trace`, `det`.

```
Nmod_mat_expr::binary operation(??) const
```

The following binary functions are made available as member functions: `solve`, `mul_classical`, `mul_strassen`.

```
Nmod_mat_expr::ternary operation(??, ??) const
```

The following ternary functions are made available as member functions: `solve_tril`, `solve_tril_recursive`, `solve_tril_classical`, `solve_triu`, `solve_triu_recursive`, `solve_triu_classical`.

```
Nmod_mat_expr operator?(??, ??)
```

Arithmetic operators `+ - *` are overloaded when provided by `nmod_mat_t`.

```
Nmod_mat_expr operator-(Nmod_mat_expr)
```

The unary negation operator is overloaded.

63.20.1 Conversion

```
static nmod_matxx::reduce(Fmpz_mat_expr, mp_limb_t modulus)
```

See `fmpz_mat_get_nmod_mat`.

63.20.2 Input and output

```
int print(Nmod_mat_expr)
```

63.20.3 Memory management

```
nmod_matxx::nmod_matxx(slong m, slong n, mp_limb_t modulus)
```

See `nmod_mat_init`.

63.20.4 Basic properties and manipulation

```
?? Nmod_mat_expr::at(T:fits_into_slong, U:fits_into_slong)
   const
```

Unified coefficient access to the matrix entries.

```
sslong Nmod_mat_expr::rows() const

sslong Nmod_mat_expr::cols() const
```

Obtain the number of rows/columns in this matrix. These functions never cause evaluation (the matrix size is computed from the operations in the expression template and the size of the input matrices).

```
bool Nmod_mat_expr::is_zero() const

bool Nmod_mat_expr::is_empty() const

bool Nmod_mat_expr::is_square() const
```

```
mp_limb_t Nmod_mat_expr::modulus() const
```

```
void Nmod_mat_target::set_zero()
```

```
static nmod_matxx nmod_matxx::zero(slong rows, slong cols,
    mp_limb_t modulus)
```

See `nmod_mat_zero`.

63.20.5 Random matrix generation

```
void Nmod_mat_target::set_randtest(frandxx&)
```

```
void Nmod_mat_target::set_randfull(frandxx&)
```

```
void Nmod_mat_target::set_randrank(frandxx&, slong rank)
```

```
void Nmod_mat_target::set_randtril(frandxx&, bool unit)
```

```
void Nmod_mat_target::set_randtriu(frandxx&, bool unit)
```

See `nmod_mat_randtest` etc.

```
static nmod_matxx nmod_matxx::randtest(slong rows, slong
    cols, mp_limb_t M, frandxx&)
```

```
static nmod_matxx nmod_matxx::randfull(slong rows, slong
    cols, mp_limb_t M, frandxx&)
```

```
static nmod_matxx nmod_matxx::randrank(slong rows, slong
    cols, mp_limb_t M, frandxx&, slong rank)
```

```
static nmod_matxx nmod_matxx::randtril(slong rows, slong
    cols, mp_limb_t M, frandxx&, bool unit)
```

```
static nmod_matxx nmod_matxx::randtriu(slong rows, slong
    cols, mp_limb_t M, frandxx&, bool unit)
```

Static versions of the above.

```
int Nmod_mat_target::set_randpermdiag(frandxx&, const Vec&
    v)
```

`M.set_randpermdiag(Rand, V)` has the same effect as `nmod_mat_randpermdiag(m, rand, V._array(), V.size())`, where `m` and `rand` are the underlying C structs corresponding to `M` and `Rand`.

One possibility for Vec is `nmod_vecxx`.

```
void Nmod_target::apply_randops(frandxx&, slong count)
```

See `nmod_mat_randops`.

63.20.6 Transpose

```
Nmod_mat_expr transpose(Nmod_mat_expr)
```

63.20.7 Matrix multiplication

The overloaded `operator*` can be used for both matrix-matrix and matrix-scalar multiplication. Finer control can be obtained with the following functions.

```
Nmod_mat_expr mul_classical(Nmod_mat_expr, Nmod_mat_expr)
```

```
Nmod_mat_expr mul_strassen(Nmod_mat_expr, Nmod_mat_expr)
```

63.20.8 Trace

```
Nmod_expr trace(Nmod_mat_expr)
```

63.20.9 Determinant and rank

```
Nmod_expr det(Nmod_mat_expr)
```

```
slong rank(Nmod_mat_expr)
```

63.20.10 Inverse

```
Nmod_mat_expr inv(Nmod_mat_expr A)
```

Compute the inverse of the square matrix A. Raises `flint_exception` if A is singular. The modulus is required to be prime.

63.20.11 Triangular solving

```
Nmod_mat_expr solve_triu(Nmod_mat_expr, Nmod_mat_expr, bool
    unit)

Nmod_mat_expr solve_triu_classical(Nmod_mat_expr,
    Nmod_mat_expr, bool unit)

Nmod_mat_expr solve_triu_recursive(Nmod_mat_expr,
    Nmod_mat_expr, bool unit)

Nmod_mat_expr solve_tril(Nmod_mat_expr, Nmod_mat_expr, bool
    unit)

Nmod_mat_expr solve_tril_classical(Nmod_mat_expr,
    Nmod_mat_expr, bool unit)

Nmod_mat_expr solve_tril_recursive(Nmod_mat_expr,
    Nmod_mat_expr, bool unit)
```

63.20.12 Non-singular square solving

```
Nmod_mat_expr solve(Nmod_mat_expr B, Nmod_mat_expr X)

Nmod_vec_expr solve(Nmod_mat_expr B, Nmod_vec_expr X)
```

See `nmod_mat_solve` and `nmod_mat_solve_vec`. Raises `flint_exception` if B is singular.

63.20.13 LU decomposition

```
Tuple<slong, permxx> Nmod_mat_target::set_lu(bool
    rank_check = false)

Tuple<slong, permxx> Nmod_mat_target::set_lu_classical(bool
    rank_check = false)

Tuple<slong, permxx> Nmod_mat_target::set_lu_recursive(bool
    rank_check = false)
```

See `nmod_mat_lu` etc.

63.20.14 Reduced row echelon form

```
void Nmod_mat_target::set_rref()
```

63.20.15 Nullspace

```
Ltuple<slong, nmod_matxx>_expr nullspace(Nmod_mat_expr)
```

63.21 nmod_poly_matxx

The class `nmod_poly_matxx` wraps `nmod_poly_mat_t`. Like `nmod_matxx`, instances of `nmod_poly_matxx` always have an associated `nmodxx_ctx` storing the operating modulus. No expression may involve more than one modulus at a time.

Contrary to `nmod_poly_mat_t`, it is *not* valid to use instances of `nmod_poly_matxx` with zero rows or columns.

Like `fmpz_matxx`, many operations on `nmod_poly_matxx` do not support aliasing. The details can be found in the documentation of `nmod_poly_mat_t`. Since `nmod_poly_matxx` does not use temporary merging, evaluation of subexpressions never creates new aliases.

```
nmodxx_ctx_srcref Nmod_poly_mat_expr::estimate_ctx() const
```

Obtain the relevant context. This never causes evaluation.

```
Nmod_poly_mat_expr::unary operation() const
```

The following unary functions are made available as member functions: `det`, `det_fflu`, `det_interpolate`, `trace`, `sqr`, `sqr_classical`, `sqr_interpolate`, `sqr_KS`, `transpose`.

```
Nmod_poly_mat_expr::binary operation(??) const
```

The following binary functions are made available as member functions: `solve`, `solve_fflu`, `mul_classical`, `mul_interpolate`, `mul_KS`, `pow`.

```
Nmod_mat_expr Nmod_poly_mat_expr::operator()(Nmod_expr)
    const
```

`operator()` is overloaded for matrix evaluation.

```
Nmod_poly_mat_expr operator?(??, ??)
```

Arithmetic operators `+ - *` are overloaded when provided by `nmod_poly_mat_t`.

```
Nmod_poly_mat_expr operator-(Nmod_poly_mat_expr)
```

The unary negation operator is overloaded.

63.21.1 Input and output

```
int print_pretty(Nmod_poly_mat_expr, const char*)
```

63.21.2 Memory management

```
nmod_poly_matxx::nmod_poly_matxx(slong m, slong n,
    mp_limb_t modulus)
```

See nmod_poly_mat_init.

63.21.3 Basic assignment and manipulation

```
?? Nmod_poly_mat_expr::at(T:fits_into_slong,
   U:fits_into_slong) const
```

Unified coefficient access to the matrix entries.

63.21.4 Standard matrices

```
static nmod_poly_matxx nmod_poly_matxx::zero(slong rows,
   slong cols, mp_limb_t n)

static nmod_poly_matxx nmod_poly_matxx::one(slong rows,
   slong cols, mp_limb_t n)

void Nmod_poly_mat_target::set_zero()

void Nmod_poly_mat_target::set_one()
```

63.21.5 Random matrix generation

```
void Nmod_poly_mat_target::set_randtest(frandxx&, slong)

void Nmod_poly_mat_target::set_randtest_sparse(frandxx&,
   slong, float)

static nmod_poly_matxx nmod_poly_matxx::randtest(slong
   rows, slong cols, mp_limb_t n, slong len)

static nmod_poly_matxx
   nmod_poly_matxx::randtest_sparse(slong rows, slong cols,
   mp_limb_t n, slong len, float density)
```

See nmod_poly_mat_randtest etc.

63.21.6 Basic comparison and properties

```
sslong Nmod_poly_mat_expr::rows() const

sslong Nmod_poly_mat_expr::cols() const
```

Obtain the number of rows/columns in this matrix. These functions never cause evaluation (the matrix size is computed from the operations in the expression template and the size of the input matrices).

```
bool Nmod_poly_mat_expr::is_zero() const

bool Nmod_poly_mat_expr::is_one() const

bool Nmod_poly_mat_expr::is_empty() const

bool Nmod_poly_mat_expr::is_square() const

mp_limb_t Nmod_poly_mat_expr::modulus() const
```

63.21.7 Norms

```
sslong Nmod_poly_mat_expr::max_length() const
```

63.21.8 Arithmetic

The overloaded operators `+ - *` can be used for both matrix-matrix and matrix-scalar multiplication, and matrix-matrix addition/subtraction. Finer control can be obtained with the following functions.

```
Nmod_poly_mat_expr mul_classical(Nmod_poly_mat_expr,
   Nmod_poly_mat_expr)

Nmod_poly_mat_expr mul_interpolate(Nmod_poly_mat_expr,
   Nmod_poly_mat_expr)
```

63.21.9 Row reduction

Beware that compared to the C interface, the flintxx row reduction interface changes some argument orders. This is to facilitate default arguments.

```
slong find_pivot_any(Nmod_poly_mat_expr, slong, slong,
   slong)
```

See `nmod_poly_mat_find_pivot_any`.

```
slong find_pivot_partial(Nmod_poly_mat_expr, slong, slong,
   slong)
```

See `nmod_poly_mat_find_pivot_partial`.

```
Ltuple<slong, nmod_poly_matxx, fmpzxx>_expr
   fflu(Nmod_poly_mat_expr A, permxx* perm = 0, bool
   rankcheck = false)
```

See `nmod_poly_mat_fflu`.

```
Ltuple<slong, nmod_poly_matxx, fmpzxx>_expr
   rref(Nmod_poly_mat_expr A)
```

See `nmod_poly_mat_rref`.

63.21.10 Transpose

```
Nmod_poly_mat_expr transpose(Nmod_poly_mat_expr A)
```

Compute the transpose of A.

63.21.11 Trace

```
Nmod_poly_expr trace(Nmod_poly_mat_expr)
```

63.21.12 Determinant and rank

```
Nmod_poly_expr det(Nmod_poly_mat_expr)
```

```
Nmod_poly_expr det_fflu(Nmod_poly_mat_expr)

Nmod_poly_expr det_interpolate(Nmod_poly_mat_expr)

slong rank(Nmod_poly_mat_expr)
```

63.21.13 Inverse

```
Ltuple<bool, nmod_poly_matxx, nmod_polyxx>_expr
    inv(Nmod_poly_mat_expr A)
```

`ltupleref(worked, M, P)= inv(A)` has the same effect as `worked = nmod_poly_mat_inv(m, p, a)`, where `m, p, a` are the C structs underlying `M, P, A`.

63.21.14 Nullspace

```
Ltuple<slong, nmod_poly_matxx>_expr
    nullspace(Nmod_poly_mat_expr)
```

63.21.15 Solving

```
Ltuple<bool, nmod_poly_matxx, nmod_polyxx>_expr
    solve(Nmod_poly_mat_expr, Nmod_poly_mat_expr)

Ltuple<bool, nmod_poly_matxx, nmod_polyxx>_expr
    solve_fflu(Nmod_poly_mat_expr, Nmod_poly_mat_expr)

Ltuple<bool, nmod_poly_matxx, nmod_polyxx>_expr
    solve_fflu_precomp( const permxx&, Nmod_poly_mat_expr B,
    Nmod_poly_mat_expr FFLU, Nmod_poly_mat_expr X)
```

`ltupleref(worked, M, P)= solve(A, X)` has the same effect as `worked = nmod_poly_mat_solve(m, p, a, x)`, where `m, p, a, x` are the C structs underlying `M, P, A, X`.

63.22 fmpz_mod_polyxx

```
Fmpz_mod_poly_expr::unary operation() const
```

The following unary functions are made available as member functions: `derivative`, `integral`, `make_monic`, `sqr`.

```
Fmpz_mod_poly_expr::binary operation() const
```

The following binary functions are made available as member functions:
`compose_divconquer`, `compose_horner`, `div_basecase`,
`div_divconquer`, `div_newton`, `divrem`,
`divrem_basecase`, `divrem_divconquer`,
`divrem`, `divrem_f`,
`gcd`, `gcd_euclidean`, `gcd_euclidean_f`, `gcd_f`,
`gcdinv`, `invmod`, `inv_series_newton`,
`shift_left`, `shift_right`, `pow`,
`rem_basecase`, `xgcd`, `xgcd_euclidean`.

```
Fmpz_mod_poly_expr::ternary operation(??, ??) const
```

The following ternary functions are made available as member functions:
`compose_mod`, `compose_mod_horner`,
`compose_series_brent_kung`, `mullow`,
`mulmod`, `powmod_binexp`, `pow_trunc`,
`pow_trunc_binexp`.

```
Fmpz_mod_poly_expr
   Fmpz_mod_poly_expr::operator()(Fmpz_mod_poly_expr) const

Fmpz_mod_expr Fmpz_mod_poly_expr::operator()(Fmpz_mod_expr)
   const
```

The `operator()` is overloaded for evaluation or composition, depending on the argument.

```
Fmpz_mod_poly_expr operator?(??, ??)
```

Arithmetic operators `+ - * %` are overloaded when provided by `nmod_poly_t`.

```
Fmpz_mod_poly_expr operator-(Fmpz_mod_poly_expr)
```

The unary negation operator is overloaded.

63.22.1 Input and output

```
print(Fmpz_mod_poly_expr)

print(FILE*, Fmpz_mod_poly_expr)

print_pretty(Fmpz_mod_poly_expr, const char* var)

print_pretty(FILE*, Fmpz_mod_poly_expr, const char* var)

read(Fmpz_mod_poly_target)

read(FILE*, Fmpz_mod_poly_target)
```

63.22.2 Memory management

```
fmpz_mod_polyxx::fmpz_mod_polyxx(Fmpz_expr n)

fmpz_mod_polyxx::fmpz_mod_polyxx(Fmpz_expr n, slong alloc)

void Fmpz_mod_poly_target realloc(slong alloc)

void Fmpz_mod_poly_target::fit_length(slong len)

void Fmpz_mod_poly_target::_normalise()

void Fmpz_mod_poly_target::truncate(slong)
```

63.22.3 Randomisation

```
void Fmpz_mod_poly_mat_target::set_randtest(frandxx&, slong)

void
   Fmpz_mod_poly_mat_target::set_randtest_irreducible(frandxx&,
   slong)

void
   Fmpz_mod_poly_mat_target::set_randtest_not_zero(frandxx&,
   slong)
```

See `fmpz_mod_poly_randtest`, etc.

```
static fmpz_mod_polyxx fmpz_mod_polyxx::randtest(Fmpz_expr,
   frandx&, slong)

static fmpz_mod_polyxx
   fmpz_mod_polyxx::randtest_not_zero(Fmpz_expr, frandx&,
   slong)

static fmpz_mod_polyxx
   fmpz_mod_polyxx::randtest_irreducible(Fmpz_expr,
   frandx&, slong)
```

Static versions of the above.

63.22.4 Attributes

```
fmpzxx_srcref Fmpz_mod_poly_mat_expr::modulus() const
```

Obtain the relevant modulus. This never causes evaluation.

```
slong Fmpz_mod_poly_expr::length() const

slong Fmpz_mod_poly_expr::degree() const

?? Fmpz_mod_poly_expr::lead() const
```

Unified coefficient access for the leading coefficient. The result is undefined if the length of the polynomial is zero.

63.22.5 Assignment and swap

```
void Fmpz_mod_poly_target::zero_coeffs(slong i, slong j)

void Fmpz_mod_poly_target::set_zero()

static fmpz_mod_polyxx fmpz_mod_polyxx::zero(Fmpz_expr m)
```

63.22.6 Conversion

```
Fmpz_mod_poly_target
   Fmpz_mod_poly_target::operator=(T:is_unsigned_integer)

Fmpz_mod_poly_target
   Fmpz_mod_poly_target::operator=(Fmpz_expr)

Fmpz_mod_poly_target
   Fmpz_mod_poly_target::operator=(Fmpz_poly_expr)
```

See `fmpz_mod_poly_set_ui`, `fmpz_mod_poly_set_fmpz` and `fmpz_mod_poly_set_fmpz_poly`.

```
Fmpz_poly_expr Fmpz_mod_poly_expr::to<fmpz_polyxx>() const
```

63.22.7 Comparison

```
bool Fmpz_mod_poly_expr::is_zero() const
```

63.22.8 Getting and setting coefficients

```
Fmpz_expr Fmpz_mod_poly_expr::get_coeff(slong n) const
```

```
void Fmpz_mod_target::set_coeff(slong i, Fmpz_expr)

void Fmpz_mod_target::set_coeff(slong i,
   T:is_unsigned_integer)
```

63.22.9 Shifting

```
Fmpz_mod_poly_expr shift_left(Fmpz_mod_poly_expr,
   T:fits_into_slong)

Fmpz_mod_poly_expr shift_right(Fmpz_mod_poly_expr,
   T:fits_into_slong)
```

63.22.10 Multiplication

The overloaded `operator*` can be used for both poly-poly and poly-scalar multiplication. Finer control can be obtained using the following functions.

```
Fmpz_mod_poly_expr mullow(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr, slong)

Fmpz_mod_poly_expr sqr(Fmpz_mod_poly_expr)

Fmpz_mod_poly_expr mulmod(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr, Fmpz_mod_poly_expr)
```

63.22.11 Powering

```
Fmpz_mod_poly_expr pow(Fmpz_mod_poly_expr,
   T:is_unsigned_integer)

Fmpz_mod_poly_expr pow_binexp(Fmpz_mod_poly_expr,
   T:is_unsigned_integer)

Fmpz_mod_poly_expr pow_trunc(Fmpz_mod_poly_expr,
   T:is_unsigned_integer, T:fits_into_slong)

Fmpz_mod_poly_expr pow_trunc_binexp(Fmpz_mod_poly_expr,
   T:is_unsigned_integer, T:fits_into_slong)

Fmpz_mod_poly_expr powmod_binexp(Fmpz_mod_poly_expr,
   T:is_unsigned_integer, Fmpz_mod_poly_expr)

Fmpz_mod_poly_expr powmod_binexp(Fmpz_mod_poly_expr,
   Fmpz_expr, Fmpz_mod_poly_expr)
```

63.22.12 Division

The overloaded operators `/ %` can be used for division and remainder. Finer control can be obtained using the following functions.

```
Ltuple<fmpz_mod_polyxx, fmpz_mod_polyxx>_expr divrem(
   Fmpz_mod_poly_expr A, Fmpz_mod_poly_expr B)
```

```
Ltuple<fmpz_mod_polyxx, fmpz_mod_polyxx>_expr
   divrem_basecase( Fmpz_mod_poly_expr A,
   Fmpz_mod_poly_expr B)

Ltuple<fmpz_mod_polyxx, fmpz_mod_polyxx>_expr
   divrem_divconquer( Fmpz_mod_poly_expr A,
   Fmpz_mod_poly_expr B)

Ltuple<fmpzxx, fmpz_mod_polyxx, fmpz_mod_polyxx>_expr
   divrem_f( Fmpz_mod_poly_expr A, Fmpz_mod_poly_expr B)

Fmpz_mod_poly_expr div_basecase(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr)

Fmpz_mod_poly_expr rem_basecase(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr)

Fmpz_mod_poly_expr rem(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr)

slong Fmpz_mod_poly_target::remove(Fmpz_mod_poly_expr)
```

63.22.13 Power series inversion

```
Fmpz_mod_poly_expr inv_series_newton(Fmpz_mod_poly_expr,
   T:fits_into_slong)
```

63.22.14 Greatest common divisor

```
Fmpz_mod_poly_expr gcd(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr)

Fmpz_mod_poly_expr gcd_euclidean(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr)

Ltuple<fmpz_mod_polyxx, fmpz_mod_polyxx,
   fmpz_mod_polyxx>_expr xgcd( Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr)

Ltuple<fmpz_mod_polyxx, fmpz_mod_polyxx,
   fmpz_mod_polyxx>_expr xgcd_euclidean(
   Fmpz_mod_poly_expr, Fmpz_mod_poly_expr)

Ltuple<fmpz_mod_polyxx, fmpz_mod_polyxx,
   fmpz_mod_polyxx>_expr gcdinv( Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr)

Ltuple<fmpzxx, fmpz_mod_polyxx>_expr
   gcd_f(Fmpz_mod_poly_expr, Fmpz_mod_poly_expr)

Ltuple<fmpzxx, fmpz_mod_polyxx>_expr
   gcd_euclidean_f(Fmpz_mod_poly_expr, Fmpz_mod_poly_expr)

Fmpz_mod_poly_expr invmod(Fmpz_mod_poly_expr f,
   Fmpz_mod_poly_expr g)
```

See `fmpz_mod_poly_invmod`. Raises `flint_exception` if f and g are not coprime.

63.22.15 Derivative

```
Fmpz_mod_poly_expr derivative(Fmpz_mod_poly_expr)
```

63.22.16 Evaluation

```
Fmpz_mod_expr evaluate(Fmpz_mod_poly_expr, Fmpz_mod_expr)
```

63.22.17 Composition

Basic composition can be achieved with the overloaded `operator()`. Finer control can be obtained using the following functions.

```
Fmpz_mod_poly_expr compose(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr)

Fmpz_mod_poly_expr compose_horner(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr)

Fmpz_mod_poly_expr compose_divconquer(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr)
```

63.22.18 Modular composition

```
Fmpz_mod_poly_expr compose_mod(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr, Fmpz_mod_poly_expr)

Fmpz_mod_poly_expr compose_mod_horner(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr, Fmpz_mod_poly_expr)

Fmpz_mod_poly_expr
   compose_mod_divconquer(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr, Fmpz_mod_poly_expr)

Fmpz_mod_poly_expr
   compose_mod_brent_kung(Fmpz_mod_poly_expr,
   Fmpz_mod_poly_expr, Fmpz_mod_poly_expr)
```

63.22.19 Radix conversion

```
fmpz_mod_poly_radixxx::fmpz_mod_poly_radixxx(Fmpz_poly_expr,
   slong deg)
```

Initialise temporary data for radix conversion. See `fmpz_mod_poly_radix_init`.

```
Fmpz_mod_poly_vec_expr Fmpz_mod_poly_expr::radix(const
   fmpz_mod_poly_radxxx&)

Fmpz_mod_poly_vec_expr radix(Fmpz_mod_poly_expr F, const
   fmpz_mod_poly_radxxx&)
```

Perform radix conversion. See `fmpz_mod_poly_radix`. Note that computing the output vector size requires knowing the degree of `F`. In the current implementation, this will result in evaluating `F` twice. In order to avoid this, pass in `F` in evaluated form, or do not form expressions requiring temporaries.

63.23 fmpz_mod_poly_factorxx

```
bool Fmpz_mod_poly_expr::is_squarefree() const
```

```
bool Fmpz_mod_poly_expr::is_irreducible() const
```

```
bool Fmpz_mod_poly_expr::is_irreducible_ddf() const
```

```
bool Fmpz_mod_poly_expr::is_irreducible_rabin() const
```

```
slong Fmpz_mod_poly_target::remove(Fmpz_mod_poly_expr)
```

```
fmpz_mod_poly_factorxx::nmod_poly_factorxx()
```

Initialise an empty factorisation.

```
fmpz_mod_poly_factorxx::nmod_poly_factorxx(const
   nmod_poly_factorxx& o)
```

Copy a factorisation.

```
bool fmpz_mod_poly_factorxx::operator==(const
   nmod_poly_factorxx&)
```

Compare two factorisations.

```
ulong fmpz_mod_poly_factorxx::size() const
```

Return the number of stored factors.

```
slong fmpz_mod_poly_factorxx::exp(slong i) const
```

```
slong& fmpz_mod_poly_factorxx::exp(slong i)
```

Obtain the exponent of the ith factor.

```
fmpz_mod_polyxx_srcref nmod_poly_factorxx::p(slong i) const
```

```
fmpz_mod_polyxx_ref nmod_poly_factorxx::p(slong i)
```

Obtain the ith factor.

```
void fmpz_mod_poly_factorxx::realloc(slong a)
```

```
void fmpz_mod_poly_factorxx::fit_length(slong a)
```

```
void fmpz_mod_poly_factorxx::insert(Fmpz_mod_poly_expr p,
   slong e)
```

```
void fmpz_mod_poly_factorxx::concat(const
   nmod_poly_factorxx&)
```

```
void fmpz_mod_poly_factorxx::set_factor(Fmpz_mod_poly_expr)
```

```
void
   fmpz_mod_poly_factorxx::set_factor_cantor_zassenhaus(Fmpz_mod_poly_expr)

void
   fmpz_mod_poly_factorxx::set_factor_berlekamp(Fmpz_mod_poly_expr)

void
   fmpz_mod_poly_factorxx::set_factor_kaltofen_shoup(Fmpz_mod_poly_expr)

void
   fmpz_mod_poly_factorxx::set_factor_squarefree(Fmpz_mod_poly_expr)
```

Factorise a polynomial and store its factors. See `fmpz_mod_poly_factor` etc.

```
void
   fmpz_mod_poly_factorxx::set_factor_equal_deg_probab(frandxx&,
   Fmpz_mod_poly_expr, slong)

void
   fmpz_mod_poly_factorxx::set_factor_equal_deg(Fmpz_mod_poly_expr,
   slong)
```

See `fmpz_mod_poly_factor_equal_deg_prob` and `fmpz_mod_poly_factor_equal_deg`.

```
void
   fmpz_mod_poly_factorxx::set_factor_distinct_deg(Fmpz_mod_poly_expr
   p, std::vector<slong>& degs)
```

See `fmpz_mod_poly_factor_distinct_deg`. Note that `degs` must have sufficient size to hold all factors. The size of `degs` is not modified.

```
fmpz_mod_poly_factorxx factor(Fmpz_mod_poly_expr)

fmpz_mod_poly_factorxx
   factor_cantor_zassenhaus(Fmpz_mod_poly_expr)

fmpz_mod_poly_factorxx factor_berlekamp(Fmpz_mod_poly_expr)

fmpz_mod_poly_factorxx
   factor_kaltofen_shoup(Fmpz_mod_poly_expr)

fmpz_mod_poly_factorxx factor_squarefree(Fmpz_mod_poly_expr)
```

63.24 padicxx

The type `padicxx` wraps the C interface `padic_t`, and the type `padicxx_ctx` wraps `padic_ctx_t`.

Evaluating composite expressions requires temporary objects, which must be initialised to a certain precision and with a certain context. The padicxx library employs the following rules:

- In any compound expression, there must only be one context involved.
- Temporary objects are initialised to the maximum precision of any subexpression.

In most use cases, all objects in a compound expression have the same precision, and so temporary expressions are evaluated to this precision. If you need temporary subexpressions to be evaluated to higher precision, the `toN` method can be used on immediates to increase their effective precision, thus (potentially) increasing the precision of intermediates.

63.24.1 Context

```
padicxx_ctx::padicxx_ctx(Fmpz_src p, slong min, slong max,
   padic_print_mode mode)
```

Initialize a padic context. See `padic_ctx_init`.

```
padic_ctx_t& padicxx_ctx::_ctx() const
```

Obtain a reference to the underlying C data structure. Note that this reference is mutable even if the instance of `padicxx_ctx` it is obtained from is not. This is because the context contains data which is not user-visible, and the C functions change them.

If this is called on a constant instance of `padicxx_ctx`, you must ensure that no user-visible state is changed.

```
padic_print_mode& padicxx_ctx::mode()
```

```
padic_print_mode padicxx_ctx::mode() const
```

63.24.2 C++ particulars

```
padicxx_ctx_srcref Padic_src::get_ctx() const
```

```
padic_ctx_t& Padic_src::_ctx() const
```

Obtain a reference to the context of this instance.

```
padicxx_ctx_srcref Padic_expr::estimate_ctx() const
```

Obtain a reference to a context occurring in a subexpression. As per the first rule in the introduction to this section, all such contexts are the same by definition.

```
Padic_expr::unary operation() const
```

The following unary functions are made available as member functions: `exp`, `exp_balanced`, `exp_rectangular`, `inv`, `log`, `log_balanced`, `log_satoh`, `sqrt`, `teichmuller`.

```
Padic_expr Padic_expr::pow(T:fits_into_slong) const
```

```
padicxx_srcref Padic_src::toN(sslong N) const
```

Obtain a new version of the operand, with changed effective precision.

63.24.3 Input and output

```
int print(Padic_expr)
```

```
int print(FILE*, Padic_expr)
```

63.24.4 Data structures

```
Fmpz_expr Padic_expr::unit() const
```

See `padic_unit`.

```
slong Padic_expr::val() const
```

```
slong& Padic_target::val()
```

```
slong Padic_expr::prec() const

slong& Padic_target::prec()
```

Obtain the precision of this instance. See `padic_prec`. Note that this never requires evaluation.

63.24.5 Memory management

```
padicxx::padicxx(padicxx_ctx_srcref)
```

Initialize padic number to default precision. See `padic_init`.

```
padicxx::padicxx(padicxx_ctx_srcref c, slong N)
```

Initialize padic number to precision N. See `padic_init2`.

```
void Padic_target::reduce()
```

See `padic_reduce`.

63.24.6 Randomisation

```
static padicxx padicxx::randtest(frandxx& state,
   padicxx_ctx_srcref ctx, slong prec = PADIC_DEFAULT_PREC)

static padicxx padicxx::randtest_int(frandxx& state,
   padicxx_ctx_srcref ctx, slong prec = PADIC_DEFAULT_PREC)

static padicxx padicxx::randtest_not_zero(frandxx& state,
   padicxx_ctx_srcref ctx, slong prec = PADIC_DEFAULT_PREC)
```

Obtain a random padic number of precision `prec`. See `padic_randtest`, `padic_randtest_int` and `padic_randtest_not_zero`.

63.24.7 Conversion

```
Padic_target Padic_target::operator=(T:is_integer)

Padic_target Padic_target::operator=(Fmpz_expr)

Padic_target Padic_target::operator=(Fmpq_expr)

padicxx padicxx::from_QQ(Fmpq_expr, padicxx_ctx_srcref)

padicxx padicxx::from_QQ(Fmpz_expr, padicxx_ctx_srcref)

padicxx padicxx::from_QQ(T:is_integer, padicxx_ctx_srcref)

padicxx padicxx::from_QQ(Fmpq_expr, padicxx_ctx_srcref,
   sslong N)

padicxx padicxx::from_QQ(Fmpz_expr, padicxx_ctx_srcref,
   sslong N)

padicxx padicxx::from_QQ(T:is_integer, padicxx_ctx_srcref,
   sslong N)

void Padic_target::set_zero()
```

```
void Padic_target::set_one()

padicxx padicxx::zero(padicxx_ctx_srcref)

padicxx padicxx::zero(padicxx_ctx_srcref, sslong N)

padicxx padicxx::one(padicxx_ctx_srcref)

padicxx padicxx::one(padicxx_ctx_srcref, sslong N)

bool Padic_expr::is_zero() const

bool Padic_expr::is_one() const

fmpzxx Padic_expr::to<fmpzxx>() const
```

Convert self to `fmpzxx`, if possible. See `padic_get_fmpz`.

```
fmpqxx Padic_expr::to<fmpqxx>() const
```

Convert self to `fmpqxx`. See `padic_get_fmpz`.

```
std::string Fmpz_expr::to_string() const
```

63.24.8 Arithmetic operations

The overloaded operators `+ - * / << >>` can be used for arithmetic operations, provided these are implemented for `padic_t`.

```
Padic_expr inv(Padic_expr)

Padic_expr sqrt(Padic_expr)
```

Compute square root. May raise `flint_exception` if no square root exists. See `padic_sqrt`.

```
Padic_expr pow(Padic_expr, T:fits_into_slong)
```

63.24.9 Exponential

```
Padic_expr exp(Padic_expr)

Padic_expr exp_rectangular(Padic_expr)

Padic_expr exp_balanced(Padic_expr)
```

Compute the exponential function. These may raise `flint_exceptions` if the series do not converge.

63.24.10 Logarithm

```
Padic_expr log(Padic_expr)

Padic_expr log_rectangular(Padic_expr)

Padic_expr log_balanced(Padic_expr)

Padic_expr log_satoh(Padic_expr)
```

Compute the logarithm function. These may raise `flint_exception`s if the series do not converge.

63.24.11 Special functions

```
Padic_expr teichmuller(Padic_expr)

Fmpz_expr padic_val_fac(Fmpz_expr, Fmpz_expr)

ulong padic_val_fac(T:is_unsigned_integer, Fmpz_expr)
```

63.25 padic_polyxx

The type `padic_polyxx` wraps `padic_poly`. Like `padicxx`, every instance of `padic_polyxx` contains a reference to a context `padicxx_ctx`, and stores its own precision. The same rules regarding temporary expressions apply to `padic_polyxx` as to `padicxx`.

63.25.1 C++ particulars

```
padicxx_ctx_srcref Padic_poly_src::get_ctx() const

padic_ctx_t& Padic_poly_src::_ctx() const
```

Obtain a reference to the context of this instance.

```
padicxx_ctx_srcref Padic_poly_expr::estimate_ctx() const
```

Obtain a reference to a context occurring in a subexpression.

```
Padic_poly_expr::unary operation() const
```

The following unary functions are made available as member functions: `derivative`.

```
Padic_poly_expr::binary operation() const
```

The following binary functions are made available as member functions: `pow`, `compose_pow`, `inv_series`, `shift_left`, `shift_right`.

```
padic_polyxx_srcref Padic_poly_src::toN(sslong N) const
```

Obtain a new version of the operand, with changed effective precision.

63.25.2 Input and output

```
int print(Padic_expr)

int print(FILE*, Padic_expr)

int print_pretty(Padic_expr, const char*)

int print_pretty(FILE*, Padic_expr, const char*)
```

63.25.3 Memory management

```
padic_polyxx::padic_polyxx(padicxx_ctx_srcref)
```

Initialise to zero. See `padic_poly_init`.

```
padic_polyxx::padic_polyxx(padicxx_ctx_srcref, slong prec,
   slong alloc = 0)
```

See padic_poly_init2.

```
void Padic_poly_target realloc(slong alloc)

void Padic_poly_target::fit_length(slong len)

void Padic_poly_target::canonicalise()

void Padic_poly_target::reduce()

void Padic_poly_target::truncate(slong)
```

63.25.4 Polynomial parameters

```
slong Padic_poly_expr::length() const

slong Padic_poly_expr::degree() const

slong Padic_expr::val() const

slong& Padic_target::val()

slong Padic_expr::prec() const

slong& Padic_target::prec()
```

63.25.5 Randomisation

```
static padic_polyxx padic_polyxx::randtest(frandxx& state,
   padicxx_ctx_srcref ctx, slong len, slong prec =
   PADIC_DEFAULT_PREC)

static padic_polyxx padic_polyxx::randtest_val(frandxx&
   state, padicxx_ctx_srcref ctx, slong len, slong val,
   slong prec = PADIC_DEFAULT_PREC)

static padic_polyxx
   padic_polyxx::randtest_not_zero(frandxx& state, slong
   len, padicxx_ctx_srcref ctx, slong prec =
   PADIC_DEFAULT_PREC)
```

63.25.6 Assignment an basic manipulation

The overloaded operator= can be used for assignments. Additionally, we provide the following functions.

```
padic_polyxx padic_polyxx::from_QQ(T:is_integer,
   padicxx_ctx_srcref, sslong N)

padic_polyxx padic_polyxx::from_QQ(Fmpz_expr,
   padicxx_ctx_srcref, sslong N)

padic_polyxx padic_polyxx::from_QQ(Fmpq_expr,
   padicxx_ctx_srcref, sslong N)
```

```
padic_polyxx padic_polyxx::from_QQ(T:is_integer,
   padicxx_ctx_srcref)

padic_polyxx padic_polyxx::from_QQ(Fmpz_expr,
   padicxx_ctx_srcref)

padic_polyxx padic_polyxx::from_QQ(Fmpq_expr,
   padicxx_ctx_srcref)

padic_polyxx padic_polyxx::from_QQX(Fmpz_poly_expr,
   padicxx_ctx_srcref, sslong N)

padic_polyxx padic_polyxx::from_QQX(Fmpq_poly_expr,
   padicxx_ctx_srcref, sslong N)

padic_polyxx padic_polyxx::from_QQX(Fmpz_poly_expr,
   padicxx_ctx_srcref)

padic_polyxx padic_polyxx::from_QQX(Fmpz_poly_expr,
   padicxx_ctx_srcref)

padic_polyxx padic_polyxx::from_ground(Padic_expr)

fmpz_polyxx Padic_poly_expr::to<fmpz_polyxx>() const
```

Convert to an integer polynomial. Raises `flint_exception` if the polynomial is not p-adically integral. See `padic_poly_get_fmpz_poly`.

```
fmpq_polyxx Padic_poly_expr::to<fmpq_polyxx>() const
```

See `padic_poly_get_fmpq_poly`.

```
padic_polyxx padic_polyxx::zero(const padic_polyxx_ctx&)

padic_polyxx padic_polyxx::zero(const padic_polyxx_ctx&,
   sslong N)

padic_polyxx padic_polyxx::one(const padic_polyxx_ctx&)

padic_polyxx padic_polyxx::one(const padic_polyxx_ctx&,
   sslong N)
```

63.25.7 Getting and setting coefficients

```
Padic_expr Padic_poly_expr::get_coeff(slong n) const

void Padic_poly_target::set_coeff(slong i, Padic_expr)
```

63.25.8 Comparison

The overloaded `operator==` can be used for comparison.

```
bool Padic_poly_expr::is_zero() const

bool Padic_poly_expr::is_one() const
```

63.25.9 Arithmetic

The overloaded operators `+ - *` can be used for arithmetic.

63.25.10 Powering

```
Padic_poly_expr pow(Padic_poly_expr , T:fits_into_slong)
```

63.25.11 Series inversion

```
Padic_poly_expr inv_series_newton(Padic_poly_expr ,
   T:fits_into_slong)
```

63.25.12 Derivative

```
Padic_poly_expr derivative(Padic_poly_expr)
```

63.25.13 Shifting

```
Padic_poly_expr shift_left(Padic_poly_expr ,
   T:fits_into_slong)

Padic_poly_expr shift_right(Padic_poly_expr ,
   T:fits_into_slong)
```

63.25.14 Evaluation and composition

The overloaded `operator()` can be used for both evaluation and composition.

```
Padic_expr evaluate(Padic_poly_expr , Padic_expr)

Padic_poly_expr compose(Padic_poly_expr , Padic_poly_expr)

Padic_poly_expr compose_pow(Padic_poly_expr ,
   T:fits_into_slong)
```

63.25.15 Testing

```
bool Padic_poly_src::is_canonical() const

bool Padic_poly_src::is_reduced() const
```

63.26 padic_matxx

The type `padic_matxx` wraps `padic_mat`. Like `padicxx`, every instance of `padic_matxx` contains a reference to a context `padicxx_ctx`, and stores its own precision. The same rules regarding temporary expressions apply to `padic_matxx` as to `padicxx`.

63.26.1 C++ particulars

```
padicxx_ctx_srcref Padic_mat_src::get_ctx() const
```

```
padic_ctx_t& Padic_mat_src::_ctx() const
```

Obtain a reference to the context of this instance.

```
padicxx_ctx_srcref Padic_mat_expr::estimate_ctx() const
```

Obtain a reference to a context occurring in a subexpression.

```
padic_matxx_srcref Padic_mat_src::toN(sslong N) const
```

Obtain a new version of the operand, with changed effective precision.

```
slong Padic_mat_expr::rows() const
```

```
slong Padic_mat_expr::cols() const
```

Obtain the number of rows/columns of this matrix. This never evaluates.

```
slong Padic_mat_expr::val() const
```

```
slong& Padic_mat_target::val()
```

```
Padic_mat_expr Padic_mat_expr::transpose() const
```

63.26.2 Input and output

```
int print(Padic_mat_expr)
```

```
int print(FILE*, Padic_mat_expr)
```

```
int print_pretty(Padic_mat_expr)
```

```
int print_pretty(FILE*, Padic_mat_expr)
```

63.26.3 Memory management

```
padic_matxx::padic_matxx(padicxx_ctx_srcref, slong rows,
   slong cols)
```

See padic_mat_init.

```
padic_matxx::padic_matxx(padicxx_ctx_srcref, slong rows,
   slong cols, slong prec)
```

See padic_mat_init2.

```
void Padic_mat_target::canonicalise()
```

```
void Padic_mat_target::reduce()
```

```
bool Padic_mat_src::is_canonical() const
```

```
bool Padic_mat_src::is_reduced() const
```

```
bool Padic_mat_src::is_square() const
```

```
bool Padic_mat_src::is_empty() const
```

63.26.4 Basic assignment

Overloaded `operatior=` can be used for assignment.

```
void Padic_mat_target::set_zero()

void Padic_mat_target::set_one()

padic_matxx padic_matxx::zero(padicxx_ctx_srcref)

padic_matxx padic_matxx::zero(padicxx_ctx_srcref, sslong N)

padic_matxx padic_matxx::one(padicxx_ctx_srcref)

padic_matxx padic_matxx::one(padicxx_ctx_srcref, sslong N)
```

63.26.5 Conversion

Converting from a `fmpq_matxx` can be done using `operator=`, or the following functions.

```
padic_matxx padic_matxx::from_QQ(Fmpq_mat_expr,
   padicxx_ctx_srcref)

fmpq_matxx Padic_mat_expr::to<fmpq_matxx>() const
```

63.26.6 Entries

```
?? Padic_mat_expr::at(slong i, slong j)
```

Unified coefficient access to the underlying integer matrix. See `padic_mat_entry`.

```
Fmpz_expr Padic_mat_expr::get_entry(slong i, slong j)

void Padic_mat_target::set_entry(slong i, slong j,
   Padic_expr)
```

63.26.7 Comparison

Overloaded `operator==` can be used for comparison.

```
bool Padic_mat_expr::is_zero() const
```

63.26.8 Random matrix generation

```
static padic_polyxx padic_polyxx::randtest(slong rows,
   slong cols, frandxx& state, padicxx_ctx_srcref ctx,
   slong prec = PADIC_DEFAULT_PREC)
```

63.26.9 Transpose

```
Padic_mat_expr transpose(Padic_mat_expr)
```

63.26.10 Arithmetic

Overloaded operators `+ - * /` can be used for arithmetic.

63.27 qadicxx

The type `qadicxx` wraps the C interface `qadic_t`, and the type `qadicxx_ctx` wraps `qadic_ctx_t`.

Evaluating composite expressions requires temporary objects, which must be initialised to a certain precision and with a certain context. The same rules apply as for `padicxx`.

63.27.1 Context

```
qadicxx_ctx::qadicxx_ctx(Fmpz_src p, sslong min, slong max,
   padic_print_mode mode, const char* var = "x")
```

Initialize a qadic context. See `qadic_ctx_init_conway`.

```
qadic_ctx_t& qadicxx_ctx::_ctx() const
```

Obtain a reference to the underlying C data structure. Note that this reference is mutable even if the instance of `qadicxx_ctx` it is obtained from is not. This is because the context contains data which is not user-visible, and the C functions change them.

If this is called on a constant instance of `qadicxx_ctx`, you must ensure that no user-visible state is changed.

```
padicxx_ctx_srcref qadicxx_ctx::pctx() const
```

Obtain a reference to the underlying padic context.

63.27.2 C++ particulars

```
padicxx_ctx_srcref Qadic_src::get_ctx() const
```

```
const qadicxx_ctx& Qadic_src::get_qctx() const
```

```
qadic_ctx_t& Qadic_src::_ctx() const
```

Obtain a reference to the context of this instance.

```
const qadicxx_ctx& Qadic_expr::estimate_ctx() const
```

Obtain a reference to a context occurring in a subexpression. As per the first rule in the introduction to this section, all such contexts are the same by definition.

```
Qadic_expr::unary operation() const
```

The following unary functions are made available as member functions: `exp`, `exp_balanced`, `exp_rectangular`, `inv`, `log`, `log_balanced`, `teichmuller`, `trace`, `norm`, `norm_analytic`, `norm_resultant`.

```
Qadic_expr Qadic_expr::pow(Fmpz_expr) const
```

```
Qadic_expr Qadic_expr::frobenius(T:fits_into_slong) const
```

```
qadicxx_srcref Qadic_src::toN(sslong N) const
```

Obtain a new version of the operand, with changed effective precision.

63.27.3 Data structures

```
int print_pretty(Qadic_expr)
```

```
int print_pretty(FILE*, Qadic_expr)
```

63.27.4 Data structures

```
slong Qadic_expr::val() const
```

```
slong Qadic_expr::prec() const
```

Obtain the precision of this instance. See `qadic_prec`. Note that this never requires evaluation.

63.27.5 Memory management

```
qadicxx::qadicxx(const qadicxx_ctx&)
```

Initialize qadic number to default precision. See `qadic_init`.

```
qadicxx::qadicxx(const qadicxx_ctx& c, slong N)
```

Initialize qadic number to precision N. See `qadic_init2`.

```
void Qadic_target::reduce()
```

See `qadic_reduce`.

63.27.6 Randomisation

```
static qadicxx qadicxx::randtest(frandxx& state, const
   qadicxx_ctx& ctx, slong prec = PADIC_DEFAULT_PREC)
```

```
static qadicxx qadicxx::randtest_int(frandxx& state, slong
   val, const qadicxx_ctx& ctx, slong prec =
   PADIC_DEFAULT_PREC)
```

```
static qadicxx qadicxx::randtest_val(frandxx& state, const
   qadicxx_ctx& ctx, slong prec = PADIC_DEFAULT_PREC)
```

```
static qadicxx qadicxx::randtest_not_zero(frandxx& state,
   const qadicxx_ctx& ctx, slong prec = PADIC_DEFAULT_PREC)
```

Obtain a random qadic number of precision `prec`. See `qadic_randtest`, `qadic_randtest_int` and `qadic_randtest_not_zero`.

63.27.7 Conversion

```
Qadic_target Qadic_target::operator=(T:is_unsigned_integer)
```

```
Qadic_target Qadic_target::operator=(Padic_expr)
```

```
qadicxx qadicxx::from_ground(Padic_expr, const qadicxx_ctx&)
```

```
void Qadic_target::set_zero()
```

```
void Qadic_target::set_one()
```

```
void Qadic_target::set_gen(const qadicxx_ctx&)
```

```
qadicxx qadicxx::zero(const qadicxx_ctx&)
```

```
qadicxx qadicxx::zero(const qadicxx_ctx&, sslong N)

qadicxx qadicxx::one(const qadicxx_ctx&)

qadicxx qadicxx::one(const qadicxx_ctx&, sslong N)

qadicxx qadicxx::gen(const qadicxx_ctx&)

qadicxx qadicxx::gen(const qadicxx_ctx&, sslong N)

bool Qadic_expr::is_zero() const

bool Qadic_expr::is_one() const

padicxx Qadic_expr::to<padicxx>() const
```

Convert self to `padicxx`, if possible. See `qadic_get_padic`.

63.27.8 Arithmetic operations

The overloaded operators `+ - * / << >>` can be used for arithmetic operations, provided these are implemented for `qadic_t`.

```
Qadic_expr inv(Qadic_expr)

Qadic_expr pow(Qadic_expr, Fmpz_expr)
```

63.27.9 Exponential

```
Qadic_expr exp(Qadic_expr)

Qadic_expr exp_rectangular(Qadic_expr)

Qadic_expr exp_balanced(Qadic_expr)
```

Compute the exponential function. These may raise `flint_exception`s if the series do not converge.

63.27.10 Logarithm

```
Qadic_expr log(Qadic_expr)

Qadic_expr log_balanced(Qadic_expr)
```

Compute the logarithm function. These may raise `flint_exception`s if the series do not converge.

63.27.11 Special functions

```
Qadic_expr teichmuller(Qadic_expr)

Padic_expr trace(Qadic_expr)

Padic_expr norm(Qadic_expr)

Padic_expr norm_analytic(Qadic_expr)
```

```
Padic_expr norm_resultant(Qadic_expr)
```

63.28 arithxx

The `arithxx` module wraps the `arith` module, i.e. provides functions for computing number theoretic functions.

63.28.1 Primorials

```
Fmpz_expr primorial(T:fits_into_slong)
```

63.28.2 Harmonic numbers

```
Fmpq_expr harmonic_number(T:fits_into_slong)
```

63.28.3 Stirling numbers

```
Fmpz_expr stirling_number_1u(T:fits_into_slong,
   T:fits_into_slong)

Fmpz_expr stirling_number_1(T:fits_into_slong,
   T:fits_into_slong)

Fmpz_expr stirling_number_2(T:fits_into_slong,
   T:fits_into_slong)

Fmpz_vec_expr stirling_number_1u_vec(T:fits_into_slong,
   T:fits_into_slong)

Fmpz_vec_expr stirling_number_1_vec(T:fits_into_slong,
   T:fits_into_slong)

Fmpz_vec_expr stirling_number_2_vec(T:fits_into_slong,
   T:fits_into_slong)

Fmpz_vec_expr stirling_number_1u_vec_next(Fmpz_vec_expr v,
   T:fits_into_slong n)

Fmpz_vec_expr stirling_number_1_vec_next(Fmpz_vec_expr v,
   T:fits_into_slong n)

Fmpz_vec_expr stirling_number_2_vec_next(Fmpz_vec_expr v,
   T:fits_into_slong n)
```

Given the vector v of length k, compute the next vector of Stirling numbers. The size of the new vector is $k+1$ if $k = n$, and else k.

See `arith_stirling_number_1u_vec_next` etc.

```
Fmpz_mat_expr stirling_matrix_1u(T:fits_into_slong m,
   T:fits_into_slong)

Fmpz_mat_expr stirling_matrix_1(T:fits_into_slong m,
   T:fits_into_slong)
```

```
Fmpz_mat_expr stirling_matrix_2(T:fits_into_slong m,
   T:fits_into_slong)
```

Compute an $m \times n$ Stirling matrix.

See `arith_stirling_matrix_1u` etc.

63.28.4 Bell numbers

```
Fmpz_expr bell_number(T:is_unsigned_integer)

Fmpz_expr bell_number_bsplit(T:is_unsigned_integer)

Fmpz_expr bell_number_multi_mod(T:is_unsigned_integer)

Fmpz_vec_expr bell_number_vec(T:is_unsigned_integer)

Fmpz_vec_expr
   bell_number_vec_recursive(T:is_unsigned_integer)

Fmpz_vec_expr
   bell_number_vec_multi_mod(T:is_unsigned_integer)

Nmod_expr bell_number_nmod(T:is_unsigned_integer,
   Nmodxx_ctx_src)

Nmod_vec_expr bell_number_nmod_vec(T:is_unsigned_integer,
   Nmodxx_ctx_src)

Nmod_vec_expr
   bell_number_nmod_vec_recursive(T:is_unsigned_integer,
   Nmodxx_ctx_src)

Nmod_vec_expr
   bell_number_nmod_vec_series(T:is_unsigned_integer,
   Nmodxx_ctx_src)

double bell_number_size(ulong n)
```

63.28.5 Bernoulli numbers and polynomials

```
Fmpq_expr bernoulli_number(T:is_unsigned_integer)

Fmpq_vec_expr bernoulli_number_vec(T:fits_into_slong)

Fmpz_expr bernoulli_number_denom(T:is_unsigned_integer)

double bernoulli_number_size(ulong)

Fmpq_poly_expr bernoulli_polynomial(T:is_unsigned_integer)
```

63.28.6 Euler numbers and polynomials

```
Fmpq_expr euler_number(T:is_unsigned_integer)

Fmpq_vec_expr euler_number_vec(T:fits_into_slong)

double euler_number_size(ulong)
```

```
Fmpq_poly_expr euler_polynomial(T:is_unsigned_integer)
```

63.28.7 Legendre polynomials

```
Fmpq_poly_expr legendre_polynomial(T:is_unsigned_integer)

Fmpz_poly_expr chebyshev_t_polynomial(T:is_unsigned_integer)

Fmpz_poly_expr chebyshev_u_polynomial(T:is_unsigned_integer)
```

63.28.8 Multiplicative functions

```
Fmpz_expr euler_phi(Fmpz_expr)

int moebius_mu(Fmpz_expr)

Fmpz_expr divisor_sigma(Fmpz_expr, ulong)

Fmpz_poly_expr divisors(Fmpz_expr)

Fmpz_expr ramanujan_tau(Fmpz_expr)

Fmpz_poly_expr ramanujan_tau_series(T:fits_into_slong)
```

63.28.9 Cyclotomic polynomials

```
Fmpz_poly_expr cyclotomic_polynomial(T:is_unsigned_integer)

Fmpz_poly_expr cos_minpoly(T:is_unsigned_integer)
```

63.28.10 Swinnerton-Dyer polynomials

```
Fmpz_poly_expr
   swinnerton_dyer_polynomial(T:is_unsigned_integer)
```

63.28.11 Landau's function

```
Fmpz_vec_expr landau_function_vec(T:is_unsigned_integer)
```

63.28.12 Dedekind sums

```
Fmpz_expr dedekind_sum_naive(Fmpz_expr, Fmpz_expr)

Fmpz_expr dedekind_sum_coprime_large(Fmpz_expr, Fmpz_expr)

Fmpz_expr dedekind_sum_coprime(Fmpz_expr, Fmpz_expr)

Fmpz_expr dedekind_sum(Fmpz_expr, Fmpz_expr)

double dedekind_sum_d(double, double)
```

63.28.13 Number of partitions

```
Fmpz_vec_expr number_of_partitions_vec(T:fits_into_slong)
```

```
Nmod_vec_expr
   number_of_partitions_nmod_vec(T:fits_into_slong)

Fmpz_expr number_of_partitions(T:is_unsigned_integer)
```

63.28.14 Sums of squares

```
Fmpz_expr sum_of_squares(T:is_unsigned_integer, Fmpz_expr)

Fmpz_vec_expr sum_of_squares(T:is_unsigned_integer,
   T:fits_into_slong)
```

§64. profiler

64.1 Timer based on the cycle counter

```
void timeit_start(timeit_t t)
```

```
void timeit_stop(timeit_t t)
```

Gives wall and user time - useful for parallel programming.

Example usage:

```
timeit_t t0;

// ...

timeit_start(t0);

// do stuff, take some time

timeit_stop(t0);

flint_printf("cpu = %wd ms  wall = %wd ms\n", t0->cpu,
   t0->wall);
```

```
void start_clock(int n)
```

```
void stop_clock(int n)
```

```
double get_clock(int n)
```

Gives time based on cycle counter.

First one must ensure the processor speed in cycles per second is set correctly in `profiler.h`, in the macro definition `#define FLINT_CLOCKSPEED`.

One can access the cycle counter directly by `get_cycle_counter()` which returns the current cycle counter as a `double`.

A sample usage of clocks is:

```
init_all_clocks();

start_clock(n);

// do something

stop_clock(n);

flint_printf("Time in seconds is %f.3\n", get_clock(n));
```

where `n` is a clock number (from 0-19 by default). The number of clocks can be changed by altering `FLINT_NUM_CLOCKS`. One can also initialise an individual clock with `init_clock(n)`.

64.2 Framework for repeatedly sampling a single target

```
void prof_repeat(double *min, double *max, profile_target_t
   target, ulong count)
```

Allows one to automatically time a given function. Here is a sample usage:

Suppose one has a function one wishes to profile:

```
void myfunc(ulong a, ulong b);
```

One creates a struct for passing arguments to our function:

```
typedef struct
{
    ulong a, b;
} myfunc_t;
```

a sample function:

```
void sample_myfunc(void * arg, ulong count)
{
    myfunc_t * params = (myfunc_t *) arg;

    ulong a = params->a;
    ulong b = params->b;

    for (ulong i = 0; i < count; i++)
    {
        prof_start();
        myfunc(a, b);
        prof_stop();
    }
}
```

Then we do the profile

```
double min, max;

myfunc_t params;

params.a = 3;
params.b = 4;

prof_repeat(&min, &max, sample_myfunc, &params);

flint_printf("Min time is %lf.3s, max time is
   %lf.3s\n", min, max);
```

If either of the first two parameters to `prof_repeat` are `NULL`, that value is not stored.

One may set the minimum time in microseconds for a timing run by adjusting `DURATION_THRESHOLD` and one may set a target duration in microseconds by adjusting `DURATION_TARGET` in `profiler.h`.

64.3 Memory usage

`void get_memory_usage(meminfo_t meminfo)`

Obtains information about the memory usage of the current process. The meminfo object contains the slots `size` (virtual memory size), `peak` (peak virtual memory size), `rss` (resident set size), `hwm` (peak resident set size). The values are stored in kilobytes (1024 bytes). This function currently only works on Linux.

64.4 Simple profiling macros

`macro TIMEIT_REPEAT(timer, reps)`

`macro TIMEIT_END_REPEAT(timer, reps)`

Repeatedly runs the code between the `TIMEIT_REPEAT` and the `TIMEIT_END_REPEAT` markers, automatically increasing the number of repetitions until the elapsed time exceeds the timer resolution. The macro takes as input a predefined `timeit_t` object and an integer variable to hold the number of repetitions.

`macro TIMEIT_START`

`macro TIMEIT_STOP`

Repeatedly runs the code between the `TIMEIT_START` and the `TIMEIT_STOP` markers, automatically increasing the number of repetitions until the elapsed time exceeds the timer resolution, and then prints the average elapsed cpu and wall time for a single repetition.

`macro TIMEIT_ONCE_START`

`macro TIMEIT_ONCE_STOP`

Runs the code between the `TIMEIT_ONCE_START` and the `TIMEIT_ONCE_STOP` markers exactly once and then prints the elapsed cpu and wall time. This does not give a precise measurement if the elapsed time is short compared to the timer resolution.

`macro SHOW_MEMORY_USAGE`

Retrieves memory usage information via `get_memory_usage` and prints the results.

§65. interfaces

Interfaces to other packages

65.1 Introduction

In this chapter we provide interfaces to various external packages.

65.2 NTL Interface

The NTL interface allows conversion between NTL objects and FLINT objects and vice versa. The interface is built using C++ and is not built as a part of the FLINT library library by default. To build the NTL interface one must specify the location of NTL with the `--with-ntl=path` option to configure. NTL version 5.5.2 or later is required.

`void fmpz_set_ZZ(fmpz_t rop, const ZZ& op)`

Converts an NTL `ZZ` to an `fmpz_t`.

Assumes the `fmpz_t` has already been allocated to have sufficient space.

`void fmpz_get_ZZ(ZZ& rop, const fmpz_t op)`

Converts an `fmpz_t` to an NTL `ZZ`. Allocation is automatically handled.

`void fmpz_set_ZZ_p(fmpz_t rop, const ZZ_p& op)`

Converts an NTL `ZZ_p` to an `fmpz_t`.

Assumes the `fmpz_t` has already been allocated to have sufficient space.

`void fmpz_get_ZZ_p(ZZ_p& rop, const fmpz_t op)`

Converts an `fmpz_t` to an NTL `ZZ_p`. Allocation is automatically handled. Requires that `ZZ_p::init()` has already been called.

`void fmpz_poly_get_ZZX(ZZX& rop, const fmpz_poly_t op)`

Converts an `fmpz_poly_t` to an NTL `ZZX`.

`void fmpz_poly_set_ZZX(fmpz_poly_t rop, const ZZX& op)`

Converts an NTL `ZZX` to an `fmpz_poly_t`.

```
void fmpz_mod_poly_get_ZZ_pX(ZZ_pX& rop, const
   fmpz_mod_poly_t op)
```

Converts an `fmpz_mod_poly_t` to an NTL `ZZ_pX`. Requires that `ZZ_p::init()` has already been called.

```
void fmpz_mod_poly_set_ZZ_pX(fmpz_mod_poly_t rop, const
   ZZ_pX& op)
```

Converts an NTL `ZZ_pX` to an `fmpz_mod_poly_t`.

```
void fq_get_ZZ_pE(ZZ_pE& rop, const fq_t op, const fq_ctx_t
   ctx)
```

Converts an `fq_t` to an NTL `ZZ_pE`. Requires that `ZZ_pE::init()` has already been called.

```
void fq_set_ZZ_pE(fq_t rop, const ZZ_pE& op, const fq_ctx_t
   ctx)
```

Converts and NTL `ZZ_pE` to an `fq_t`.

```
void fq_poly_get_ZZ_pEX(ZZ_pEX& rop, const fq_poly_t op,
   const fq_ctx_t ctx)
```

Converts an `fq_poly_t` to an NTL `ZZ_pEX`. Requires that `ZZ_pE::init()` has already been called.

```
void fq_poly_set_ZZ_pE(fq_poly_t rop, const ZZ_pE& op,
   const fq_ctx_t ctx)
```

Converts and NTL `ZZ_pEX` to an `fq_poly_t`.

```
void fq_get_zz_pE(zz_pE& rop, const fq_t op, const fq_ctx_t
   ctx)
```

Converts an `fq_t` to an NTL `zz_pE`. Requires that `zz_pE::init()` has already been called.

```
void fq_set_zz_pE(fq_t rop, const zz_pE& op, const fq_ctx_t
   ctx)
```

Converts and NTL `zz_pE` to an `fq_t`.

```
void fq_poly_get_zz_pEX(zz_pEX& rop, const fq_poly_t op,
   const fq_ctx_t ctx)
```

Converts an `fq_poly_t` to an NTL `zz_pEX`. Requires that `zz_pE::init()` has already been called.

```
void fq_poly_set_zz_pE(fq_poly_t rop, const zz_pE& op,
   const fq_ctx_t ctx)
```

Converts and NTL `zz_pEX` to an `fq_poly_t`.

§A. Extending the C++ wrapper

A.1 Introduction

This chapter is geared towards FLINT developers who wish to extend the C++ wrapper, chiefly by adding new functions operating on existing wrapper classes, or by adding altogether new wrapper classes for data types they implemented in C. Part of the design effort of flintxx went into trying to make it possible to do this kind of extension with only cursory knowledge of the syntax of C++, without having to understand in detail things such as partial template specialisation.

The easiest way to get started is probably to read `examples/fooxx.cpp`. As a matter of fact I hope that most day to day work on the wrapper should be doable by just copying similar code from other data types, so after reading the example you may already know everything you need.

A.2 Overview of flintxx

The flintxx library is composed of a variety of parts. The main expression template class `expression` resides in `flintxx/expression.h`. Concrete classes derive from it, and thereby automatically acquire overloaded operators etc. to construct and evaluate expression templates. Of course, this is only possible after telling the flintxx how to do the evaluation, by specialising evaluation rules defined in `flintxx/rules.h`. This file also provides convenience macros `FLINT_DEFINE_GET` etc., which can be used to simplify defining common rules. Using only these files it would already be possible to interact with flintxx.

In many situations one needs to define rules which work with varying argument types, if those types satisfy certain conditions. This can be achieved using the so-called `enable_if` idioms together with some meta programming (implemented in `flintxx/mp.h`) and type traits (mainly in `flintxx/traits.h` and `flintxx/expression_traits.h`). These are the files a third party developer should use to interact with flintxx.

In writing wrappers for FLINT C types, there are some more common idioms. These are usually expressed as macros `FLINTXX_???` and are defined in `flintxx/flint_classes.h`. As illustrated in the example, the `FLINTXX_???` macros are meant to complement, not supersede, the `FLINT_???` macros.

The above considerations and examples should explain how to modify any existing class, or write a simple new one, like `fmpzxx`. The remaining subsections deal with particular difficulties that require work beyond what has been explained so far.

A.2.1 Patterns for implementing unified coefficient access

Recall that "unified coefficient access" refers to automatically adjusting the return type of coefficient access methods, depending on the argument. Let's focus on the `fmpqxx` example, and in particular the `num` method. Consider the expression `x.num()`. There are three cases: if `x` is of type `fmpqxx` or `fmpqxx_ref`, this should return `fmpzxx_ref`. If `x` is of type `const fmpqxx`, or `fmpqxx_srcref`, this should return `fmpzxx_srcref`. Otherwise (i.e. if `x` is a lazy expression template), this should return a lazy expression template.

The way this is implemented is using a traits type. First of all, we need a new lazy function to obtain the numerator:

```
FLINT_DEFINE_UNOP(fmpqxx_num)
```

We call the function `fmpqxx_num` to hide it somewhat, and discourage calling `num` as a global function (because this won't have the unified coefficient access properties). Next (even before the expression template definition), we put the generic traits:

```
template<class Fmpq>
struct fmpq_traits {
    typedef FLINT_UNOP_BUILD_RETTYPE(fmpqxx_num, fmpzxx,
       Fmpq) numreturn_t;
    typedef numreturn_t cnumreturn_t;
    static numreturn_t num(const Fmpq& f)
        {return fmpqxx_num(f);}
    // ...
};
```

This has two typedefs `numreturn_t` and `cnumreturn_t`, which are the types the `num()` method should return on non-constant and constant instances, respectively. As this trait deals with the lazy expression case, we should return a lazy expression in both the constant and non-constant case. Thus we get by with only one static function, which can be called with a constant or non-constant argument, and which creates a new lazy expression involving our new function `fmpqxx_num`.

The `fmpqxx_expression` class can use the traits as follows:

```
// class fmpqxx_expression
    typedef detail::fmpq_traits<fmpqxx_expression> traits_t;
    typename traits_t::numreturn_t num()
        {return traits_t::num(*this);}
```

After the definition of the expression template classes, we put the following specialisations:

```
template<>
struct fmpq_traits<fmpqxx_srcref>
{
    typedef fmpzxx_srcref numreturn_t;
    typedef fmpzxx_srcref cnumreturn_t;
    template<class T>
    static cnumreturn_t num(T f)
        {return cnumreturn_t::make(fmpq_numref(f._fmpq()));}
};
template<>
struct fmpq_traits<fmpqxx_ref>
{
    typedef fmpzxx_ref numreturn_t;
    typedef fmpzxx_ref cnumreturn_t;
```

```
    template<class T>
    static cnumreturn_t num(T f)
        {return cnumreturn_t::make(fmpq_numref(f._fmpq()));}
};
template<> struct fmpq_traits<fmpqxx>
{
    typedef fmpzxx_ref numreturn_t;
    typedef fmpzxx_srcref cnumreturn_t;
    template<class T>
    static cnumreturn_t num(const T& f)
        {return cnumreturn_t::make(fmpq_numref(f._fmpq()));}
    template<class T>
    static numreturn_t num(T& f)
        {return numreturn_t::make(fmpq_numref(f._fmpq()));}
};
```

Note how we have to take care of quite a few special cases, and there is not actually any obvious duplication going on here. The way the code is written, calling `num()` on a constant instance of `fmpqxx_ref` will yield a writable object. This makes sense – given a constant instance, one can just make a (non-constant) copy, and write through it – but is not really mandatory. In general, flintxx does not support this kind of writing through constant instances of reference classes. For example, `set_zero` will not work on a `const fmpqxx_ref`. Thus it would be possible to share some code between the ref and nonref implementations of the traits classes. This is done for example in `padicxx`.

Finally, in the `rules` namespace, we have to place an implementation of `fmpqxx_num`:

```
FLINT_DEFINE_UNARY_EXPR_COND(fmpqxx_num_op, fmpzxx,
    FMPQXX_COND_S,
    fmpz_set(to._fmpz(), fmpq_numref(from._fmpq())))
```

The example of traits usage we see here illustrates how careful one has to be to avoid circular class dependencies. In particular, the traits can only be specialised after the expression template class has been defined and the special immediate cases have been `typedef`d. However, they have to be specialised before any of the related templates is instantiated.

Note also how the `fmpq_traits<fmpqxx_ref>` functions take their argument by value, thus automatically obtaining non-constant versions, while declaring the argument type via a template. This again delays instantiation, and breaks circular dependencies.

A.2.2 Nmod classes

Another common complication is when the underlying C objects are not "default constructible". For example, the `nmod_*` interfaces (`nmod_poly`, `nmod_mat`, etc.) always require a modulus for initialisation. This is mainly problematic when evaluating compound expressions which require temporary objects. In principle flintxx has an easy workaround for this: the expression template class can define a `create_temporary()` method which is used to instantiate a temporary object of the same type as the return type of the current expression. However, this only really shifts the problem: now the expression template class has to figure out its own modulus.

To complicate matters further, the moduli are not stored in a single type. The structures `nmod_mat_t`, `nmod_poly_t`, `nmod_poly_mat_t` all store a modulus. Even an expression built out of (say) `nmod_poly_mat` can return a (say) `nmod_poly` (e.g. the determinant), and so the `nmod_polyxx` class must be able to instantiate a temporary for an expression which does not contain any immediate polynomials!

As a final complication, the C objects do not actually only store the modulus n, but also extra information computed from n. This extra information is conveniently wrapped up in `nmod_t`, although most of the C interfaces do not expose this data structure directly.

These problems are overcome as follows. First of all, we do not allow any mixing of moduli in compound expressions. Thus the only task of the `create_temporary` method is to locate *any* modulus inside a subexpression. Next, we require objects with a modulus to be recognisable by type.[1] With these conventions in place, it is a matter of bookkeeping (and searching the subexpression tree) to locate a modulus.

Additionally, we have a class `nmodxx_ctx_srcref` (which is not an expression template) which stores a reference to an `nmod_t`. All classes which want to provide moduli must implement a `_ctx()` method which returns such an `nmodxx_ctx_srcref` object.

In practice, there is a type trait `traits::has_nmodxx_ctx<T>` which should be specialised for all (immediate) classes wishing to participate in the modulus determination. For example, the following lines enable looking up moduli in `nmod_poly`:

```
namespace traits {
template<> struct has_nmodxx_ctx<nmod_polyxx>
    : mp::true_ { };
template<> struct has_nmodxx_ctx<nmod_polyxx_ref>
    : mp::true_ { };
template<> struct has_nmodxx_ctx<nmod_polyxx_srcref>
    : mp::true_ { };
} // traits
```

In addition to this, as mentioned above, the `create_temporary` method has to be overridden:

```
// class nmod_polyxx
    nmodxx_ctx_srcref estimate_ctx() const
    {
        return tools::find_nmodxx_ctx(*this);
    }

    evaluated_t create_temporary() const
    {
        return evaluated_t(estimate_ctx());
    }
```

The function `tools::find_nmodxx_ctx` is implemented in `nmod_vec.h` and traverses the subexpression tree, searching for objects containing moduli, and returning the modulus of the first such object found.

A.2.3 Matrix classes

A related but subtly different problem is posed by matrix classes. Again, these are not default instantiable, but one instead needs to specify their dimension (numbers of rows and columns). Two new problems arise. First, it is very common to mix matrices of differing dimensions in one expression (consider transposing a non-square matrix). Second, FLINT does no resizing of matrices.

The latter is problematic, because flintxx uses a technique we call "temporary merging" where temporary objects needed in a calculation are only identified by type, and allowed to be reused. This reduces memory allocation and deallocation.

[1] This is why we disallow empty matrices over $\mathbb{Z}/n\mathbb{Z}[X]$. The `nmod_poly_mat_t` does not store a modulus, so we have to look at an entry.

In the current design, temporary merging cannot be used with matrix classes. The type trait `traits::use_temporary_merging<T>` must be used to disable it. After that, we can override `create_temporary` in the usual way, except that the method will now be called more often.[2]

However, the `create_temporary` method still has to figure out the dimensions of the resulting matrix. For this, we use the type trait `outsize`. There are static functions `outsize<operation>::rows(m)` and `outsize<operation>::cols(m)` which yield the dimensions. Of course, the `outsize` trait has to be specialised appropriately.

All in all, this is quite a lot of work, much of which is independent of the actual coefficient ring. For this reason, we have the helper file `flintxx/matrix.h`. It contains the following:

(i) The `matrix_traits` template.

(ii) Definition of operations for common matrix functions (e.g. transpose).

(iii) The `matrices::outsize` and `matrices::outsize_generic` templates, and appropriate specialisations for common functions (e.g. transpose, multiplication).

(iv) The templates `generic_traits_ref`, `generic_traits_srcref` and `generic_traits_nonref` to simplify unified coefficient access.

(v) Some convenience macros.

To see how these things are used, consider the example of `fmpz_matxx`. We begin with the generic traits type for unified coefficient access:

```
namespace detail {
template<class Mat>
struct fmpz_matxx_traits
    : matrices::generic_traits<Mat> { };
} // detail
```

This defines a traits type, which will be used to implement the `at(i, j)` method with unified access. The implementation defined above is the generic one, which is implemented via a call to the lazy operation `mat_at`.

Next, inside the `fmpz_matxx_expression` class, the trait is used as follows:

```
// class fmpz_matxx_expression
    typedef
       detail::fmpz_matxx_traits<fmpz_matxx_expression>
       traits_t;

    template<class Expr>
    static evaluated_t create_temporary_rowscols(
            const Expr&, slong rows, slong cols)
    {
        return evaluated_t(rows, cols);
    }

    FLINTXX_DEFINE_MATRIX_METHODS(traits_t)
```

The macro `FLINTXX_DEFINE_MATRIX_METHODS` adds methods `at`, `rows`, `cols` and `create_temporary` for us (the function `create_temporary_rowscols` is used by the implementation of `create_temporary`). However, these methods cannot work unless we provide more information to the matrix helpers. The main thing we have to do is to specialise the `matrix_traits` trait:

[2] Actually, additional care has to be taken with regard to ltuples. We ignore this here for simplicity.

```
template<>
struct matrix_traits<fmpz_matxx>
{
    template<class M> static slong rows(const M& m)
    {
        return fmpz_mat_nrows(m._mat());
    }
    template<class M> static slong cols(const M& m)
    {
        return fmpz_mat_ncols(m._mat());
    }

    template<class M> static fmpzxx_srcref at(const M& m,
       slong i, slong j)
    {
        return fmpzxx_srcref::make(fmpz_mat_entry(m._mat(),
           i, j));
    }
    template<class M> static fmpzxx_ref at(M& m, slong i,
       slong j)
    {
        return fmpzxx_ref::make(fmpz_mat_entry(m._mat(), i,
           j));
    }
};
```

This trait means that the class `fmpz_matxx` (and all the classes built from it, like reference types and lazy expressions) wants to use the matrix helpers framework. It also specifies how to determine the dimension of immediate objects, and how to access their coefficients. (The framework will never call these trait functions with non-immediates, but we use a template nonetheless so the functions also work on reference types.)

Furthermore, we have to specialise some more type traits to disable temporary merging. This is done using the following convenience macro:

```
// temporary instantiation stuff
FLINTXX_DEFINE_TEMPORARY_RULES(fmpz_matxx)
```

The matrix class is now almost operational. In standard expressions (involving transpose, matrix multiplication etc.) temporary objects of the correct dimensions will be allocated automatically. What remains to be done is to implement the rest of the unified coefficient access, and matrix dimension rules for more esoteric functions.

The unified coefficient access implementation can in principle be done precisely as described in a previous subsection. However, it is essentially the same in all matrix classes, which is why we provide default implementations:

```
template<>
struct fmpz_matxx_traits<fmpz_matxx_srcref>
    : matrices::generic_traits_srcref<fmpzxx_srcref> { };
template<>
struct fmpz_matxx_traits<fmpz_matxx_ref>
    : matrices::generic_traits_ref<fmpzxx_ref> { };
template<> struct fmpz_matxx_traits<fmpz_matxx>
    : matrices::generic_traits_nonref<fmpzxx_ref,
       fmpzxx_srcref> { };
```

We still have to provide a `mat_at` rule:

```
FLINT_DEFINE_THREEARY_EXPR_COND3(mat_at_op, fmpzxx,
        FMPZ_MATXX_COND_S, traits::fits_into_slong,
           traits::fits_into_slong,
        fmpz_set(to._fmpz(), fmpz_mat_entry(e1._mat(), e2,
           e3)))
```

Now all that remains to be done is to inform the matrices framework of dimension calculations for functions it is not aware of. For `fmpz_matxx`, one example is a particular multiplication algorithm. Of course, the rule is just the same as for ordinary multiplication, so the specialisation is very simple:

```
namespace matrices {
template<>
struct outsize<operations::mul_multi_mod_op>
    : outsize<operations::times> { };
} // matrices
```

In general, if there is no rule, a default case will be used. If the number of arguments is not two, this assumes that the first argument is a matrix, and that the dimensions of the output will match the dimensions of this first argument. If there are two arguments, at least one of which is a matrix, then the dimension of one of those is returned (with no guarantee which if there are two matrices). This default rule works for matrix addition, matrix-scalar multiplication (on either side), and "elementwise" operations.

A.2.4 Padic classes

The classes representing padic numbers (`padicxx`, `padic_polyxx`, `padic_matxx` and `qadic`) come with their own difficulties. These are twofold.

First of all, padic number data is split into two data structures: the actual number data (e.g. `padic_t`), and a *context* (`padic_ctx_t`) containing meta data. Any operation on padic numbers needs to be passed both the number data and a context. In order to facilitate this, all C++ wrappers for padics store a reference to a context. We can then use a lookup scheme similar to the `nmod_*` case to find a context suitable for a whole expression.

One difference is that the context reference has to be stored with every immediate object (e.g. in `padic_data`). This also includes the reference types, and hence requires specialising `flint_classes::ref_data` and `flint_classes::srcref_data`.

Secondly, all padic computations are necessarily just approximations. That is to say, like real numbers, padic numbers cannot in general be stored exactly. Thus every padic number also stores its own precision. When the C library is asked to perform an operations, such as `a = b + c`, it will treat `b` and `c` as exact padic numbers (similarly to how we can treat a decimal approximation to a real number, such as 3.14, as the exact real number $3.140000\ldots$). Then it implements an algorithm which is equivalent to performing the computation `b + c` exactly and truncating to the precision of `c` in the end. This gives very well-defined semantics, and very fine control over the behaviour, but is unfortunately not very well-suited to implementing automatic evaluation of compound expressions. In flintxx, we have decided to (notionally) assign a precision to every compound expression. All temporaries used will be instantiated with this precision.

The precision of a compound expression is computed as the maximum of the precision of all of its (immediate) subexpressions. In order to still be able to change this precision, all immediates have a `.toN(n)` method which (notionally) raises or lowers their precision. This is implemented by extending the functionality of the `padicxx_srcref` type. Namely, in `padicxx_srcref`, in addition to a `const padic_struct*` and `padicxx_ctx_srcref`, we also store an additional `slong N` representing the notional precision. The usual constructors of `padicxx_srcref` just initialise `N` from the underlying `padic_struct`. But

we provide additional constructors which allow setting this parameter explicitly. The `toN` method makes use of these constructors.

We thus see that quite a bit of infrastructure is needed to build a new padic class. To facilitate this, `padicxx.h` defines some convenience macros. In summary, the following steps are necessary:

(i) Implement a traits type for computing `prec()` and `val()`. In the general case, `prec()` should return `tools::padic_output_prec`.

(ii) In the expression class, typedef `traits_t` and invoke the macro `PADICXX_DEFINE_STD`.[3]

(iii) Enable the trait `has_padicxx_ctx` for your immediate classes.

(iv) Specialise the traits type. For reference and non-reference immediates, it should return the precision stored with the objects. For srcref, it should return the N field in `_data()`.[4]

(v) Invoke the macro `PADICXX_DEFINE_REF_STRUCTS(classname, structname, precname)`. Here `classname` is the name of the non-reference immediate (e.g. `padicxx`), `structname` is the name of the underlying C struct (e.g. `padic_struct`) and `precname` is the name of the function for accessing the precision (e.g. `padic_prec`). This specialises the `flint_classes::ref_data` and `flint_classes::srcref_data` templates, adding a context reference and `toN` support.

The implementation of `qadicxx` is slightly different since it uses a larger context. But this contains `padic_ctx` as a substruct, and hence it is not difficult to incorporate `qadicxx` into the context lookup infrastructure. See the source for more details.

A.2.5 Vector classes

Vectors do not have very good representation in FLINT. Usually, but not universally, a vector of type `foo_t` is represented as `foo_struct*`, with size passed in explicitly. Some vector types have support functions in FLINT, typically all underscored.

In flintxx we implement some first class vectors, but the interfaces are not very well developed and relatively ad-hoc.

A.3 Some tidbits and caveats

One C++ idiom which may initially look staggering is `enable_if`. A typical usage is as follows:

```
template<class T>
void foo(T,
     typename enable_if<is_signed_integer<T> >::type* = 0)
{
     cout << 'a';
}

template<class T>
void foo(T,
     typename enable_if<is_unsigned_integer<T> >::type* = 0)
```

[3] This adds context estimation, `toN`, `prec` and `val` methods. Alternatively invoke a subset of `PADICXX_DEFINE_CTX`, `PADICXX_DEFINE_ESTIMATE_CTX`, `PADICXX_DEFINE_TON`, `PADICXX_DEFINE_PREC`, `PADICXX_DEFINE_VAL`.

[4] `val()` can be an ordinary unified access implementation.

```
{
    cout << 'b';
}

...
foo(4);    // a
foo(4u);   // b
foo("x"); // error
```

What happens syntactically here is that we define two functions called `foo`. In C++ this is allowed, such a function is called *overloaded*, and depending on the arguments it is called with, the so-called overloading algorithm decides which instance to call. Additionally, both functions we define are actually *function templates*. Normally, a function template like `template<class T> void foo(T)` can be called with arbitrary arguments, and so overloading makes little sense.[5] But of course the point to observe is that our overloaded functions have an additional argument, which depends on `T`. Syntactically, there is an un-named pointer argument, which defaults to zero.[6] What happens now is that depending on `T`, the function signature may be syntactically invalid! Indeed, the template `enable_if<cond>` has a member `type` only if `cond` is true. Otherwise `enable_if<cond>` is just completely empty, and the function signature makes no sense. But this is fine, according to the SFINAE[7] rule of C++, if something like this happens during overload resolution, the algorithm just discards this possibility. So in general, what the overload algorithm does it looks for all function templates (and functions) which the overloaded call could possibly match (this by itself is a complicated procedure, because of so-called argument-dependent lookup ...), then tries to instantiate all the function templates, discards all which have substitution failures, and then tries to find the best match in the remaining list.

Similar rules also apply in resolving partially specialised template types. In this case, the pattern usually looks like this:

```
template<class T, class Enable = void>
struct foo
{
    static const int a = 0;
};

template<class T>
struct foo<T,
    typename enable_if<is_unsigned_integer<T> >::type>
{
    static const int a = 1;
};

template<class T>
struct foo<T,
    typename enable_if<is_signed_integer<T> >::type>
{
    static const int a = 2;
};
```

[5] One may still overload e.g. `foo(T&)` and `foo(const T&)`, and say `foo(int)`; the overloading algorithm determines a "best match", and so this overloading can still be useful. However the details of what constitutes a best match are complicated, and if there are several equally-good matches (none of which is "best") then an error will be issued.

[6] It is not terribly important that this is a pointer type, it could be say int, but `enable_if<...>::type` is defined to be void, and so converting to a pointer is the easiest way to turn this into a function argument.

[7] "Substitution Failure Is Not An Error"

```
...
foo<const char*>::a; // 0
foo<int>::a;         // 2
foo<unsigned>::a;    // 1
```

It may seem tempting to enable member functions conditionally on class template arguments, like this:

```
template<class T>
struct foo
{
    void bar(typename enable_if<is_integer<T> >::type* = 0)
       {...}
};
```

But this is not possible. The problem is, more or less, that when instantiating `foo<int>`, after replacing `T` by `int`, the entire class signature has to be well-formed, which it just is not. Nonetheless it is typically straightforward to workaround this. You just have to ensure that the signature of your member function is well-formed for any template argument `T`. Then in the function body, which is instantiated only when the function is actually called, you can add a `STATIC_ASSERT` to make it plain that this function should only be called with integers, etc.

Another thing to keep in mind is that in C++98, the expressions `> >` and `>>` are very different. The latter is a shift operator, and cannot be used to close template argument lists. Thus expressions like `not_<is_integer<T>>` are invalid in C++98, one has to write `not_<is_intiger<T> >` instead.

On a similar note, the meaning of an expression involving template arguments can in some situations depend on the template parameter, and sometimes this has to be made explicit. A typical situation is `T::something`, which may refer to either a static member value, or a member type of T. If the latter is desired, in most situations one needs to write `typename T::something`. Similarly if `T` has a member template (function, say), then one cannot write `T::something<int>()`, but instead has to write `T::template something<int>()`. Most modern compilers will suggest the alternative syntax when such an error is encountered.

A.4 Rules and standard methods

A typical expression template class begins with the following lines of code:

```
template<class Operation, class Data>
class some_expression
    : public
       expression<derived_wrapper<some_expression>,
       Operation, Data>
{
    // ...
};
```

We document here methods this class inherits from its base, and how they relate to rules.

There are the following public typedefs:

ev_traits_t A specialisation of `detail::evaluation_traits`. Used to compute the rule for evaluation.

derived_t The specialised derived class.

evaluated_t The resulting type of evaluating this expression.

evaluation_return_t The return type of `evaluate()`. This differs from the above for immediates, where evaluation returns a reference instead of a copy.

data_t The same as `Data`.

operation_t The same as `Operation`.

A.4.1 Standard methods

```
data_t& some_expression::_data()
```

```
const data_t& some_expression::_data() const
```

Obtain the data related to this expression template.

```
evaluated_t some_expression::create_temporary() const
```

Default instantiate a temporary. Override this if your class is not default instantiable.

```
template<class T> T some_expression::to() const
```

Convert self to type `T` (after evaluating). Uses `rules::conversion`.

```
void some_expression::print(std::ostream& o) const
```

Print self to `o`. Uses `rules::print` or `rules::to_string`.

```
int some_expression::print(FILE* f = stdout) const
```

Print self to `f`. Uses `rules::cprint`.

```
int some_expression::print_pretty(FILE* f = stdout) const
```

Print self to `f`. Uses `rules::print_pretty`.

```
template<class T> int some_expression::print_pretty(const
   T& extra, FILE* f = stdout) const
```

Print self to `f`. Uses `rules::print_pretty` with two arguments.

```
int some_expression::read(FILE* f = stdin)
```

Read self from `f`. Uses `rules::read`.

```
const evaluation_return_t some_expression::evaluate() const
```

Evaluate self.

```
template<class T> void some_expression::set(const T& t)
```

Assign `t` to self. Uses evaluation and/or `rules::assignment`.

```
template<class T> bool some_expression::equals(const T& t)
   const
```

Determine if `t` is equal to self. Uses `rules::equals` or `rules::cmp`.

A.4.2 Global functions

In addition to member functions, flintxx also provides a number of global functions. In general these operate on sets of arguments at least one of which derives from `expression`, and are conditionally enabled only if the relevant operation is implemented (via a rule).

```
template<class Expr> std::ostream& operator<<(std::ostream&
   o, const Expr& e)
```

Print `e` to `o`. Uses the member `print`.

```
template<class Expr1, class Expr2> bool operator==(const
   Expr1&, const Expr2&)

template<class Expr1, class Expr2> bool operator!=(const
   Expr1&, const Expr2&)
```

Compare two expressions. Uses the member `equals`.

```
template<class Expr1, class Expr2> bool operator??(const
   Expr1&, const Expr2&)
```

Relational operators `< > <= =>` are implemented using `rules::cmp`.

```
template<class Expr1, class Expr2> ?? operator??(const
   Expr1&, const Expr2&)
```

Arithmetic operators `+ - * / % & | ^ << >>` are implemented by constructing new expression templates with operation `operations::plus` etc.

```
template<class Expr1> ?? operator??(const Expr1&)
```

Unary operators `- ~` are implemented by constructing new expression templates with operation `operations::negate` and `operations::complement`.

```
template<class Expr1, class Expr2> ?? operator?=(const
   Expr1&, const Expr2&)
```

Arithmetic-assignment operators `+= -= *= /= %= |= &= ^=`.

```
template<class Expr1> int print(const Expr1&)

template<class Expr1> int print(FILE*f, const Expr1&)

template<class Expr1> int print_pretty(const Expr1&)

template<class Expr1> int print_pretty(FILE*f, const Expr1&)

template<class Expr1, class T> int print_pretty(const
   Expr1&, const T& extra)

template<class Expr1, class T> int print_pretty(FILE*f,
   const Expr1&, const T& extra)
```

Forward to member.

```
template<class Expr1, class Expr2> void swap(Expr1& e1,
   Expr2& e2)
```

Swap `e1` and `e2` using `rules::swap`. Note that via ADL, this can be used by STL containers.

A.4.3 flintxx classes

The flint wrapper classes share some other common interfaces. These have to be enabled using the convenience macros in `flintxx/flint_classes.h` (q.v.). Here `accessname` and `ctype` are specified via the macros. For e.g. `fmpz_polyxx` these are `_poly` and `fmpz_poly_struct`.

```
?? some_expression::accessname()

?? some_expression::accessname() const
```

Obtain a reference to the underlying C struct. This is only available on immediate expressions.

```
some_expression_ref::some_expression_ref(some_expression&)

some_expression_srcref::some_expression_srcref(const
   some_expression&)

some_expression_srcref::some_expression_srcref(some_expression_ref)
```

Build a reference type. Note that these are *implicit* constructors.

```
static some_expression_ref some_expression_ref::make(ctype*)

static some_expression_srcref
   some_expression_srcref::make(const ctype*)
```

Build a reference type from a pointer to the underlying C struct.

A.5 Convenience macros

A.5.1 flintxx/rules.h

```
FLINT_DEFINE_GET2(name, totype, fromtype1, fromtype2, eval)
```

Specialise a getter called `name`, which takes arguments `e1` of type `fromtype1` and `e2` of type `fromtype2`. It returns `totype` by executing `eval`.

```
FLINT_DEFINE_GET(name, totype, fromtype, eval)
```

Same as `FLINT_DEFINE_GET2(name, totype, fromtype, fromtype, eval)`.

```
FLINT_DEFINE_GET_COND(name, totye, cond, eval)
```

Specialise a getter called `name`, which takes an argument `from` of type `T:cond` It returns `totype` by executing `eval`.

```
FLINT_DEFINE_DOIT(name, totype, fromtype, eval)
```

Specialise a doit rule called `name`, which takes arguments `to` of type `totype&` and `from` of type `const fromtype&`, and executes `eval`.

```
FLINT_DEFINE_DOIT_COND(name, totype, cond, eval)
```

Same as above, but takes `const T& from` for any `T:cond`.

```
FLINT_DEFINE_DOIT_COND2(name, cond1, cond2, eval)
```

Same as `FLINT_DEFINE_DOIT_COND`, but takes `T& to` and `const U& from` for any `T` satisfying `cond1<T>` and `U` satisfying `cond2<U>`.

`FLINT_DEFINE_PRINT_COND(cond, eval)`

Specialise the `cprint` rule. This takes a arguments `FILE* to` and `const T& from` for any `T:cond`. It prints `from` to `to` and returns `int` by executing `eval`.

`FLINT_DEFINE_PRINT_PRETTY_COND(cond, eval)`

Same as above, but with `print_pretty` instead of `cprint`.

`FLINT_DEFINE_PRINT_PRETTY_COND2(cond, extratype, eval)`

Same as above, but takes an additional argument `extratype extra`. Useful e.g. when printing polynomials and taking an extra variable name.

`FLINT_DEFINE_READ_COND(cond, eval)`

Specialise the `read` rule. This takes a arguments `FILE* from` and `T& to` for any `T:cond`. It reads `to` from `from` and returns `int` by executing `eval`.

`FLINT_DEFINE_UNARY_EXPR_(name, rtype, type, eval)`

Specialise the unary expression rule for `operations::name` with nominal return type `rtype`. It takes arguments `V& to` and `const type& from`. Here `V` is any type which `rtype` can be evaluated into. Executes `eval`.

`FLINT_DEFINE_UNARY_EXPR(name, type, eval)`

Same as `FLINT_DEFINE_UNARY_EXPR_(name, type, type, eval)`.

`FLINT_DEFINE_BINARY_EXPR2(name, rtype, type1, type2, eval)`

Specialise the binary expression rule for `operations::name` of nominal return type `rtype`, and arguments `type1` and `type2`.

`FLINT_DEFINE_BINARY_EXPR(name, type, eval)`

Same as `FLINT_DEFINE_BINARY_EXPR2(name, type, type, type, eval)`.

`FLINT_DEFINE_CBINARY_EXPR(name, type, eval)`

Same as above, but with `commutative_binary_expression` instead of `binary_expression`.

`FLINT_DEFINE_BINARY_EXPR_COND(name, type, cond, eval)`

`FLINT_DEFINE_CBINARY_EXPR_COND(name, type, cond, eval)`

Specialise the (commutative) binary expression rule for `operations::name` of nominal return type `type`, and arguments `type` and `T:cond`.

```
FLINT_DEFINE_BINARY_EXPR_COND2(name, rettype, cond1, cond2,
    eval)
```

```
FLINT_DEFINE_CBINARY_EXPR_COND2(name, rettype, cond1,
    cond2, eval)
```

Specialise the (commutative) binary expression rule for `operations::name` of nominal return type `rettype`, and arguments `T:cond1` and `U:cond2`.

```
FLINT_DEFINE_THREEARY_EXPR_COND3(name, rettype, cond1,
   cond2, cond3, eval)

FLINT_DEFINE_FOURARY_EXPR_COND4(name, retttype, cond1 ...
   cond4, eval)

FLINT_DEFINE_FIVEARY_EXPR_COND5(name, rettype, cond1 ...
   cond5, eval)

FLINT_DEFINE_SIXARY_EXPR_COND6(name, rettype, cond1 ...
   cond6, eval)

FLINT_DEFINE_SEVENARY_EXPR_COND7(name, rettype, cond1 ...
   cond7, eval)
```

Specialise higher order rules, similarly to the above.

```
FLINT_DEFINE_THREEARY_EXPR(name, retttype, T1, T2, T3, eval)
```

Specialise a threeary expression rule unconditionally.

A.5.2 flintxx/expression.h

```
FLINT_DEFINE_UNNOP(name)

FLINT_DEFINE_BINOP(name)

FLINT_DEFINE_THREEARY(name)

FLINT_DEFINE_FOURARY(name)

FLINT_DEFINE_FIVEARY(name)

FLINT_DEFINE_SIXARY(name)

FLINT_DEFINE_SEVENARY(name)
```

Introduce a new n-ary operation `operations::##name##_op` and make it available. This has to be called in namespace `flint`.

```
FLINT_DEFINE_UNNOP_HERE(name)

FLINT_DEFINE_BINOP_HERE(name)

FLINT_DEFINE_THREEARY_HERE(name)

FLINT_DEFINE_FOURARY_HERE(name)

FLINT_DEFINE_FIVEARY_HERE(name)

FLINT_DEFINE_SIXARY_HERE(name)

FLINT_DEFINE_SEVENARY_HERE(name)
```

Make the n-ary operation `operations::##name##_op` available in the current namespace.

```
FLINT_DEFINE_THREEARY_HERE_2DEFAULT(name, type1, val1,
   type2, val2)
```

Make the threeary operation `name` available in current namespace, but with only two arguments, the second of which is of type `type1` and defaults to `val1`, and the third argument always (implicitly) of type `type2` and value `val2`. The suggested usage of this macro is to first call `FLINT_DEFINE_THREEARY_HERE` (or `FLINT_DEFINE_THREEARY`), and then call `FLINT_DEFINE_THREEARY_HERE_2DEFAULT`. The effect will be an operation which can be invoked with 1, 2 or 3 arguments.

```
FLINT_UNOP_ENABLE_RETTYPE(name, T1)

FLINT_BINOP_ENABLE_RETTYPE(name, T1, T2)

FLINT_THREEARY_ENABLE_RETTYPE(name, T1, T2, T3)

FLINT_FOURARY_ENABLE_RETTYPE(name, T1, T2, T3, T4)

FLINT_FIVEARY_ENABLE_RETTYPE(name, T1, T2, T3, T4, T5)

FLINT_SIXARY_ENABLE_RETTYPE(name, T1, T2, T3, T4, T5, T6)

FLINT_SEVENARY_ENABLE_RETTYPE(name, T1, T2, T3, T4, T5, T6,
   T7)
```

Obtain the resulting type of invoking `name` with arguments of types `T1`, ..., `Tn` if this is possible. Otherwise results in an (SFINAE) error.

```
FLINT_UNOP_BUILD_RETTYPE(name, rettype, T)
```

Obtain the resulting type (i.e. expression template) of invoking `name` with argument type `T`, assuming the nominal return type is `rettype`. This version is sometimes necessary to break cyclic dependencies.

A.5.3 flintxx/flint_classes.h

```
FLINTXX_DEFINE_BASICS(name)
```

Add standard constructors (forwarded to `data_t`, and implicit ones for reference types). Here `name` is the name of the expression template class.

```
FLINTXX_DEFINE_C_REF(name, ctype, accessname)
```

Enable the reference types scheme.

```
FLINTXX_DEFINE_FORWARD_STATIC(funcname)
```

Add a statically forwarded constructor (similar to `make` for reference types) which invokes a static constructor of the same name of `data_t`.

```
FLINTXX_DEFINE_MEMBER_UNOP_RTYPE(rettype, name)
```

Add a no-argument member function which applies self to the lazy function `name`, where `name` has nominal return type `rettype`. (The return type has to be specified to break circular dependencies.)

```
FLINTXX_DEFINE_MEMBER_UNOP(name)
```

Same as above, but where the nominal return type is the (evaluated type of the) current expression template class.

```
FLINTXX_DEFINE_MEMBER_BINOP(name)
```

`FLINTXX_DEFINE_MEMBER_3OP(name)`

`FLINTXX_DEFINE_MEMBER_4OP(name)`

`FLINTXX_DEFINE_MEMBER_5OP(name)`

Add a member function which `n-1` arguments, the result of which is to invoke `name` on self and the arguments (in that order).

`FLINTXX_COND_S(Base)`

`FLINTXX_COND_T(Base)`

Expands to a condition (which can be passed to e.g. `FLINT_DEFINE_CBINARY_EXPR_COND2`) appropriate for testing a source/target of type `Base`.

`FLINTXX_DEFINE_TO_STR(Base, eval)`

Add a `to_string` rule which works well with the `*_get_str` functions in FLINT.

`FLINTXX_DEFINE_SWAP(Base, eval)`

Add a swap rule.

`FLINTXX_DEFINE_CONVERSION_TMP(totype, Base, eval)`

Define a conversion rule from `Base` to `totype`, which default-constructs a temporary object `to` of type `totype`, then executes `eval`, and then returns `to`.

`FLINTXX_DEFINE_CMP(Base, eval)`

`FLINTXX_DEFINE_EQUALS(Base, eval)`

Define a cmp/equality rule.

`FLINTXX_DEFINE_ASSIGN_STR(Base, eval)`

Define a string assignment rule (used by many polynomial classes).

A.5.4 flintxx/matrix.h

`FLINTXX_DEFINE_MATRIX_METHODS(Traits)`

Inside a matrix expression template class definition, given the unified access traits `Traits` appropriate for this class, define the standard methods `rows, cols, create_temporary`.

`FLINTXX_DEFINE_TEMPORARY_RULES(Matrix)`

Given a matrix expression template class `Matrix`, define appropriate temporary instantiation rule, disable temporary merging, etc.

A.6 Helper functions

A.6.1 flintxx/flint_exception.h

```
void execution_check(bool worked, const std::string& where,
    const std::string& context)
```

If `worked` is true, do nothing. Else raise a `flint_exception` with message `context + "computation failed: "+ where`.

A.6.2 permxx.h

```
slong* maybe_perm_data(permxx* p)
```

Return `0` if `p == 0`, and else the underlying data. It is helpful to use this together with `traits::is_maybe_perm` as condition.

References

[1] John Abbott, Manuel Bronstein, and Thom Mulders, *Fast deterministic computation of determinants of dense matrices*, In proceedings of ACM International Symposium on Symbolic and Algebraic Computation, ACM Press, 1999, pp. 1997–2004.

[2] Tom Apostol, *Modular functions and dirichlet series in number theory*, second ed., Springer, 1997.

[3] Andrew Arnold and Michael Monagan, *Calculating cyclotomic polynomials*, Mathematics of Computation **80** (2011), no. 276, 2359–2379.

[4] Robert Baillie and Jr. Wagstaff, Samuel S., *Lucas pseudoprimes*, Mathematics of Computation **35** (1980), no. 152, pp. 1391–1417.

[5] D. Berend and T. Tassa, *Improved bounds on Bell numbers and on moments of sums of random variables*, Probability and Mathematical Statistics **30** (2010), pp. 185–205.

[6] R. P. Brent and H. T. Kung, *Fast algorithms for manipulating formal power series*, J. ACM **25** (1978), no. 4, 581–595.

[7] J.P. Buhler, R.E. Crandall, and R.W. Sompolski, *Irregular primes to one million*, Math. Comp. **59** (1992), no. 2000, 717–722.

[8] Henri Cohen, *A course in computational algebraic number theory*, second ed., Springer, 1996.

[9] Richard Crandall and Carl Pomerance, *Prime numbers: A computational perspective*, second ed., Springer, August 2005.

[10] Marc Deleglise, Jean-Louis Niclas, and Paul Zimmermann, *Landau's function for one million billions*, J. Théor. Nombres Bordeaux **20** (2009), no. 3, 625–671.

[11] Pierre Dusart, *The kth prime is greater than* $k(\ln k + \ln\ln k - 1)$ *for* $k \geq 2$, Math. Comp. **68** (1999), no. 225, 411–415.

[12] Jason E. Gower and Samuel S. Wagstaff, Jr., *Square form factorization*, Math. Comp. **77** (2008), no. 261, 551–588.

[13] Torbjörn Granlund and Niels Möller, *Improved division by invariant integers*, IEEE Transactions on Computers **99** (2010), no. PrePrints, draft version available at http://www.lysator.liu.se/~nisse/archive/draft-division-paper.pdf.

[14] Torbjörn Granlund and Peter L. Montgomery, *Division by invariant integers using multiplication*, SIGPLAN Not. **29** (1994), 61–72.

[15] Guillaume Hanrot and Paul Zimmermann, *Newton iteration revisited*, http://www.loria.fr/~zimmerma/papers/fastnewton.ps.gz, 2004.

[16] William Hart, *A one line factoring algorithm*, http://sage.math.washington.edu/home/wbhart/onelinefactor.pdf, 2009.

[17] Peter Henrici, *A subroutine for computations with rational numbers*, J. ACM **3** (1956), no. 1, 6–9, http://doi.acm.org/10.1145/320815.320818.

[18] Ellis Horowitz, *Algorithms for rational function arithmetic operations*, Annual ACM Symposium on Theory of Computing: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing (Denver) (1972), 108–118, http://doi.acm.org/10.1145/800152.804903.

[19] Donald Knuth, *Notes on generalized dedekind sums*, Acta Arithmetica **33** (1977), 297–325.

[20] ———, *The art of computer programming vol. 2, seminumerical algorithms*, third ed., Addison–Wesley, Reading, Massachusetts, 1997.

[21] R. F. Lukes, C. D. Patterson, and H. C. Williams, *Some results on pseudosquares*, Math. Comp. **65** (1996), no. 213, 361–372, S25–S27, available at http://www.ams.org/journals/mcom/1996-65-213/S0025-5718-96-00678-3/S0025-5718-96-00678-3.pdf.

[22] Jean-Pierre Massias and Guy Robin, *Bornes effectives pour certaines fonctions concernant les nombres premiers*, J. Théor. Nombres Bordeaux **8** (1996), no. 1, 215–242.

[23] Thom Mulders, *On short multiplications and divisions*, AAECC **11** (2000), 69–88.

[24] George Nakos, Peter Turner, and Robert Williams, *Fraction-free algorithms for linear and polynomial equations*, ACM SIGSAM Bull. **31** (1997), no. 3, 11–19.

[25] Hans Rademacher, *On the partition function $p(n)$*, Proc. London Math. Soc **43** (1937), 241–254.

[26] J. Barkley Rosser and Lowell Schoenfeld, *Approximate formulas for some functions of prime numbers*, Illinois J. Math. **6** (1962), 64–94.

[27] K. Thull and C. Yap, *A unified approach to HGCD algorithms for polynomials and integers*, (1990).

[28] W. Watkins and J. Zeitlin, *The minimal polynomial of* $\cos(2\pi/n)$, The American Mathematical Monthly **100** (1993), no. 5, 471–474.

[29] A. L. Whiteman, *A sum connected with the series for the partition function*, Pacific Journal of Mathematics **6** (1956), no. 1, 159–176.

[30] D. Zeilberger, *The J.C.P. Miller recurrence for exponentiating a polynomial, and its q-analog*, Journal of Difference Equations and Applications **1** (1995), 57–60.