

FLINT

Fast Library for Number Theory

Version 2.1.0

9 March 2011

William Hart*, Fredrik Johansson†, Sebastian Pancratz‡

* Supported by EPSRC Grant EP/G004870/1

† Supported by Austrian Science Foundation (FWF) Grant Y464-N18

‡ Supported by European Research Council Grant 204083

Contents

1	Introduction	1
2	Building and using FLINT	3
3	Test code	5
4	Reporting bugs	7
5	Contributors	9
6	Example programs	11
7	FLINT macros	13
8	fmpz	15
8.1	Introduction	15
8.2	Simple example	16
8.3	Memory management	16
8.4	Random generation	16
8.5	Conversion	17
8.6	Input and output	18
8.7	Basic properties and manipulation	19
8.8	Comparison	19
8.9	Basic arithmetic	19
8.10	Greatest common divisor	22
8.11	Modular arithmetic	22
8.12	Bit packing and unpacking	22
8.13	Chinese remaindering	23
9	fmpz_vec	25
9.1	Memory management	25
9.2	Randomisation	25
9.3	Bit sizes	25
9.4	Input and output	25
9.5	Assignment and basic manipulation	26
9.6	Comparison	26
9.7	Sorting	27
9.8	Addition and subtraction	27
9.9	Scalar multiplication and division	27
9.10	Gaussian content	29
10	fmpz_mat	31
10.1	Introduction	31
10.2	Simple example	31
10.3	Memory management	32

10.4	Random matrix generation	32
10.5	Basic assignment and manipulation	34
10.6	Input and output	34
10.7	Comparison	35
10.8	Transpose	35
10.9	Addition and subtraction	35
10.10	Multiplication	35
10.11	Inverse	36
10.12	Determinant	36
10.13	Rank	37
10.14	Nonsingular solving	37
10.15	Kernel	38
10.16	Fraction-free row reduction	38
11	fmpz_poly	39
11.1	Introduction	39
11.2	Simple example	39
11.3	Definition of the fmpz_poly_t type	40
11.4	Memory management	40
11.5	Polynomial parameters	41
11.6	Assignment and basic manipulation	41
11.7	Randomisation	43
11.8	Getting and setting coefficients	43
11.9	Comparison	44
11.10	Addition and subtraction	44
11.11	Scalar multiplication and division	45
11.12	Bit packing	46
11.13	Multiplication	46
11.14	Powering	49
11.15	Shifting	50
11.16	Norms	51
11.17	Greatest common divisor	51
11.18	Gaussian content	52
11.19	Euclidean division	52
11.20	Power series division	55
11.21	Pseudo division	56
11.22	Derivative	58
11.23	Evaluation	58
11.24	Composition	60
11.25	Signature	60
11.26	Input and output	61
12	fmpq_poly	65
12.1	Introduction	65
12.2	Memory management	65
12.3	Polynomial parameters	67
12.4	Accessing the numerator and denominator	67
12.5	Random testing	67
12.6	Assignment, swap, negation	68
12.7	Setting and getting coefficients	69
12.8	Comparison	70
12.9	Addition and subtraction	70
12.10	Scalar multiplication and division	71
12.11	Multiplication	73

12.12	Powering	73
12.13	Shifting	73
12.14	Euclidean division	74
12.15	Power series division	74
12.16	Derivative and integral	75
12.17	Square roots	75
12.18	Transcendental functions	76
12.19	Evaluation	79
12.20	Composition	79
12.21	Gaussian content	80
12.22	Square-free	80
12.23	Input and output	81
13	nmod_vec	83
13.1	Memory management	83
13.2	Modular reduction and arithmetic	83
13.3	Random functions	84
13.4	Basic manipulation and comparison	84
13.5	Arithmetic operations	85
14	nmod_poly	87
14.1	Memory management	87
14.2	Polynomial properties	88
14.3	Assignment and basic manipulation	88
14.4	Randomisation	88
14.5	Getting and setting coefficients	89
14.6	Input and output	89
14.7	Comparison	90
14.8	Shifting	90
14.9	Addition and subtraction	90
14.10	Scalar multiplication and division	91
14.11	Bit packing and unpacking	91
14.12	Multiplication	91
14.13	Powering	93
14.14	Division	94
14.15	Derivative and integral	97
14.16	Evaluation	98
14.17	Composition	98
14.18	GCD	99
14.19	Square roots	99
14.20	Transcendental functions	100
15	nmod_mat	105
15.1	Introduction	105
15.2	Memory management	105
15.3	Printing	106
15.4	Random matrix generation	106
15.5	Matrix arithmetic	106
15.6	Row reduction, solving	108
16	arith	111
16.1	Introduction	111
16.2	Factoring integers	111
16.3	Primorials	112
16.4	Harmonic numbers	112

16.5	Stirling numbers	112
16.6	Bell numbers	113
16.7	Bernoulli numbers and polynomials	114
16.8	Euler numbers and polynomials	115
16.9	Legendre polynomials	116
16.10	Multiplicative functions	116
16.11	Swinerton-Dyer polynomials	117
16.12	Partition function	117
16.13	Landau's function	118
17	ulong_extras	119
17.1	Random functions	119
17.2	Basic arithmetic	120
17.3	Miscellaneous	120
17.4	Basic arithmetic with precomputed inverses	120
17.5	Greatest common divisor	122
17.6	Jacobi and Kronecker symbols	123
17.7	Modular Arithmetic	123
17.8	Prime number generation and counting	124
17.9	Primality testing	125
17.10	Square root and perfect power testing	127
17.11	Factorisation	128
17.12	Arithmetic functions	131
17.13	Factorials	131
18	long_extras	133
18.1	Random functions	133
19	longlong.h	135
19.1	Auxiliary asm macros	135
20	mpn_extras	137
20.1	Utility functions	137
20.2	Divisibility	137
20.3	Special numbers	138
21	profiler	139
21.1	Timer based on the cycle counter	139
21.2	Framework for repeatedly sampling a single target	140

§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, though we do not yet provide this facility.

FLINT is currently maintained by William Hart of Warwick University in the UK. Its main authors are William Hart, Sebastian Pancratz, Fredrik Johansson, Andy Novocin and David Harvey (no longer active).

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. As of version 2.0, FLINT required GCC version 2.96 or later, MPIR 2.1.1 or later and MPFR 3.0.0 or later.

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. Building and using 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 MPIR version 2.1.1 or later and MPFR 3.0.0 or later.

To configure FLINT you must specify where 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/local` in the `lib` and `include` directories as usual, simply specify the top level location, e.g. `/usr/local` when configuring FLINT. If a library is built in its source tree, specify the top level source 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/local/  
            --with-mpfr=/home/user1/mpfr/
```

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

Note that FLINT builds static and shared libraries by default. If you do not require one of these then you may pass `--disable-static` or `--disable-shared` to `configure`. When running `make check` a shared library is required.

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.

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

```
make  
make check
```

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 and MPIR and MPFR with the options `-lflint -lmpfr -lgmp`.

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.

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
```

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

```
make -j4 check
```

§3. 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.

§4. 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 MPIR and MPFR as well as precise details of how to replicate the bug.

Note that FLINT needs to be linked against version 2.1.1 or later of MPIR, version 3.0.0 or later of MPFR and must be compiled with gcc version 2.96 or later.

§5. 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 main authors of FLINT to date have been William Hart, David Harvey (no longer active), Fredrik Johansson, Sebastian Pancratz and Andy Novocin.

Other significant contributions to FLINT have been made by Jason Papadopoulos, Gonzalo Tornaria, David Howden, Burcin Erocal, Tom Boothby, Daniel Woodhouse, Tomasz Lechowski, Richard Howell-Peak and Peter Shrimpton.

Additional research was contributed by Daniel Scott and Daniel Ellam.

Further patches and bug reports have been made by Michael Abshoff, Didier Deshommes, Craig Citro, Timothy Abbot, Carl Witty, Jaap Spies, Kiran Kedlaya, William Stein, Robert Bradshaw and many others.

Some code (`longlong.h` and `clz_tab.c`) has been used from an LGPL v2+ version of the GMP library. The main author of the GMP library is Torbjorn Granlund.

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

§6. 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:

`delta_qexp` Computes the first n terms of the delta function, e.g. `build/examples/delta_qexp 1000000` will compute the first one million terms 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.

§7. 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 `LONG_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 `unsigned long x`. If `x` is zero, returns 0.

§8. fmpz

Arbitrary precision integers

8.1 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 `long` 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.

8.2 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);
printf("^2□=□");
fmpz_print(y);
printf("\n");
fmpz_clear(x);
fmpz_clear(y);
```

The output is:

```
7^2 = 49
```

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

8.3 Memory management

```
void fmpz_init(fmpz_t f)
```

A small `fmpz_t` is initialised, i.e. just a `long`. 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 `long`. 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.

8.4 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, 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.

8.5 Conversion

```
ulong fmpz_get_si(const fmpz_t f)
```

Returns f as a signed long. The result is undefined if f does not fit into a long.

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

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

```
double fmpz_get_d_2exp(long * 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 $f = r * 2^{\text{exp}}$, 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, long val)
```

Sets f to the given signed long value.

```
void fmpz_set_ui(fmpz_t f, ulong val)
```

Sets f to the given unsigned long value.

```
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, 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.

8.6 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 MPFR.

```
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 MPFR.

```
int fmpz_print(fmpz_t x)
```

Prints the value x to `stdout`, without a carriage return. 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 `printf` from the standard library and `mpz_out_str` from MPFR.

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

Prints the value x to `file`, without a carriage return. 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 `printf` from the standard library and `mpz_out_str` from MPPIR.

8.7 Basic properties and manipulation

```
size_t mpz_sizeinbase(const mpz_t f, int b)
```

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

```
mp_bitcnt_t mpz_bits(const mpz_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 mpz_size(const mpz_t f)
```

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

```
int mpz_sgn(const mpz_t f)
```

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

```
void mpz_swap(mpz_t f, mpz_t g)
```

Efficiently swaps f and g . No data is copied.

```
void mpz_set(mpz_t f, const mpz_t g)
```

Sets f to the same value as g .

```
void mpz_zero(mpz_t f)
```

Sets f to zero.

8.8 Comparison

```
int mpz_cmp(const mpz_t f, const mpz_t g)
```

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

```
int mpz_cmp_ui(const mpz_t f, ulong g)
```

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

```
int mpz_cmpabs(const mpz_t f, const mpz_t g)
```

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

```
int mpz_equal(const mpz_t f, const mpz_t g)
```

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

```
int mpz_is_zero(const mpz_t f)
```

Returns 1 if f is 0, otherwise returns 0.

```
int mpz_is_one(const mpz_t f)
```

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

8.9 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, unsigned long x)
```

Sets f to $g + x$ where x is an unsigned long.

```
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, unsigned long x)
```

Sets f to $g - x$ where x is an unsigned long.

```
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, long x)
```

Sets f to $g \times x$ where x is a signed long.

```
void fmpz_mul_ui(fmpz_t f, const fmpz_t g, unsigned long x)
```

Sets f to $g \times x$ where x is an unsigned long.

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

Sets f to $g \times x \times y$ where x and y are of type unsigned long.

```
void fmpz_mul_2exp(fmpz_t f, const fmpz_t g, unsigned long 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, unsigned long x)
```

Sets f to $f + g \times x$ where x is an unsigned long.

```
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, unsigned long x)
```

Sets f to $f - g \times x$ where x is an unsigned long.

```
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, long 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(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, long 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_q_2exp(fmpz_t f, const fmpz_t g, ulong exp)
```

Sets f to g divided by 2^{exp} , but rounded down.

```
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_q_si(fmpz_t f, const fmpz_t g, long 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.

```
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, long 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.

```
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 unsigned long. 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 unsigned long, without changing g . If x is 0 an exception will result.

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

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

```
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.

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

Sets f to $n!$ where n is an unsigned long.

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

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

8.10 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.

8.11 Modular arithmetic

```
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.

8.12 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`.

8.13 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_unsigned(fmpz_t out, fmpz_t r1, fmpz_t m1,
                          ulong r2, ulong m2)
```

Uses the Chinese Remainder Theorem to compute the unique integer $0 \leq x < M$ congruent to r_1 modulo m_1 and r_2 modulo m_2 , where $M = m_1 \times m_2$. It is assumed that m_1 and m_2 are positive integers greater than 1 and coprime, and that $0 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$. The result x is stored in `out`.

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

Uses the Chinese Remainder Theorem to compute the unique integer $-M/2 \leq x < M/2$ congruent to r_1 modulo m_1 and r_2 modulo m_2 , where $M = m_1 \times m_2$. It is assumed that m_1 and m_2 are positive integers greater than 1 and coprime, and that $-m_1 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$. The result x is stored in `out`.

```
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, 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 a signed 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 $[-(N-1)/2, N/2]$. 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_limb_t * primes,
                   long 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.

§9. fmpz_vec

Vectors over \mathbf{Z}

9.1 Memory management

```
fmpz * _fmpz_vec_init(long len)
```

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

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

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

9.2 Randomisation

```
void _fmpz_vec_randtest(fmpz * f, flint_rand_t state, long  
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, long 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.

9.3 Bit sizes

```
long _fmpz_vec_max_bits(const fmpz * vec, long 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.

```
ulong _fmpz_vec_max_limbs(const fmpz * vec, long 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.

9.4 Input and output

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

Prints the vector of given length to `stdout`.

For further details, see `_fmpz_vec_fprint()`.

9.5 Assignment and basic manipulation

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

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

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

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

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

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

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

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

9.6 Comparison


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

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

9.7 Sorting

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

Sorts the coefficients of vec in ascending order.

9.8 Addition and subtraction

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

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

9.9 Scalar multiplication and division

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

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

```
void _fmpz_vec_scalar_mul_si(fmpz * vec1, const fmpz * vec2,
                             long len2, long c)
```

Sets $(vec1, len2)$ to $(vec2, len2)$ multiplied by c , where c is a signed long.

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

Sets $(vec1, len2)$ to $(vec2, len2)$ multiplied by c , where c is an unsigned long.

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

Sets $(vec1, len2)$ to $(vec2, len2)$ multiplied by 2^{exp} .

```
void _fmpz_vec_scalar_divexact_fmpz(fmpz * vec1, const fmpz
                                     * vec2, long 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, long len2, long 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, long 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, long len2, long 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, long 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, long 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_tdiv_q_fmpz(fmpz * vec1, const fmpz *
    vec2, long 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, long len2, long 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, long 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_addmul_fmpz(fmpz * vec1, const fmpz *
    vec2, long 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, long len2, long c)
```

Adds $(vec2, len2)$ times c to $(vec1, len2)$, where c is a signed long.

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

Adds $(vec2, len2)$ times $c * 2^{\text{exp}}$ to $(vec1, len2)$, where c is a signed long.

```
void _fmpz_vec_scalar_submul_fmpz(fmpz * vec1, const fmpz *
    vec2, long 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, long len2, long c)
```

Subtracts $(vec2, len2)$ times c from $(vec1, len2)$, where c is a signed long.

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

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

9.10 Gaussian content

```
void _fmpz_vec_content(fmpz_t res, const fmpz * vec, long
    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.

§10. fmpz_mat

Matrices over \mathbf{Z}

10.1 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 $0, 1, \dots, m - 1$ and columns of index $0, 1, \dots, n - 1$. 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.

10.2 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);
printf("A=\n");
fmpz_mat_print_pretty(A);
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]]
```

10.3 Memory management

```
void fmpz_mat_init(fmpz_mat_t mat, long rows, long 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.

10.4 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^{bits^2} . 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, long 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,
    long 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_randedet(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,
    long 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.

10.5 Basic assignment and manipulation

```
void fmpz_mat_set(fmpz_mat_t mat1, 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, 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, long i, long 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_unit(fmpz_mat_t mat)
```

Sets `mat` to the unit matrix, having 1:s on the main diagonal and 0:s elsewhere. If `mat` is nonsquare, it is set to the truncation of a unit matrix.

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

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

10.6 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()`.

10.7 Comparison

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

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

10.8 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.

10.9 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.

10.10 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, fmpz_mat_t A,
                             fmpz_mat_t B, long bits)
```

```
void fmpz_mat_mul_multi_mod(fmpz_mat_t C, fmpz_mat_t A,
    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.

10.11 Inverse

```
void fmpz_mat_inv(fmpz_mat_t B, fmpz_t d, const fmpz_mat_t
    A)
```

Sets `(B, d)` to the inverse matrix of the square matrix A , i.e. computes an integer matrix B and an integer d such that $AB = BA = dI$, the identity matrix.

If A is singular, d will be set to zero and the value of B will be undefined. In general, `(B, d)` will not be reduced to lowest terms. A and B are allowed to be the same object.

10.12 Determinant

```
void fmpz_mat_det(fmpz_t det, const fmpz_mat_t A)
```

Sets `det` to the determinant of the square matrix A . This function automatically chooses between `fmpz_mat_det_cofactor`, `fmpz_mat_det_bareiss` and `fmpz_mat_det_multi_mod` (with `proved = 1`), depending on the size of the matrix.

The matrix of dimension 0×0 is defined to have determinant 1.

```
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_multi_mod(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_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 .

10.13 Rank

```
long 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 .

10.14 Nonsingular solving

The following functions allow solving matrix-vector equations $Ax = b$ or 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: all functions compute an integer vector or matrix \hat{x} and a separate denominator d (`den`) such that $A(\hat{x}/d) = b$, or 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. 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.

```
void fmpz_mat_solve(fmpz * x, fmpz_t den, const fmpz_mat_t
    A, const fmpz * b)
```

Solves the matrix-vector equation $Ax = b$ where A is a nonsingular square matrix of size m and b is an integer vector of length m . This function calls `fmpz_mat_solve_cramer` for small matrices and `fmpz_mat_solve_fraction_free_LU` otherwise.

```
void fmpz_mat_solve_cramer(fmpz * x, fmpz_t den, const
    fmpz_mat_t A, const fmpz * b)
```

Solves the matrix-vector equation $Ax = b$ using Cramer's rule. Only systems of size up to 3×3 are allowed.

```
void fmpz_mat_solve_fraction_free_LU(fmpz * x, fmpz_t den,
    const fmpz_mat_t A, const fmpz * b)
```

Solves the matrix-vector equation $Ax = b$ by computing a fraction-free LU decomposition of A and then solving the triangular systems $Ly = b$, $Ux = y$ using fraction-free forward and backward substitution.

```
void fmpz_mat_solve_mat(fmpz_mat_t X, fmpz_t den, const
    fmpz_mat_t A, const fmpz_mat_t B)
```

Solves $AX = B$ for the m -by- n matrix X , where A is an m -by- m integer matrix and B is an m -by- n integer matrix. Equivalently, solves $Ax = b$ for each respective column vector x in X and b in B .

This function computes a fraction-free LU decomposition of A and reuses the LU decomposition to solve the column equations one by one.

```
void _fmpz_mat_solve_fraction_free_LU_precomp(fmpz * b,
    const fmpz_mat_t LU)
```

Solves the numerator part of the matrix-vector equation $Ax = b$, given a precomputed fraction-free LU decomposition of A . `b` is overwritten with the solution vector in-place. If the rows of the system matrix were permuted during the LU decomposition, replace `b[i]` with `b[perm[i]]` in the input.

10.15 Kernel

```
long fmpz_mat_kernel(fmpz_mat_t B, const fmpz_mat_t A)
```

Computes a basis for the right kernel (or null space) of A and returns the dimension of the kernel. B is set to 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).

10.16 Fraction-free row reduction

```
long _fmpz_mat_rowreduce(long * perm, fmpz_mat_t mat, int options)
```

Row reduces the matrix in-place using fraction-free Gaussian elimination. The number of rows m and columns n may be arbitrary.

This function effectively implements the algorithms FFGE, FFGJ and FFLU given in [?], but with pivoting. The `options` parameter is a bitfield which may be set to any combination of the following flags; use `options = 0` to disable all, resulting in FFLU:

- **ROWREDUCE_FAST_ABORT**
If set, the function immediately aborts and returns 0 when (if) the matrix is detected to be rank-deficient (singular). In this event, the state of the matrix will be undefined.
- **ROWREDUCE_FULL**
If set, performs fraction-free Gauss-Jordan elimination (FFGJ), i.e. eliminates the elements above each pivot element as well as those below. If not set, regular Gaussian elimination is performed and only the elements below pivots are eliminated.
- **ROWREDUCE_CLEAR_LOWER**
If set, clears (i.e. zeros) elements below the pivots (FFGE). If not set, the output becomes the fraction-free LU decomposition of the matrix with L in the lower triangular part.

Pivoting (to avoid division by zero entries) is performed by permuting the vector of row pointers in-place. The matrix entries themselves retain their original order in memory.

The permutation order will also 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 return value r is the rank of the matrix, multiplied by a sign indicating the parity of row interchanges. If $r = 0$, the matrix has rank zero, unless `ROWREDUCE_FAST_ABORT` is set, in which case $r = 0$ indicates any deficient rank. Otherwise, the leading nonzero entries of $a[0], a[1], \dots, a[|r| - 1]$ will point to the successive pivot elements. If $|r| = m = n$, the determinant of the matrix is given by $\text{sgn}(r) \times a[|r| - 1][|r| - 1]$.

§11. fmpz_poly

Polynomials over \mathbf{Z}

11.1 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.

11.2 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); printf("\n");
fmpz_poly_print(y); 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
```

11.3 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`.

11.4 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, long 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, long 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, long 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, long newlen)
```

Demotes the coefficients of `poly` beyond `newlen` and sets the length of `poly` to `newlen`.

11.5 Polynomial parameters

```
long fmpz_poly_length(const fmpz_poly_t poly)
```

Returns the length of `poly`. The zero polynomial has length zero.

```
long fmpz_poly_degree(const fmpz_poly_t poly)
```

Returns the degree of `poly`, which is one less than its length.

```
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.

```
long 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.

11.6 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, long 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, long 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, long
    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, long i, long j)
```

Sets the coefficients of x^i, \dots, 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, long
    len, long n)
```

Sets `(res, n)` to the reverse of `(poly, n)`, where `poly` is in fact an array of length `len`. Assumes that $0 < \text{len} \leq n$. 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, long 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, long 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.

11.7 Randomisation

```
void fmpz_poly_randtest(fmpz_poly_t f, flint_rand_t state,
    long 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, long 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, long 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.

11.8 Getting and setting coefficients

```
void fmpz_poly_get_coeff_fmpz(fmpz_t x, const fmpz_poly_t
    poly, long n)
```

Sets x to the n th 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.

```
long fmpz_poly_get_coeff_si(const fmpz_poly_t poly, long n)
```

Returns coefficient n of `poly` as a `long`. The result is undefined if the value does not fit into a `long`. 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, long 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, long
    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, long 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, long n, long
    x)
```

Sets coefficient n of `poly` to the `long` 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, long n, ulong
    x)
```

Sets coefficient n of `poly` to the `unsigned long` 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.

11.9 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.

11.10 Addition and subtraction

```
void _fmpz_poly_add(fmpz * res, const fmpz * poly1, long
    len1, const fmpz * poly2, long 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, long
    len1, const fmpz * poly2, long 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`.

11.11 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_si(fmpz_poly_t poly1, fmpz_poly_t
                             poly2, long x)
```

Sets `poly1` to `poly2` times the signed long `x`.

```
void fmpz_poly_scalar_mul_ui(fmpz_poly_t poly1, fmpz_poly_t
                              poly2, ulong x)
```

Sets `poly1` to `poly2` times the unsigned long `x`.

```
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_si(fmpz_poly_t poly1, fmpz_poly_t
                              poly2, long x)
```

Sets `poly1` to `poly2` divided by the long `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 unsigned long `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, long x)
```

Sets `poly1` to `poly2` divided by the long `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 unsigned long `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 coefficient is exact for every coefficient.

```
void fmpz_poly_scalar_divexact_si(fmpz_poly_t poly1,
    fmpz_poly_t poly2, long x)
```

Sets `poly1` to `poly2` divided by the long `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 unsigned long `x`, assuming the coefficient is exact for every coefficient.

11.12 Bit packing

```
void _fmpz_poly_bit_pack(mp_ptr arr, const fmpz * poly, long
    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`.

```
void _fmpz_poly_bit_unpack(fmpz * poly, long 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`.

```
void _fmpz_poly_bit_unpack_unsigned(fmpz * poly, long 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.

11.13 Multiplication

```
void _fmpz_poly_mul_classical(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long 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_mullassical(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long len2, long n)
```

Sets (res, n) to the first n coefficients of $(poly1, len1)$ multiplied by $(poly2, len2)$.

Assumes $0 < n \leq len1 + len2 - 1$. Assumes neither $len1$ nor $len2$ is zero.

```
void fmpz_poly_mullassical(fmpz_poly_t res, const
    fmpz_poly_t poly1, const fmpz_poly_t poly2, long n)
```

Sets res to the first n coefficients of $poly1 * poly2$.

```
void _fmpz_poly_mulhighclassical(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long len2, long
    start)
```

Sets the first $start$ coefficients of res to zero and the remainder to the corresponding coefficients of $(poly1, len1) * (poly2, len2)$.

Assumes $start \leq len1 + len2 - 1$. Assumes neither $len1$ nor $len2$ is zero.

```
void fmpz_poly_mulhighclassical(fmpz_poly_t res, const
    fmpz_poly_t poly1, const fmpz_poly_t poly2, long 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_mulmidclassical(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long 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 $len1 \geq len2 > 0$.

```
void fmpz_poly_mulmidclassical(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 $len1 \geq len2$.

```
void _fmpz_poly_mulkaratsuba(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long len2)
```

Sets $(res, len1 + len2 - 1)$ to the product of $(poly1, len1)$ and $(poly2, len2)$. Assumes $len1 \geq len2 > 0$. Allows zero-padding of the two input polynomials. No aliasing of inputs with outputs is allowed.

```
void fmpz_poly_mulkaratsuba(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_mullowkaratsuba_n(fmpz * res, const fmpz *
    poly1, const fmpz * poly2, long 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_mullowkaratsuba_n(fmpz_poly_t res, const
    fmpz_poly_t poly1, const fmpz_poly_t poly2, long 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, long 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, long 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, long
    len1, const fmpz * poly2, long 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_mul_KS(fmpz * res, const fmpz * poly1,
    long len1, const fmpz * poly2, long len2, long 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_mul_KS(fmpz_poly_t res, const fmpz_poly_t
    poly1, const fmpz_poly_t poly2, long n)
```

Sets `res` to the lowest `n` coefficients of the product of `poly1` and `poly2`.

```
void _fmpz_poly_mul(fmpz * res, const fmpz * poly1, long
    len1, const fmpz * poly2, long 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.

```
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_mul(fmpz * res, const fmpz * poly1, long
    len1, const fmpz * poly2, long len2, long 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, long 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, long 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.

11.14 Powering

```
void _fmpz_poly_pow_multinomial(fmpz * res, const fmpz *
    poly, long len, ulong e)
```

Computes `res = polye`. 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 = polye` using a generalisation of binomial expansion called the J.C.P. Miller pure recurrence [? ?]. 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 $P(x) = p_0 + p_1x + \dots + p_mx^m$ with $p_0 \neq 0$ and let

$$P(x)^n = a(n, 0) + a(n, 1)x + \dots + a(n, mn)x^{mn}.$$

Then $a(n, 0) = p_0^n$ and, for all $1 \leq k \leq mn$,

$$a(n, k) = (kp_0)^{-1} \sum_{i=1}^m p_i ((n+1)i - k) a(n, k - i).$$

```
void _fmpz_poly_pow_binomial(fmpz * res, const fmpz * poly,
    ulong e)
```

Computes `res = polye` 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 = polye` 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,
    long len, const int * a, int n)
```

Given a star chain $1 = a_0 < a_1 < \dots < a_n = e$ computes `res = polye`.

A star chain is an addition chain $1 = a_0 < a_1 < \dots < a_n$ 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 $\text{res} = \text{poly}^e$ 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,
    long len, ulong e)
```

Sets $\text{res} = \text{poly}^e$ using left-to-right binary exponentiation as described in [?, p. 461].

Assumes that $\text{len} > 0$, $e > 1$. Assumes that res is an array of length at least $e * (\text{len} - 1) + 1$. Does not support aliasing.

```
void fmpz_poly_pow_binexp(fmpz_poly_t res, const fmpz_poly_t
    poly, ulong e)
```

Computes $\text{res} = \text{poly}^e$ 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,
    long len, ulong e)
```

Sets $\text{res} = \text{poly}^e$ whenever $0 \leq e \leq 4$.

Assumes that $\text{len} > 0$ and that res is an array of length at least $e * (\text{len} - 1) + 1$. Does not support aliasing.

```
void _fmpz_poly_pow(fmpz * res, const fmpz * poly, long len,
    ulong e)
```

Sets $\text{res} = \text{poly}^e$, assuming that e , $\text{len} > 0$ and that res has space for $e * (\text{len} - 1) + 1$ coefficients. Does not support aliasing.

```
void fmpz_poly_pow(fmpz_poly_t res, const fmpz_poly_t poly,
    ulong e)
```

Computes $\text{res} = \text{poly}^e$. 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, long 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, long 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.

11.15 Shifting

```
void _fmpz_poly_shift_left(fmpz * res, const fmpz * poly,
    long len, long 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, long n)
```

Sets `res` to `poly` shifted left by n coeffs. Zero coefficients are inserted.

```
void _fmpz_poly_shift_right(fmpz * res, const fmpz * poly,
    long len, long 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, long 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.

11.16 Norms

```
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`.

11.17 Greatest common divisor

```
void _fmpz_poly_gcd_subresultant(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long 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 [?, Algorithm 3.3.1].

```
void _fmpz_poly_gcd(fmpz * res, const fmpz * poly1, long
    len1, const fmpz * poly2, long 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` might not be 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_resultant(fmpz_t res, const fmpz * poly1,
    long len1, const fmpz * poly2, long 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 $f(x) = a_m x^m + \dots + a_0$ and $g(x) = b_n x^n + \dots + b_0$ 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 [?, Algorithm 3.3.7].

11.18 Gaussian content

```
void _fmpz_poly_content(fmpz_t res, const fmpz * poly, long
    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, long 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.

11.19 Euclidean division

```
void _fmpz_poly_divrem_basecase(fmpz * Q, fmpz * R, const
    fmpz * A, long lenA, const fmpz * B, long lenB)
```

Computes $(Q, \text{lenA} - \text{lenB} + 1)$, (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 $\text{len}(A), \text{len}(B) > 0$. Allows zero-padding in $(A, \text{len}A)$. 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 $\text{len}(B) - 1$ 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, long lenB)
```

Computes $(Q, \text{len}B)$, $(BQ, 2 \text{len}B - 1)$ such that $BQ = B \times Q$ and $A = BQ + R$ where each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B . We assume that $\text{len}(A) = 2 \text{len}(B) - 1$. If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over \mathbf{Q} .

Assumes $\text{len}(B) > 0$. Allows zero-padding in $(A, \text{len}A)$. Requires a temporary array $(W, 2 \text{len}B - 1)$. 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_poly_divrem_divconquer(fmpz * Q, fmpz * R, const
    fmpz * A, long lenB, const fmpz * B, long lenB)
```

Computes $(Q, \text{len}A - \text{len}B + 1)$, $(R, \text{len}A)$ such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ 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 $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in $(A, \text{len}A)$. 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 $\text{len}(B) - 1$ 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,
    long lenA, const fmpz * B, long lenB)
```

Computes $(Q, \text{len}A - \text{len}B + 1)$, $(R, \text{len}A)$ such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ 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 $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in $(A, \text{len}A)$. 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 $\text{len}(B) - 1$ 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, long lenA, const fmpz * B, long lenB)
```

Computes the quotient $(Q, \text{lenA} - \text{lenB} + 1)$ of (A, lenA) divided by (B, lenB) .

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ 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 $\text{len}(A), \text{len}(B) > 0$. 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 $\text{len}(B) - 1$ 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_low_divconquer_recursive(fmpz * Q,
    fmpz * BQ, const fmpz * A, const fmpz * B, long lenB)
```

Divide and conquer division of $(A, 2 \text{lenB} - 1)$ by (B, lenB) , computing only the bottom $\text{len}(B) - 1$ coefficients of BQ .

Assumes $\text{len}(B) > 0$. Requires BQ to have length at least $2 \text{len}(B) - 1$, although only the bottom $\text{len}(B) - 1$ 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, long lenB)
```

Recursive short division in the balanced case.

Computes the quotient (Q, lenB) of $(A, 2 \text{lenB} - 1)$ upon division by (B, lenB) . Requires $\text{len}(B) > 0$. Needs a temporary array `temp` of length $2 \text{len}(B) - 1$. Does not support any aliasing.

For further details, see [?].

```
void _fmpz_poly_div_divconquer(fmpz * Q, const fmpz * A,
    long lenA, const fmpz * B, long lenB)
```

Computes the quotient $(Q, \text{lenA} - \text{lenB} + 1)$ of (A, lenA) upon division by (B, lenB) . Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. 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 $\text{len}(B) - 1$ 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, long lenA,
    const fmpz * B, long lenB)
```

Computes the quotient $(Q, \text{lenA} - \text{lenB} + 1)$ of (A, lenA) divided by (B, lenB) .

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ 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 $\text{len}(A) \geq \text{len}(B) > 0$. 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 $\text{len}(B) - 1$ 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_basecase(fmpz * R, const fmpz * A, long
    lenA, const fmpz * B, long 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 $\text{len}(B) - 1$ 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 $\text{len}(A), \text{len}(B) > 0$. 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_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 $\text{len}(B) - 1$ 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, long lenA,
    const fmpz * B, long 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 $\text{len}(B) - 1$ 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 $\text{len}(A) \geq \text{len}(B) > 0$. 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 $\text{len}(B) - 1$ 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.

11.20 Power series division

```
void _fmpz_poly_inv_series_newton(fmpz * Qinvt, const fmpz *
    Q, long 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.

```
void fmpz_poly_inv_series_newton(fmpz_poly_t Qinvt, const
    fmpz_poly_t Q, long 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 * Qinvt, const fmpz * Q, long
    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 Qinvt, const
    fmpz_poly_t Q, long 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, long 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 $n \geq 1$.

11.21 Pseudo division

```
void _fmpz_poly_pseudo_divrem_basecase(fmpz * Q, fmpz * R,
    ulong * d, const fmpz * A, long lenA, const fmpz * B,
    long lenB)
```

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 $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that Q can fit $\text{len}(A) - \text{len}(B) + 1$ coefficients, and that R can fit $\text{len}(A)$ 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_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, long lenB, const fmpz * B,
    long lenA)
```

Computes $(Q, \text{lenA} - \text{lenB} + 1)$, (R, lenA) such that $\ell^d A = BQ + R$, only setting the bottom $\text{len}(B) - 1$ coefficients of R to their correct values. The remaining top coefficients of (R, lenA) may be arbitrary.

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 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, long lenA, const fmpz * B, long lenB)
```

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that Q can fit $\text{len}(A) - \text{len}(B) + 1$ coefficients, and that R can fit $\text{len}(A)$ 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 $\max\{0, \text{len}(A) - \text{len}(B) + 1\}$.

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 $\text{len}(B) < 32$.

```
void _fmpz_poly_pseudo_rem_cohen(fmpz * R, const fmpz * A,
    long lenA, const fmpz * B, long lenB)
```

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that R can fit $\text{len}(A)$ 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 $\max\{0, \text{len}(A) - \text{len}(B) + 1\}$.

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 $\text{len}(B) < 32$.

This function uses the algorithm described in [?, Algorithm 3.1.2].

```
void _fmpz_poly_pseudo_divrem(fmpz * Q, fmpz * R, ulong * d,
    const fmpz * A, long lenA, const fmpz * B, long lenB)
```

If ℓ is the leading coefficient of B , then computes $(Q, \text{lenA} - \text{lenB} + 1)$, $(R, \text{lenB} - 1)$ and d such that $\ell^d A = BQ + R$. This function is used for simulating division over \mathbf{Q} .

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that Q can fit $\text{len}(A) - \text{len}(B) + 1$ coefficients, and that R can fit $\text{len}(A)$ coefficients, although on exit only the bottom $\text{len}(B)$ 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 supported.

```
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, long lenA, const fmpz * B, long lenB)
```

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, long lenA, const fmpz * B, long lenB)
```

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.

11.22 Derivative

```
void _fmpz_poly_derivative(fmpz * rpoly, const fmpz * poly,
    long len)
```

Sets $(\text{rpoly}, \text{len} - 1)$ to the derivative of $(\text{poly}, \text{len})$. 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 .

11.23 Evaluation

```
void _fmpz_poly_evaluate_divconquer_fmpz(fmpz_t res, const
    fmpz * poly, long len, const fmpz_t a)
```

Evaluates the polynomial $(\text{poly}, \text{len})$ 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, long 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,
    long 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, long len, const fmpz_t anum, const
    fmpz_t aden)
```

Evaluates the polynomial (f, len) at the rational $(\text{anum}, \text{aden})$ using Horner's rule, and sets $(\text{rnum}, \text{rden})$ to the result in lowest terms.

Aliasing between $(\text{rnum}, \text{rden})$ and $(\text{anum}, \text{aden})$ 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, long len, const fmpz_t anum, const fmpz_t
    aden)
```

Evaluates the polynomial (f, len) at the rational $(\text{anum}, \text{aden})$ and sets $(\text{rnum}, \text{rden})$ to the result in lowest terms.

Aliasing between $(\text{rnum}, \text{rden})$ and $(\text{anum}, \text{aden})$ 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, long
    len, mp_limb_t a, mp_limb_t n, mp_limb_t ninv)
```

Evaluates $(\text{poly}, \text{len})$ 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.

11.24 Composition

```
void _fmpz_poly_compose_horner(fmpz * res, const fmpz *
    poly1, long len1, const fmpz * poly2, long len2)
```

Sets `res` to the composition of $(\text{poly1}, \text{len1})$ and $(\text{poly2}, \text{len2})$.

Assumes that `res` has space for $(\text{len1}-1)*(\text{len2}-1)+ 1$ 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, long len1, const fmpz * poly2, long len2)
```

Computes the composition of $(\text{poly1}, \text{len1})$ and $(\text{poly2}, \text{len2})$ using a divide and conquer approach and places the result into `res`, assuming `res` can hold the output of length $(\text{len1} - 1)* (\text{len2} - 1)+ 1$.

Assumes $\text{len1}, \text{len2} > 0$. Does not support aliasing between `res` and any of $(\text{poly1}, \text{len1})$ and $(\text{poly2}, \text{len2})$.

```
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, long
    len1, const fmpz * poly2, long len2)
```

Sets `res` to the composition of $(\text{poly1}, \text{len1})$ and $(\text{poly2}, \text{len2})$.

Assumes that `res` has space for $(\text{len1}-1)*(\text{len2}-1)+ 1$ 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 $f(t) = g(h(t))$.

11.25 Signature

```
void _fmpz_poly_signature(long * r1, long * r2, fmpz * poly,
    long 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(long * r1, long * r2, 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 [?, Algorithm 4.1.11].

11.26 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 -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 caret `^` 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, long 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, long 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, long
    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,
    long 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.

§12. fmpq_poly

Polynomials over \mathbf{Q}

12.1 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 `long`, 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.

12.2 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, long 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, long 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, long 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, long 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, long 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, long 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.

12.3 Polynomial parameters

```
long fmpq_poly_degree(fmpq_poly_t poly)
```

Returns the degree of `poly`, which is one less than its length, as a long.

```
long fmpq_poly_length(fmpq_poly_t poly)
```

Returns the length of `poly`.

12.4 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`.

12.5 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 `LONG_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,
    long 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, long 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, long 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.

12.6 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, long 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_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_array_mpq(fmpz * poly, fmpz_t den, const
                              mpq_t * a, long 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, long 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_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, long 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.

12.7 Setting and getting coefficients

```
void fmpq_poly_get_coeff_mpq(mpq_t x, const fmpq_poly_t
    poly, long n)
```

Retrieves the `n`th coefficient of `poly`, in lowest terms.

```
void fmpq_poly_set_coeff_si(fmpq_poly_t poly, long n, long
    x)
```

Sets the `n`th coefficient in `poly` to the integer `x`.

```
void fmpq_poly_set_coeff_ui(fmpq_poly_t poly, long n, ulong
    x)
```

Sets the `n`th coefficient in `poly` to the integer `x`.

```
void fmpq_poly_set_coeff_fmpz(fmpq_poly_t poly, long n,
    const fmpz_t x)
```

Sets the `n`th coefficient in `poly` to the integer `x`.

```
void fmpq_poly_set_coeff_mpz(fmpq_poly_t rop, long n, const
    mpz_t x)
```

Sets the `n`th coefficient in `poly` to the integer `x`.

```
void fmpq_poly_set_coeff_mpq(fmpq_poly_t rop, long n, const
    mpq_t x)
```

Sets the n th coefficient in `poly` to the rational x , which is expected to be provided in lowest terms.

12.8 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, long 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.

12.9 Addition and subtraction

```
void _fmpq_poly_add(fmpz * rpoly, fmpz_t rden, const fmpz *
    poly1, const fmpz_t den1, long len1, const fmpz * poly2,
    const fmpz_t den2, long 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(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_sub(fmpz * rpoly, fmpz_t rden, const fmpz *
    poly1, const fmpz_t den1, long len1, const fmpz * poly2,
    const fmpz_t den2, long 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(fmpq_poly_t res, fmpq_poly poly1,
                  fmpq_poly poly2)
```

Sets `res` to the difference of `poly1` and `poly2`, using Henrici's algorithm.

12.10 Scalar multiplication and division

```
void _fmpq_poly_scalar_mul_si(fmpz * rpoly, fmpz_t rden,
                              const fmpz * poly, const fmpz_t den, long len, long 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, long 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, long 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_mpq(fmpz * rpoly, fmpz_t rden,
                              const fmpz * poly, const fmpz_t den, long 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, long 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_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 mpq_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, long 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, long len, long 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, long 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_mpq(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long 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, long 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_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)
```

12.11 Multiplication

```
void _fmpq_poly_mul(fmpz * rpoly, fmpz_t rden, const fmpz *
    poly1, const fmpz_t den1, long len1, const fmpz * poly2,
    const fmpz_t den2, long 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, long len1, const fmpz *
    poly2, const fmpz_t den2, long len2, long 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 $0 < n \leq \text{len1} + \text{len2} - 1$. 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, long n)
```

Sets res to the product of poly1 and poly2, truncated to length n .

12.12 Powering

```
void _fmpq_poly_pow(fmpz * rpoly, fmpz_t rden, const fmpz *
    poly, const fmpz_t den, long len, ulong e)
```

Sets (rpoly, rden) to (poly, den)^e, assuming e, len > 0. Assumes that rpoly is an array of length at least $e * (\text{len} - 1) + 1$. 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 0⁰ is defined as 1.

12.13 Shifting

```
void fmpz_poly_shift_left(fmpz_poly_t res, const fmpz_poly_t
    poly, long 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, long 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.

12.14 Euclidean division

```
void _fmpq_poly_divrem(fmpz * Q, fmpz_t q, fmpz * R, fmpz_t
    r, const fmpz * A, const fmpz_t a, long lenA, const fmpz
    * B, const fmpz_t b, long lenB)
```

Finds the quotient (*Q*, *q*) and remainder (*R*, *r*) of the Euclidean division of (*A*, *a*) by (*B*, *b*).

Assumes that $\text{lenA} \geq \text{lenB} > 0$. Assumes that *R* has space for *lenA* coefficients, although only the bottom $\text{lenB} - 1$ will carry meaningful data on exit. Supports no aliasing between the two outputs, or between the inputs and the outputs.

```
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, long lenA, const fmpz * B, const fmpz_t
    b, long lenB)
```

Finds the quotient (*Q*, *q*) of the Euclidean division of (*A*, *a*) by (*B*, *b*).

Assumes that $\text{lenA} \geq \text{lenB} > 0$. Supports no aliasing between the inputs and the outputs.

```
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, long lenA, const fmpz * B, const fmpz_t
    b, long lenB)
```

Finds the remainder (*R*, *r*) of the Euclidean division of (*A*, *a*) by (*B*, *b*).

Assumes that $\text{lenA} \geq \text{lenB} > 0$. Supports no aliasing between the inputs and the outputs.

```
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*.

12.15 Power series division

```
void _fmpq_poly_inv_series_newton(fmpz * rpoly, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long n)
```

Computes the first *n* terms of the inverse power series of *poly* using Newton iteration.

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, long 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, long 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, long 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, long 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, long 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 $n \geq 1$.

12.16 Derivative and integral

```
void _fmpq_poly_derivative(fmpz * rpoly, fmpz_t rden, const
    fmpz * poly, const fmpz_t den, long len)
```

Sets $(\text{rpoly}, \text{rden}, \text{len} - 1)$ to the derivative of $(\text{poly}, \text{den}, \text{len})$. Does nothing if $\text{len} \leq 1$. 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, long len)
```

Sets $(\text{rpoly}, \text{rden}, \text{len})$ to the integral of $(\text{poly}, \text{den}, \text{len} - 1)$. Assumes $\text{len} \geq 0$. 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.

12.17 Square roots

```
void _fmpq_poly_sqrt_series(fmpz * g, fmpz_t gden, const
    fmpz * f, const fmpz_t fden, long 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, long 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, long 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, long 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.

12.18 Transcendental functions

```
void _fmpq_poly_log_series(fmpz * g, fmpz_t gden, const fmpz
    * f, const fmpz_t fden, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long 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, long n)
```

Sets `res` to the series expansion of the hyperbolic tangent of `f` to order `n > 0`. Requires `f` to have constant term 0.

12.19 Evaluation

```
void _fmpq_poly_evaluate_fmpz(fmpz_t rnum, fmpz_t rden,
    const fmpz * poly, const fmpz_t den, long 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(mpq_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_mpq(fmpz_t rnum, fmpz_t rden, const
    fmpz * poly, const fmpz_t den, long 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_mpq(mpq_t res, const fmpq_poly_t
    poly, const mpq_t a)
```

Evaluates the polynomial `poly` at the rational `a` and sets `res` to the result.

12.20 Composition

```
void _fmpq_poly_compose(fmpz * res, fmpz_t den, const fmpz *
    poly1, const fmpz_t den1, long len1, const fmpz * poly2,
    const fmpz_t den2, long 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, long 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 mpq_t a)
```

Sets `res` to `poly` with the indeterminate rescaled by a .

12.21 Gaussian content

```
void _fmpq_poly_content(mpq_t res, const fmpz * poly, const
    fmpz_t den, long len)
```

Sets `res` to the content of `(poly, den, len)`. If `len == 0`, sets `res` to zero.

```
void fmpq_poly_content(mpq_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, long 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,
    long 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, long 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.

12.22 Square-free

```
int _fmpq_poly_is_squarefree(const fmpz * poly, const fmpz_t
    den, long len)
```

Returns whether the polynomial `(poly, den, len)` is 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.

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.

12.23 Input and output

```
int _fmpz_poly_print(const fmpz * poly, const fmpz_t den,
                    long 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 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_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 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, const
                     fmpz_t den, long 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 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_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_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.

§13. nmod_vec

Vectors over $\mathbf{Z}/n\mathbf{Z}$ for word-sized
moduli

13.1 Memory management

`mp_ptr _nmod_vec_init(long len)`

Returns a vector of the given length. The entries are not necessarily zero.

`void _nmod_vec_free(mp_ptr vec)`

Frees the memory used by the given vector.

13.2 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.

13.3 Random functions

```
void _nmod_vec_randtest(mp_ptr vec, flint_rand_t state, long
    len, nmod_t mod)
```

Sets `vec` to a random vector of the given length with entries reduced modulo mod.n .

13.4 Basic manipulation and comparison

```
void _nmod_vec_set(mp_ptr res, mp_srcptr vec, long len)
```

Copies `len` entries from the vector `vec` to `res`.

```
void _nmod_vec_zero(mp_ptr vec, long len)
```

Zeros the given vector of the given length.

```
void _nmod_vec_swap(mp_ptr a, mp_ptr b, long 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, long 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, long len)
```

Returns the maximum number of bits of any entry in the vector.

```
int _nmod_vec_equal(mp_ptr vec, mp_srcptr vec2, long len)
```

Returns 1 if (vec, len) is equal to $(\text{vec2}, \text{len})$, otherwise returns 0.

13.5 Arithmetic operations

```
void _nmod_vec_add(mp_ptr res, mp_srcptr vec1,
                  mp_srcptr vec2, long len, nmod_t mod)
```

Sets (res, len) to the sum of $(\text{vec1}, \text{len})$ and $(\text{vec2}, \text{len})$.

```
void _nmod_vec_sub(mp_ptr res, mp_srcptr vec1,
                  mp_srcptr vec2, long len, nmod_t mod)
```

Sets (res, len) to the difference of $(\text{vec1}, \text{len})$ and $(\text{vec2}, \text{len})$.

```
void _nmod_vec_neg(mp_ptr res, mp_srcptr vec, long 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,
                               long 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,
                                  long len, mp_limb_t c,
                                  nmod_t mod)
```

Adds (vec, len) times c to the vector (res, len) .

§14. nmod_poly

Polynomials over $\mathbf{Z}/n\mathbf{Z}$ for
word-sized moduli

14.1 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, long  
    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, long 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, long 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, long 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.

14.2 Polynomial properties

```
long nmod_poly_length(const nmod_poly_t poly)
```

Returns the length of the polynomial `poly`. The zero polynomial has length zero.

```
long 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`.

14.3 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, long 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, long len, long 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, long 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.

14.4 Randomisation

```
void nmod_poly_randtest(nmod_poly_t poly, flint_rand_t state, long len)
```

Generates a random polynomial with up to the given length.

14.5 Getting and setting coefficients

```
ulong nmod_poly_get_coeff_ui(const nmod_poly_t poly, ulong
    j)
```

Returns the coefficient of `poly` at index `j`, where coefficients are numbered with zero being the constant coefficient, and returns it as an `unsigned long`. If `j` refers to a coefficient beyond the end of `poly`, zero is returned.

```
void nmod_poly_set_coeff_ui(nmod_poly_t poly, ulong 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.

14.6 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 `free()`.

```
int nmod_poly_set_str(const char * s, nmod_poly_t poly)
```

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.

14.7 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.

14.8 Shifting

```
void _nmod_poly_shift_left(mp_ptr res, mp_srcptr poly, long
    len, long 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, long 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, long
    len, long k)
```

Sets `(res, len - k)` to `(poly, len)` shifted left by `k` coefficients. It is assumed that $k \leq \text{len}$ 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, long 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.

14.9 Addition and subtraction

```
void _nmod_poly_add(mp_ptr res, mp_srcptr poly1, long len1,
    mp_srcptr poly2, long 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, long len1,
    mp_srcptr poly2, long 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`.

14.10 Scalar multiplication and division

```
void nmod_poly_scalar_mul_nmod(nmod_poly_t res, const
                               nmod_poly_t poly, ulong c)
```

Sets `res` to $(\text{poly}, \text{len})$ multiplied by c , where c is reduced modulo the modulus of `poly`.

```
void _nmod_poly_make_monic(mp_ptr output, mp_srcptr input,
                          long 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`.

14.11 Bit packing and unpacking

```
void _nmod_poly_bit_pack(mp_ptr res, mp_srcptr poly, long
                        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` $<$ $3 * \text{FLINT_BITS}$.

```
void _nmod_poly_bit_unpack(mp_ptr res, long 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 $3 * \text{FLINT_BITS} > \text{bits} >$ 0. There are no restrictions on the size of the actual coefficients as stored within the bitfields.

14.12 Multiplication

```
void _nmod_poly_mul_classical(mp_ptr res, mp_srcptr poly1,
                              long len1, mp_srcptr poly2, long len2, nmod_t mod)
```

Sets $(\text{res}, \text{len1} + \text{len2} - 1)$ to the product of $(\text{poly1}, \text{len1})$ and $(\text{poly2}, \text{len2})$. Assumes `len1` \geq `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, long len1, mp_srcptr poly2, long len2, long trunc,
    nmod_t mod)
```

Sets *res* to the lower *trunc* coefficients of the product of (*poly1*, *len1*) and (*poly2*, *len2*). Assumes that $\text{len1} \geq \text{len2} > 0$ and $\text{trunc} > 0$. 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, long 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, long len1, mp_srcptr poly2, long len2, long 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 $\text{len1} \geq \text{len2} > 0$. 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, long 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, long len1,
    mp_srcptr in2, long len2, mp_bitcnt_t bits, nmod_t mod)
```

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. Assumes that $\text{len1} \geq \text{len2} > 0$.

```
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_mullow_KS(mp_ptr out, mp_srcptr in1, long
    len1, mp_srcptr in2, long len2, mp_bitcnt_t bits, long 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 $\text{len1} \geq \text{len2} > 0$ and that $0 < n \leq \text{len1} + \text{len2} - 1$.

```
void nmod_poly_mullow_KS(nmod_poly_t res, const nmod_poly_t
    poly1, const nmod_poly_t poly2, mp_bitcnt_t bits, long 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, long len1,
                   mp_srcptr poly2, long 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_mulow(mp_ptr res, mp_srcptr poly1, long
                     len1, mp_srcptr poly2, long len2, long 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 $0 < n \leq \text{len1} + \text{len2} - 1$ and that `len1 >= len2 > 0`. No aliasing of inputs and output is permitted.

```
void nmod_poly_mulow(nmod_poly_t res, const nmod_poly_t
                    poly1, const nmod_poly_t poly2, long 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, long
                       len1, mp_srcptr poly2, long len2, long 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 $0 < n \leq \text{len1} + \text{len2} - 1$. 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, long 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.

14.13 Powering

```
void _nmod_poly_pow_binexp(mp_ptr res, mp_srcptr poly, long
                           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 $(\text{len} - 1) * e + 1$ 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, long 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 $(\text{len} - 1) * e + 1$ 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, long 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, long 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, long 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, long 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.

14.14 Division

```
void _nmod_poly_divrem_basecase(mp_ptr Q, mp_ptr R, mp_ptr
    W, mp_srcptr A, long A_len, mp_srcptr B, long B_len,
    nmod_t mod)
```

Finds Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. If $\text{len}(B) = 0$ 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 $\text{len}(R) < \text{len}(B)$. If $\text{len}(B) = 0$ an exception is raised.

```
void _nmod_poly_div_basecase(mp_ptr Q, mp_ptr W, mp_srcptr
    A, long A_len, mp_srcptr B, long B_len, nmod_t mod)
```

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. 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 $\text{len}(R) < \text{len}(B)$, but returns only Q . If $\text{len}(B) = 0$ 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, long
      lenB, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $2 * \text{len}B - 1$ and B is of length $\text{len}B$. Sets BQ to the low $\text{len}B - 1$ coefficients of $B * Q$. We require that Q have space for $\text{len}B$ coefficients, that W be temporary space of size $\text{len}B - 1$ 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, long lenA, mp_srcptr B, long lenB, nmod_t
      mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$. We require that Q have space for $\text{len}A - \text{len}B + 1$ 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 $\text{len}(R) < \text{len}(B)$.

```
void _nmod_poly_divrem(mp_ptr Q, mp_ptr R, mp_srcptr A, long
      lenA, mp_srcptr B, long lenB, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$. We require that Q have space for $\text{len}A - \text{len}B + 1$ 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 $\text{len}(R) < \text{len}(B)$.

```
void _nmod_poly_div_divconquer_recursive(mp_ptr Q, mp_ptr W,
      mp_ptr V, mp_srcptr A, mp_srcptr B, long lenB, nmod_t
      mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $2 * \text{len}B - 1$ and B is of length $\text{len}B$. We require that Q have space for $\text{len}B$ coefficients and that W be temporary space of size $\text{len}B - 1$ 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, long
      lenA, mp_srcptr B, long lenB, nmod_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$, but returns only Q . We require that Q have space for $\text{len}A - \text{len}B + 1$ 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 $\text{len}(R) < \text{len}(B)$, but returns only Q .

```
void _nmod_poly_div(mp_ptr Q, mp_srcptr A, long lenA,
      mp_srcptr B, long lenB, nmod_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$, but returns only Q . We require that Q have space for $\text{len}A - \text{len}B + 1$ coefficients.

```
void nmod_poly_div(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 $\text{len}(R) < \text{len}(B)$, but returns only Q .

```
void _nmod_poly_inv_series_basecase(mp_ptr Qinvt, mp_srcptr
    Q, long n, nmod_t mod)
```

Given Q of length n whose leading coefficient is invertible modulo the given modulus, finds a polynomial Q_{inv} of length n such that the top n coefficients of the product $Q * Q_{\text{inv}}$ 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 Qinvt, const
    nmod_poly_t Q, long n)
```

Given Q of length at least n find Q_{inv} of length n such that the top n coefficients of the product $Q * Q_{\text{inv}}$ 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 Qinvt, mp_srcptr Q,
    long n, nmod_t mod)
```

Given Q of length n whose constant coefficient is invertible modulo the given modulus, find a polynomial Q_{inv} of length n such that $Q * Q_{\text{inv}}$ 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 Qinvt, const
    nmod_poly_t Q, long n)
```

Given Q find Q_{inv} such that $Q * Q_{\text{inv}}$ 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 Qinvt, mp_srcptr Q, long n,
    nmod_t mod)
```

Given Q of length n whose constant coefficient is invertible modulo the given modulus, find a polynomial Q_{inv} of length n such that $Q * Q_{\text{inv}}$ 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 Qinvt, const
    nmod_poly_t Q, long n)
```

Given Q find Q_{inv} such that $Q * Q_{\text{inv}}$ 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, long 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, long 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, long Alen,
    mp_srcptr B, long Blen, nmod_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$, but return only Q . We require that Q have space for $\text{len}A - \text{len}B + 1$ coefficients. 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 $\text{len}(R) < \text{len}(B)$, but returns only Q . 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, long Alen, mp_srcptr B, long Blen, nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$. We require that Q have space for $\text{len}A - \text{len}B + 1$ 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 $\text{len}(R) < \text{len}(B)$. The algorithm used is to call `div_newton()` and then multiply out and compute the remainder.

14.15 Derivative and integral

```
void _nmod_poly_derivative(mp_ptr x_prime, mp_srcptr x, long
    len, nmod_t mod)
```

Sets the first $\text{len} - 1$ coefficients of x_prime to the derivative of x which is assumed to be of length len . It is assumed that $\text{len} > 0$.

```
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, long
    len, nmod_t mod)
```

Set the first len coefficients of x_int to the integral of x which is assumed to be of length $\text{len} - 1$. The constant term of x_int is set to zero. It is assumed that $\text{len} > 0$. 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`.

14.16 Evaluation

```
mp_limb_t _nmod_poly_evaluate_nmod(mp_srcptr poly, long 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.

14.17 Composition

```
void _nmod_poly_compose_horner(mp_ptr res, mp_srcptr poly1,
    long len1, mp_srcptr poly2, long 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, long len1, mp_srcptr poly2, long 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, long
    len1, mp_srcptr poly2, long 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`.

14.18 GCD

```
long _nmod_poly_gcd_euclidean(mp_ptr G, mp_srcptr A, long
    lenA, mp_srcptr B, long lenB, nmod_t mod)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA` \geq `lenB` $>$ 1. 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.

14.19 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, long
    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, long 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, long 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, long n)
```

Set g to the series expansion of \sqrt{h} to order $O(x^n)$. It is assumed that h has constant term 1.

14.20 Transcendental functions

The elementary transcendental functions of a formal power series h are defined as

$$\begin{aligned}\exp(h(x)) &= \sum_{k=0}^{\infty} \frac{(h(x))^k}{k!} \\ \log(h(x)) &= \int_0^x \frac{h'(t)}{h(t)} dt \\ \operatorname{atan}(h(x)) &= \int_0^x \frac{h'(t)}{1+(h(t))^2} dt \\ \operatorname{atanh}(h(x)) &= \int_0^x \frac{h'(t)}{1-(h(t))^2} dt \\ \operatorname{asin}(h(x)) &= \int_0^x \frac{h'(t)}{\sqrt{1-(h(t))^2}} dt \\ \operatorname{asinh}(h(x)) &= \int_0^x \frac{h'(t)}{\sqrt{1+(h(t))^2}} dt\end{aligned}$$

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, long 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, long 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, long 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, long 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, long 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, long 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,
    long hlen, long 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, long 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, long 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 [?]). For small n , falls back to the basecase algorithm.

```
void nmod_poly_exp_series(nmod_poly_t g, const nmod_poly_t
    h, long 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, long 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, long n)
```

Set $g = \operatorname{atan}(h) + O(x^n)$.

```
void _nmod_poly_atanh_series(mp_ptr g, mp_srcptr h, long 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, long n)
```

Set $g = \operatorname{atanh}(h) + O(x^n)$.

```
void _nmod_poly_asin_series(mp_ptr g, mp_srcptr h, long 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, long n)
```

Set $g = \operatorname{asin}(h) + O(x^n)$.

```
void _nmod_poly_asinh_series(mp_ptr g, mp_srcptr h, long 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, long n)
```

Set $g = \operatorname{asinh}(h) + O(x^n)$.

```
void _nmod_poly_sin_series(mp_ptr g, mp_srcptr h, long 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, long n)
```

Set $g = \sin(h) + O(x^n)$.

```
void _nmod_poly_cos_series(mp_ptr g, mp_srcptr h, long 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, long n)
```

Set $g = \cos(h) + O(x^n)$.

```
void _nmod_poly_tan_series(mp_ptr g, mp_srcptr h, long 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, long n)
```

Set $g = \tan(h) + O(x^n)$.

```
void _nmod_poly_sinh_series(mp_ptr g, mp_srcptr h, long 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, long n)
```

Set $g = \sinh(h) + O(x^n)$.

```
void _nmod_poly_cosh_series(mp_ptr g, mp_srcptr h, long n,
    nmod_t mod)
```

Set $g = \cosh(h) + O(x^n)$. 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, long n)
```

Set $g = \cosh(h) + O(x^n)$.

```
void _nmod_poly_tanh_series(mp_ptr g, mp_srcptr h, long 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, long n)
```

Set $g = \tanh(h) + O(x^n)$.

§15. nmod_mat

Matrices over $\mathbf{Z}/n\mathbf{Z}$ for word-sized
moduli

15.1 Introduction

An `nmod_mat_t` represents a matrix of integers modulo n , for any non-zero n that fits in a single limb, up to $2^{32} - 1$ or $2^{64} - 1$. The implementation uses functions and data types of the `nmod_vec` module for low-level operations.

One or both dimensions of a matrix may be zero.

Except where otherwise noted, it is assumed all entries in input data are already reduced in the range $[0, n)$. It is also assumed that all arguments have the same modulus.

Functions that require the modulus to be a prime number document this requirement explicitly.

15.2 Memory management

```
void nmod_mat_init(nmod_mat_t mat, long rows, long 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 the elements of `mat` to those of `src`. It is assumed that `mat` and `src` have identical dimensions, but the moduli need not be the same. The elements will be reduced to the modulus of `mat`.

```
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.

15.3 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]
```

15.4 Random matrix generation

```
void nmod_mat_randtest(nmod_mat_t mat, flint_rand_t state)
```

Sets the elements to uniformly random numbers between 0 and $m - 1$ inclusive, where m is the modulus of `mat`.

```
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,
    long 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_randrunk(nmod_mat_t mat, long 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(fmpz_mat_t mat, long 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.

15.5 Matrix arithmetic

```
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. Entries will be reduced to the modulus of B .

```
void nmod_mat_add(nmod_mat_t C, nmod_mat_t A, nmod_mat_t B)
```

Computes $C = A + B$. Dimensions must be identical. Entries will be reduced to the modulus of C .

```
void nmod_mat_mul(nmod_mat_t C, nmod_mat_t A, nmod_mat_t B)
```

Computes the matrix product $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B . Entries will be reduced to the modulus of C .

```
void nmod_mat_mul_classical(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

Performs matrix multiplication using the classical algorithm, by calling `_nmod_mat_mul_1()` etc. The interface is identical to `nmod_mat_mul()`.

```
void nmod_mat_mul_strassen(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

Performs matrix multiplication respectively using Strassen multiplication. The interface is identical to `nmod_mat_mul()`.

```
void _nmod_mat_mul_1(nmod_mat_t C, nmod_mat_t A, nmod_mat_t
    B)
```

```
void _nmod_mat_mul_2(nmod_mat_t C, nmod_mat_t A, nmod_mat_t
    B)
```

```
void _nmod_mat_mul_3(nmod_mat_t C, nmod_mat_t A, nmod_mat_t
    B)
```

```
void _nmod_mat_mul_transpose_1(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

```
void _nmod_mat_mul_transpose_2(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

```
void _nmod_mat_mul_transpose_3(nmod_mat_t C, nmod_mat_t A,
    nmod_mat_t B)
```

Computes $C = AB$ or $C = AB^T$, respectively, using classical matrix multiplication. That is, for each i and j , C_{ij} is set to the scalar product of row i of A with column j of B (or row j of B when transposed).

The `_X` version uses a register `X` limbs wide to accumulate the scalar product, and a modular reduction is performed only at the end. The caller must ensure that elements are sufficiently small for overflow not to occur.

Let k be the scalar product size, i.e. the number of columns of A , and let a and b the largest absolute values of the elements in A and B , respectively. Then the following conditions are sufficient to guarantee correctness:

- 1 : $k \cdot a \cdot b < 2^{\text{FLINT_BITS}}$
- 2 : $k \cdot a \cdot b < 2^{(2 \cdot \text{FLINT_BITS})}$
- 3 : always

Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B .

15.6 Row reduction, solving

```
long _nmod_mat_rowreduce(nmod_mat_t mat, int options)
```

Row reduces the m -by- n matrix `mat` using standard in-place Gaussian elimination with pivoting. The modulus `mat->mod.n` must be a prime number.

The `options` parameter is a bitfield which may be set to any combination of the following flags, where using `options = 0` to disable all and perform in-place LU decomposition.

- **ROWREDUCE_FAST_ABORT.**

If set, the function immediately aborts and returns 0 if the matrix is detected to be rank-deficient, i.e. singular. In this event, the state of the matrix will be undefined.

- **ROWREDUCE_FULL.**

If set, performs Gauss–Jordan elimination, i.e. eliminates the elements above each pivot element as well as those below. If not set, regular Gaussian elimination is performed and only the elements below pivots are eliminated.

- **ROWREDUCE_CLEAR_LOWER.**

If set, clears, i.e. zeros, elements below the pivots.

If not set, the output becomes the LU decomposition of the input matrix. That is, the input matrix A is overwritten with L , U such that $A = PLU$ where P is a permutation matrix. U is stored in the upper triangular part (including the main diagonal), and L is stored with an implicit unit main diagonal in the lower triangular part.

Pivoting (to avoid division by zero entries) is performed by permuting the vector of row pointers in-place so that they point to the successive pivot rows. The matrix entries themselves retain their original order in memory.

The return value r is the rank of the matrix, multiplied by a sign indicating the parity of row interchanges. If $r = 0$, the matrix has rank zero, unless **ROWREDUCE_FAST_ABORT** is set, in which case $r = 0$ indicates any deficient rank. Otherwise, the leading non-zero entries of $a[0], a[1], \dots, a[|r| - 1]$ will point to the successive pivot elements. If $|r| = m = n$, the determinant of the matrix is given by $\text{sgn}(r)$ times the product of the entries on the main diagonal.

```
mp_limb_t nmod_mat_det(nmod_mat_t A)
```

Returns the determinant of A . The modulus of A must be a prime number.

```
long nmod_mat_rank(nmod_mat_t A)
```

Returns the rank of A . The modulus of A must be a prime number.

```
void _nmod_mat_solve_lu_precomp(mp_limb_t * b, mp_limb_t **
    LU, long n, nmod_t mod)
```

Transforms b to the solution x of $LUx = b$ where LU points to the rows of a precomputed LU factorisation of a nonsingular n -by- n matrix with modulus `mod.n`. The modulus must be a prime number.

```
int nmod_mat_solve(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 0 if A has full rank; otherwise returns 1 and sets the elements of x to undefined values.

```
int nmod_mat_solve_mat(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 0 if A has full rank; otherwise returns 1 and sets the elements of X to undefined values.

```
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.

§16. arith

Arithmetic functions

16.1 Introduction

This module implements arithmetic functions, number-theoretic and combinatorial special number sequences and polynomials.

16.2 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_unfactor(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.

16.3 Primorials

```
void fmpz_primorial(fmpz_t res, long n)
```

Sets `res` to “ n primorial” or $n\#$, the product of all prime numbers less than or equal to n .

16.4 Harmonic numbers

```
void mpq_harmonic(mpq_t res, long n)
```

Sets `res` to the n th harmonic number $H_n = 1 + 1/2 + 1/3 + \dots + 1/n$. 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 `mpn_harmonic_odd_balanced()` is used.

16.5 Stirling numbers

```
void fmpz_stirling1u(fmpz_t s, long n, long k)
```

```
void fmpz_stirling1(fmpz_t s, long n, long k)
```

```
void fmpz_stirling2(fmpz_t s, long n, long 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 $x_{(n)} = x(x-1)(x-2)\dots(x-n+1)$ is a falling factorial and $x^{(n)} = x(x+1)(x+2)\dots(x+n-1)$ 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 fmpz_stirling1u_vec(fmpz * row, long n, long klen)
```

```
void fmpz_stirling1_vec(fmpz * row, long n, long klen)
```

```
void fmpz_stirling2_vec(fmpz * row, long n, long klen)
```

Computes the row of Stirling numbers $S(n, 0)$, $S(n, 1)$, $S(n, 2)$, \dots , $S(n, klen-1)$.

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 fmpz_stirling1u_vec_next(fmpz * row, fmpz * prev, long
    n, long klen)
```

```
void fmpz_stirling1_vec_next(fmpz * row, fmpz * prev, long
    n, long klen)
```

```
void fmpz_stirling2_vec_next(fmpz * row, fmpz * prev, long
    n, long klen)
```

Given the vector `prev` containing a row of Stirling numbers $S(n-1,0)$, $S(n-1,1)$, $S(n-1,2)$, ..., $S(n-1,klen-2)$, computes and stores in the row argument $S(n,0)$, $S(n,1)$, $S(n,2)$, ..., $S(n,klen-1)$. It is assumed that `klen` is at most $n+1$.

The row and `prev` arguments are permitted to be the same, meaning that the row will be updated in-place.

```
void fmpz_stirling1u_mat(fmpz ** rows, long n)
```

```
void fmpz_stirling1_mat(fmpz ** rows, long n)
```

```
void fmpz_stirling2_mat(fmpz ** rows, long n)
```

Computes an n -by- n matrix of Stirling numbers:

```
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)
      .
      .
      .
row n-1 : S(n-1,0), S(n-1,1), ..., S(n-1,n-1)
```

In effect, if `rows` are the rows of an `fmpz_mat_t`, this stores the Stirling number analogue of Pascal's triangle as a lower triangular matrix, with ones on the main diagonal; entries above the main diagonal will not be touched, and should be zeroed by the caller if necessary.

For any n , the S_1 and S_2 matrices thus obtained are inverses of each other.

16.6 Bell numbers

```
void bell_number(fmpz_t b, ulong n)
```

Sets `b` to the n :th 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 uses a table lookup if B_n fits in a single word, and otherwise evaluates a precise truncation of the series $B_n = e^{-1} \sum_{k=0}^{\infty} \frac{k^n}{k!}$ using binary splitting.

```
void bell_number_vec(fmpz * b, long n)
```

Sets `b` to the vector of Bell numbers B_0, B_1, \dots, B_{n-1} inclusive. Automatically switches between the `recursive` and `multi_mod` algorithms depending on the size of n .

```
double 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 [?].

```
void _bell_number_vec_recursive(fmpz * b, long n)
```

Sets `b` to the vector of Bell numbers B_0, B_1, \dots, B_{n-1} inclusive. This function uses table lookup if B_{n-1} fits in a single word, and a simple triangular recurrence otherwise.

```
void _bell_number_vec_multi_mod(fmpz * b, long n)
```

Sets `b` to the vector of Bell numbers B_0, B_1, \dots, B_{n-1} inclusive.

This function evaluates the Bell numbers modulo several limb-size primes using the exponential generating function

$$e^{e^x - 1} = \sum_{n=0}^{\infty} \frac{B_n}{n!} x^n$$

and `nmod_poly` arithmetic. A tight bound for the number of needed primes is computed using `bell_number_size`, and the final integer values are recovered using balanced CRT reconstruction.

16.7 Bernoulli numbers and polynomials

```
void 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 `_bernoulli_number_zeta`.

```
void bernoulli_number_vec(fmpz * num, fmpz * den, long n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of the Bernoulli numbers $B_0, B_1, B_2, \dots, B_{n-1}$ inclusive. This function automatically chooses between the `recursive`, `zeta` and `multi_mod` algorithms according to the size of n .

```
void 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 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 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 `bernoulli_number_vec` and then rescales the coefficients efficiently.

```
void _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 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 _bernoulli_number_vec_recursive(fmpz * num, fmpz * den,
    long n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of $B_0, B_1, B_2, \dots, B_{n-1}$ inclusive.

The first few entries are computed using `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 _bernoulli_number_vec_zeta(fmpz * num, fmpz * den, long
    n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of $B_0, B_1, B_2, \dots, B_{n-1}$ inclusive. Uses repeated direct calls to `_bernoulli_number_zeta`.

```
void _bernoulli_number_vec_multi_mod(fmpz * num, fmpz * den,
    long n)
```

Sets the elements of `num` and `den` to the reduced numerators and denominators of $B_0, B_1, B_2, \dots, B_{n-1}$ 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 [?], saves about half of the time compared to the usual generating function $x/(e^x - 1)$ since the odd terms vanish.

16.8 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. $E_0, E_1, E_2, \dots = 1, 0, -1, 0, 5, 0, -61, 0, 1385, 0, \dots$. 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 euler_number(fmpz_t res, ulong n)
```

Sets `res` to the Euler number E_n . Currently calls `_euler_number_zeta`.

```
void euler_number_vec(fmpz * res, long n)
```

Computes the Euler numbers E_0, E_1, \dots, 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 `euler_number_size`, and the final integer values are recovered using balanced CRT reconstruction.

```
double 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 _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 $\chi = \{0, 1, 0, -1\}$.

16.9 Legendre polynomials

```
void legendre_polynomial(fmpq_poly_t poly, ulong n)
```

Sets `poly` to the n -th Legendre polynomial

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} \left[(x^2 - 1)^n \right].$$

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 $\gcd(n!, 2^n) = 2^{\lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \dots}$.

16.10 Multiplicative functions

```
void fmpz_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 fmpz_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 fmpz_divisor_sigma(fmpz_t res, const fmpz_t n, ulong k)
```

Sets `res` to $\sigma_k(n)$, the sum of k th powers of all divisors of n .

```
void fmpz_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 fmpz_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 `fmpz_poly_ramanujan_tau()`. Thus the speed of `fmpz_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 `fmpz_poly_ramanujan_tau()` directly.

```
void fmpz_poly_ramanujan_tau(fmpz_poly_t res, long n)
```

Sets `res` to the polynomial with coefficients $\tau(0), \tau(1), \dots, \tau(n-1)$, giving the initial n terms in the series expansion of $f(q) = q \prod_{k \geq 1} (1 - q^k)^{24}$.

The algorithm is borrowed from the `delta_qexp()` function in Sage, written by William Stein and David Harvey, and consists of expanding the series of the equivalent representation

$$f(q) = q \left(\sum_{k \geq 0} (-1)^k (2k+1) q^{k(k+1)/2} \right)^8.$$

The first squaring is done directly since the polynomial is very sparse at this point.

16.11 Swinnerton-Dyer polynomials

```
void swinnerton_dyer_polynomial(fmpz_poly_t poly, ulong n)
```

Sets `poly` to the Swinnerton-Dyer polynomial S_n , defined as the integer polynomial

$$S_n = \prod (x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \dots \pm \sqrt{p_n})$$

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.

16.12 Partition function

```
void partition_function_vec(fmpz * res, long len)
```

Computes first `len` values of the partition function $p(n)$ starting with $p(0)$. Uses inversion of Euler's pentagonal series.

```
void partition_function_nmod_vec(mp_ptr res, long 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.

16.13 Landau's function

```
void landau_function_vec(fmpz * res, long 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 [?]. The running time is $O(n^{3/2}/\sqrt{\log n})$.

§17. ulong_extras

Unsigned single limb arithmetic

17.1 Random functions

```
void n_randinit(flint_rand_t state)
```

Initialise a random state for use in random functions. Currently this function does nothing, but must be used for compatibility with future versions of flint. In particular the random functions below are not implemented in a threadsafe manner.

```
void n_randinit(flint_rand_t state)
```

Release any memory used by a random state. Currently this function does nothing, but must be used for compatibility with future versions of flint.

```
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 $p_0 = 4294967311 = \text{nextprime}(2^{32})$ on a 64-bit machine and $p_0 = \text{nextprime}(2^{16})$ on a 32-bit machine and $p_1 = \text{nextprime}(p_0)$.

```
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_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 `LONG_MAX` is increased. This random function is mainly used for testing purposes. Warning: this function is not threadsafe and is for use in test code only. It should not be used in library code.

```
mp_limb_t n_randtest_not_zero(flint_rand_t state)
```

As for `n_randtest()`, but does not return 0. Warning: this function is not threadsafe and is for use in test code only. It should not be used in library code.

```
mp_limb_t n_randprime(flint_rand_t state, unsigned long
    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.

17.2 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.

17.3 Miscellaneous

```
ulong n_revbin(ulong in, ulong bits)
```

Returns the binary reverse of `in`, assuming it is the given number of bits long, 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 $n = 0$.

17.4 Basic arithmetic with precomputed inverses

```
mp_limb_t n_mod_precomp(mp_limb_t a, mp_limb_t n, double
    ninv)
```

Returns $a \bmod n$ given a precomputed inverse of n computed by `n_precompute_inverse()`. We require $n < 2^{53}$ and $n < 2^{(\text{FLINT_BITS}-1)}$ 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 n , the binary representation of a is exact and the mantissa is less than 2, thus we see that `m * ninv` can be at most one out in the mantissa. We now truncate `m * 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. But this is impossible, as $n < 2^{53}$. Thus 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 $a \bmod n$ 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 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 have been 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 $a \bmod n$ 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_qrnd_preinv()`.

The new version uses the new algorithm of Granlund and Möller [?]. 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 $a \bmod n$ 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 $a \bmod n$ 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_qrnd_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 [?] 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 $a \bmod n$, 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 [?] 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 $ab \bmod n$ given a precomputed inverse of n computed by `n_precompute_inverse()`. We require $n < 2^{53}$ 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 $\mathbf{a} * \mathbf{b}$ is multiplied by n 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.

```
n_mulmod2_preinv(mp_limb_t a, mp_limb_t b, mp_limb_t n,
                 mp_limb_t ninv)
```

Returns $ab \bmod n$ 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()`.

17.5 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 embellishment 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_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.

17.6 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.

17.7 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 $(a^{\text{exp}}) \% n$ 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(mp_limb_t a, mp_limb_signed_t exp,
    mp_limb_t n)
```

Returns $(a^{\text{exp}}) \% n$. We require $n < 2^{\text{FLINT_D_BITS}}$ 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 $(a^{\text{exp}}) \% n$ 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 $(a^{\text{exp}}) \% n$. 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.

17.8 Prime number generation and counting

```
void n_compute_primes(ulong num_primes)
```

Precomputes `num_primes` primes and their double precomputed inverses and stores them in `flint_primes` and `flint_prime_inverse`, respectively.

The algorithm is a simple sieve of Eratosthenes with the constant array of primes `flint_small_primes` as a starting point.

The sieve works by marking all multiples of small primes in the sieve, but the sieve does not contain entries for numbers below the current cutoff (in case the function may have already been called before).

One only needs to start sieving with p^2 as all smaller multiples of p have already been marked off.

At first p^2 may be less than the start of the sieve (the old cutoff), so this case is dealt with separately, but for all primes p beyond that all multiples of p starting at p^2 are marked off in the sieve.

As the small prime cutoff is currently 1030, primes can be computed up to almost $n = 2^{20}$, in fact $\lfloor n / \log_2(n) \times 0.7 \rfloor = 74898$ primes which actually takes us to 949937.

```
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` holds, or `n_prime_pi(flint_primes[n-1]) == n`, where `flint_primes` is indexed from zero.

Currently, this function simply extends `flint_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 [?] 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 n th prime number p_n , using the mathematical indexing convention $p_1 = 2, p_2 = 3, \dots$.

This function simply ensures that `flint_primes` is large enough and then looks up `flint_primes[n-1]`.

```
void n_nth_prime_bounds(mp_limb_t *lo, mp_limb_t *hi, ulong
    n)
```

Calculates lower and upper bounds for the n th prime number p_n , $lo \leq p_n \leq hi$. 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 $n > 5$:

$$\begin{aligned} p_n &> n(\ln n + \ln \ln n - 1) \\ p_n &< n(\ln n + \ln \ln n) \\ p_n &< n(\ln n + \ln \ln n - 0.9427) \quad (n \geq 15985) \end{aligned}$$

The first inequality was proved by Dusart [?], and the last is due to Massias and Robin [?]. 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.

17.9 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 `flint_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 [?] 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 `~0L` 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 [?, 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. Up to 10^{16} this simply calls `n_is_probabprime()` which is a primality test up to that limit. Beyond that point it calls `n_is_probabprime()` and returns 0 if n is composite, then it calls `n_is_prime_pocklington()` which proves the primality of n in most cases. As a fallback, `n_is_prime_pseudosquare()` is called, which will unconditionally prove the primality of n .

```
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 j th element of the Fibonacci sequence $0, 1, 1, 2, 3, 5, \dots$, 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 [?, pp. 142].

We require that n is not divisible by 2 or 5.

```
int n_is_probabprime_BPSW(mp_limb_t n)
```

Implements the Bailey–Pomerance–Selfridge–Wagstaff probable primality test. There are no known counterexamples to this being a primality test. For further details, see [?].

```
int n_is_probabprime_lucas(mp_limb_t n)
```

For details on Lucas pseudoprimes, see [?, pp. 143].

We implement a variant of the Lucas pseudoprime test as described by Baillie and Wagstaff [?].

```
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 10^{16} it uses a number of strong probable prime tests, `n_is_strong_probabprime_precomp()`, 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, but it may well declare some composites to be prime.

17.10 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 so long as 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 so long as 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 ANDed 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 .

17.11 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 in `flint_primes` with index `start`.

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 ANDed 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 [?]. 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_SQUFOF(mp_limb_t n, ulong iters)
```

Attempts to split n using the given number of iterations of SQUFOF. Simply set `iters` to `~0L` for maximum persistence.

The version of SQUFOF implemented here is as described by Gower and Wagstaff [?].

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 certified prime, one may set `proved` to 1, otherwise set it to 0, and they will only be probable primes (with no known counterexamples to the conjecture that they are in fact all prime).

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.

17.12 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 $n = 0, 1, \dots, \text{len} - 1$. 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 .

17.13 Factorials

```
mp_limb_t n_factorial_mod2_preinv(ulong n, mp_limb_t p,
    mp_limb_t pinv)
```

Returns $n! \bmod p$ given a precomputed inverse of p as computed by `n_preinvert_limb()`.

§18. long_extras

Signed single limb arithmetic

18.1 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, LONG_MAX and LONG_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.

§19. longlong.h

64-bit arithmetic

19.1 Auxiliary asm macros

```
umul_ppmm(high_prod, low_prod, multiplier, multiplicand)
```

Multiplies two single limb integers MULTIPLIER and MULTIPLICAND, and generates a two limb product in HIGH_PROD and LOW_PROD.

```
smul_ppmm(high_prod, low_prod, multiplier, multiplicand)
```

As for umul_ppmm() but the numbers are signed.

```
udiv_qrnnnd(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_qrnnnd(quotient, remainder, high_numerator,
            low_numerator, denominator)
```

As for udiv_qrnnnd() 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.

```
sub_ddmss(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

$$\text{invxl} = (B^2 - B*x - 1)/x = (B^2 - 1)/x - B$$

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 [?, 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 $m = \text{invxl} + B$ 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_qrnmnd_preinv(q, r, nh, nl, d, di)
```

As for `udiv_qrnmnd()` 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
```

§20. mpn_extras

20.1 Utility functions

```
void 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 mpn_zero_p(mp_srcptr x, mp_size_t xsize)
```

Returns 1 if all limbs of $(x, xsize)$ are zero, otherwise 0.

20.2 Divisibility

```
int mpn_divisible_1_p(x, xsize, d)
```

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 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 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 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, \dots , 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 mpn_factor_trial(mp_srcptr x, mp_size_t xsize, long  
    start, long 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 .

20.3 Special numbers

```
void mpn_harmonic_odd_balanced(mp_ptr t, mp_size_t * tsize,
    mp_ptr v, mp_size_t * vsize, long a, long b, long n, int
    d)
```

Computes (t, tsize) and (v, vsize) such that $t/v = H_n = 1 + 1/2 + \dots + 1/n$. 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 $1 + 1/2 + \dots + 1/n$ computed without normalization; i.e. t and v should have room to fit $n!$ plus one extra limb.

§21. profiler

21.1 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);
```

```
printf("cpu=%ldms wall=%ldms\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);
```

```
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)`.

21.2 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);

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`.