FLINT 1.0.12: Fast Library for Number Theory

William B. Hart and David Harvey

July 11, 2008

Contents

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 and David Harvey of Harvard University in the US.

As of version 1.0, FLINT compiles on and supports 32 and 64 bit x86 processors, the G5 and Alpha processors, though in theory it compiles on any machine with gcc version 3.4 or later and with GMP version 4.2.1 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, e.g: all the functions in the file `fmpz_poly.c` have test functions in the file `fmpz_poly-test.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.

Next, set the environment variables `FLINT_GMP_LIB_DIR` and `FLINT_GMP_INCLUDE_DIR` to point to your GMP library and include directories respectively. Also set the environment variables `FLINT_NTL_LIB_DIR` and `FLINT_NTL_INCLUDE_DIR` to point to your NTL library and include directories respectively.

Next type:

```
source flint_env
```

in the main directory of the FLINT directory tree.

Finally type:

```
make library
```

Move the library file `libflint.so`, `libflint.dll` or `libflint.dylib` (depending on your platform) into your library path and move all the .h files in the main directory of FLINT into your include path.

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 GMP with the options `-lflint -lgmp`.

If you are using the NTL-interface, you will also need to link against NTL with the `-lntl` linker option.

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 the test programs, simply type:

```
make test
```

in the main FLINT directory.

The following is a list of the test programs which should be run:

`mpn_extras-test`

`fmpz_poly-test`

`fmpz-test`

`ZmodF-test`

`ZmodF_poly-test`

`mpz_poly-test`

`ZmodF_mul-test`

`long_extras-test`

`zmod_poly-test`

`NTL-interface-test`

4 Reporting bugs

The maintainers wish to be made aware of any and all bugs. Please send an email with your bug report to hart_wb@yahoo.com.

If possible please include details of your system, version of gcc, version of GMP and precise details of how to replicate the bug.

Note that FLINT needs to be linked against version 4.2.1 or later of GMP and must be compiled with gcc version 3.4 or later.

5 Example programs

FLINT comes with a number of example programs to demonstrate current and future FLINT features. To make the example programs, type:

```
make examples
```

The current example programs are:

`delta_qexp` Compute the first n terms of the delta function, e.g. `delta_qexp 1000000` will compute the first one million terms of the q-expansion of delta.

`BPTJCubes` Implements the algorithm of Beck, Pine, Tarrant and Jensen for finding solutions to the equation $x^3 + y^3 + z^3 = k$.

`bernoulli_zmod` Compute bernoulli numbers modulo a large number of primes.

`expmod` Computes a very large modular exponentiation.

6 FLINT macros

In the file flint.h are 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 the number of bits per limb, whichever is smaller. This will have the value 53 or 32 on currently supported architectures. Numerous functions using precomputed inverses only support operands up to `FLINT_D_BITS - 1` bits, hence the macro.

`FLINT_ABS(x)` returns the absolute value of a `long x`.

`FLINT_MIN(x, y)` returns the minimum of two `long` or two `unsigned long` values `x` and `y`.

`FLINT_MAX(x, y)` returns the maximum of two `long` or two `unsigned long` values `x` and `y`.

`FLINT_BIT_COUNT(x)` returns the number of binary bits required to represent an `unsigned long x`.

7 The fmpz_poly module

The `fmpz_poly_t` data type represents elements of $\mathbb{Z}[x]$. The `fmpz_poly` module provides routines for memory management, basic arithmetic, and conversions to/from other types.

Each coefficient of an `fmpz_poly_t` is an integer of the FLINT `fmpz_t` type.

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

7.1 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
```

7.2 Definition of the fmpz_poly_t polynomial 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 == 0`, or if the final coefficient is nonzero. 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.

We now describe the functions available in `fmpz_poly`.

7.3 Initialisation and memory management

```
void fmpz_poly_init(fmpz_poly_t poly)
```

Initialise an `fmpz_poly_t` for use. The length of `poly` is set 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.

For efficiency reasons, a call to `fmpz_poly_init` does not actually allocate any memory for coefficients. Each of the functions will automatically allocate any space needed for coefficients and in fact the easiest way to use `fmpz_poly` is to let FLINT do all the allocation automatically.

To this end, a user need only ever make calls to the `fmpz_poly_init` and `fmpz_poly_clear` memory management functions if they so wish. Naturally, more efficient code may result if the other memory management functions are also used.

```
void fmpz_poly_realloc(fmpz_poly_t poly, unsigned long alloc)
```

Shrink or expand the polynomial so that it has space for precisely `alloc` coefficients. If `alloc` is less than the current length, the polynomial is truncated (and then normalised), otherwise the coefficients and current length remain unaffected.

If the parameter `alloc` is zero, any space currently allocated for coefficients in `poly` is free'd. A subsequent call to `fmpz_poly_clear` is still permitted and does nothing.

```
void fmpz_poly_fit_length(fmpz_poly_t poly, unsigned long alloc)
```

Expand the polynomial (if necessary) so that it has space for at least `alloc` coefficients. This function will never shrink the memory allocated for coefficients and the contents of the existing coefficients and the current length remain unaffected.

```
void fmpz_poly_fit_limbs(fmpz_poly_t poly, unsigned long limbs)
```

Currently all the coefficients of an `fmpz_poly_t` have the same number of limbs of space allocated for them (plus an additional limb for the sign/size limb). This function can be used to increase the space allocated for the coefficients. As all functions in the `fmpz_poly` module automatically manage memory allocation for the user, this function should only be used when directly manipulating the coefficients by means of the functions in the `fmpz` module (described below). In a later version of FLINT, this function will become defunct, as FLINT will automatically reallocate `fmpz_t`'s when there is insufficient space, and this will include polynomial coefficients.

```
void fmpz_poly_clear(fmpz_poly_t poly)
```

Free all memory used by the coefficients of `poly`. The polynomial object `poly` cannot be used again until a subsequent call to an initialisation function is made.

7.4 Setting/retrieving coefficients

```
void fmpz_poly_get_coeff_mpz(mpz_t x, const fmpz_poly_t poly,
                                         unsigned long n)
```

Retrieve coefficient n as an `mpz_t`.

Coefficients are numbered from zero, starting with the constant coefficient.

Sets `x` to zero when $n >=$ `poly->length`.

```
void fmpz_poly_get_coeff_mpz_read_only(mpz_t x,
                          const fmpz_poly_t poly, unsigned long n)
```

Retrieve coefficient n as a read only `mpz_t`. The function must be passed an uninitialised `mpz_t`. The `mpz_t` can then be used as an input to a GMP functions, but not as an output. Its contents may be inspected, but not alterered. This function is faster than `fmpz_poly_get_coeff_mpz` which makes an extra copy of the data.

Coefficients are numbered from zero, starting with the constant coefficient.

Sets `x` to zero when $n >=$ `poly->length`.

```
void fmpz_poly_set_coeff_mpz(fmpz_poly_t poly, unsigned long n,
                                                     mpz_t x)
```

Set coefficient n to the value of the given `mpz_t`.

Coefficients are numbered from zero, starting with the constant coefficient. If n represents a coefficient beyond the current length of `poly`, zero coefficients are added in between the existing coefficients and the new coefficient, if required.

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

Retrieve coefficient n as an `fmpz_t`.

Coefficients are numbered from zero, starting with the constant coefficient.

Sets `x` to zero when $n >=$ `poly->length`.

```
void fmpz_poly_set_coeff_fmpz(fmpz_poly_t poly, unsigned long n,
                                                          fmpz_t x)
```

Set coefficient n to the value of the given `fmpz_t`.

Coefficients are numbered from zero, starting with the constant coefficient. If n represents a coefficient beyond the current length of `poly`, zero coefficients are added in between the existing coefficients and the new coefficient, if required.

```
unsigned long fmpz_poly_get_coeff_ui(const fmpz_poly_t poly,
                                                  unsigned long n)
```

Return the absolute value of coefficient n as an `unsigned long`.

Coefficients are numbered from zero, starting with the constant coefficient. If the coefficient is longer than a single limb, the first limb is returned.

Returns zero when $n >=$ `poly->length`.

```
void fmpz_poly_set_coeff_ui(fmpz_poly_t poly, unsigned long n,
                                                  unsigned long x)
```

Set coefficient n to the value of the given `unsigned long`.

Coefficients are numbered from zero, starting with the constant coefficient. If n represents a coefficient beyond the current length of `poly`, zero coefficients are added in between the existing coefficients and the new coefficient, if required.

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

Return the value of coefficient n as a `long`.

Coefficients are numbered from zero, starting with the constant coefficient. If the coefficient will not fit into a `long`, i.e. if its absolute value takes up more than `FLINT_BITS - 1` bits then the result is undefined.

Returns zero when $n >=$ `poly->length`.

```
void fmpz_poly_set_coeff_si(fmpz_poly_t poly, unsigned long n,
                                                            long x)
```

Set coefficient n to the value of the given `long`.

Coefficients are numbered from zero, starting with the constant coefficient. If n represents a coefficient beyond the current length of `poly`, zero coefficients are added in between the existing coefficients and the new coefficient, if required.

```
fmpz_t fmpz_poly_get_coeff_ptr(fmpz_poly_t poly, unsigned long n)
```

Return a reference to coefficient n (as an `fmpz_t`). 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.

Coefficients are numbered from zero, starting with the constant coefficient.

Returns NULL when $n >=$ `poly->length`.

```
fmpz_t fmpz_poly_lead(const fmpz_poly_t poly)
```

Return a reference to leading coefficient (as an `fmpz_t`) of `poly`. 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 has length zero.

7.5 String conversions and I/O

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 whitespace. The first integer gives the length of the polynomial; the remaining `length` integers are the coefficients. For example $5x^3 - x + 1$ is represented by the string "`4 1 -1 0 5`", and the zero polynomial is represented by "`0`". The coefficients may be signed and arbitrary precision.

The string representation of the functions appended by `_pretty` includes only the non-zero terms of the polynomial, starting with the one of highest degree. Each term starts with a coefficient, prepended with a sign (positive or negative), followed by the character `*`, followed by a variable name, which must be passed as a string parameter to the function, followed by a carot `^` followed by a non-negative exponent.

If the sign of the leading coefficient is positive, it is omitted. Also the exponents of the degree 1 and 0 terms are omitted, as is the variable and the `*` character in the case of the degree 0 coefficient. If the coefficient is plus or minus one, the coefficient is omitted, except for the sign.

Some examples of the `_pretty` representation are:

```
5*x^3+7*x-4
x^2+3
-x^4+2*x-1
x+1
5
```

```
int fmpz_poly_from_string(fmpz_poly_t poly, const char* s)
```

Import a polynomial from a string. If the string represents a valid polynomial the function returns 1, otherwise it returns 0.

```
char* fmpz_poly_to_string(const fmpz_poly_t poly)
char* fmpz_poly_to_string_pretty(const fmpz_poly_t poly,
                                                    const char * x)
```

Convert a polynomial to a string and return a pointer to the string. Space is allocated for the string by this function and must be freed when it is no longer used, by a call to `free`.

The `pretty` version must be supplied with a string `x` which represents the variable name to be used when printing the polynomial.

```
void fmpz_poly_fprint(const fmpz_poly_t poly, FILE* f)
void fmpz_poly_fprint_pretty(const fmpz_poly_t poly, FILE* f,
                                                    const char * x)
```

Convert a polynomial to a string and write it to the given stream.

The `pretty` version must be supplied with a string `x` which represents the variable name to be used when printing the polynomial.

```
void fmpz_poly_print(const fmpz_poly_t poly)
void fmpz_poly_print_pretty(const fmpz_poly_t poly, const char * x)
```

Convert a polynomial to a string and write it to `stdout`.

The `pretty` version must be supplied with a string `x` which represents the variable name to be used when printing the polynomial.

```
void fmpz_poly_fread(fmpz_poly_t poly, FILE* f)
```

Read a polynomial from the given stream. Return 1 if the data from the stream represented a valid polynomial, otherwise return 0.

```
void fmpz_poly_read(fmpz_poly_t poly)
```

Read a polynomial from `stdin`. Return 1 if the data read from `stdin` represented a valid polynomial, otherwise return 0.

7.6 Polynomial parameters (length, degree, max limbs, etc.)

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

Return `poly->length - 1`. The zero polynomial is defined to have degree -1.

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

Return `poly->length`. The zero polynomial is defined to have length 0.

```
unsigned long 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`.

```
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.

```
long fmpz_poly_max_bits1(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.

The assumption is made that the absolute value of each coefficient fits into an unsigned long. This function will be more efficient than the more general `fmpz_poly_max_bits` in this situation.

7.7 Assignment and basic manipulation

```
void fmpz_poly_set(fmpz_poly_t output, const fmpz_poly_t poly)
```

Set polynomial `output` equal to the polynomial `poly`.

```
void fmpz_poly_swap(fmpz_poly_t poly1, fmpz_poly_t poly2)
```

Efficiently swap two polynomials. The coefficients are not moved in memory, pointers are simply switched.

```
void fmpz_poly_zero(fmpz_poly_t poly)
```

Set the polynomial to the zero polynomial.

```
void fmpz_poly_zero_coeffs(fmpz_poly_t poly, unsigned long n)
```

Set the first n coefficients of `poly` to zero.

```
void fmpz_poly_neg(fmpz_poly_t output, fmpz_poly_t poly)
```

Negate the polynomial `poly`, i.e. set `output` to `-poly`.

```
void fmpz_poly_truncate(fmpz_poly_t poly, const unsigned long trunc)
```

If `trunc` is less than the current length of the polynomial, truncate the polynomial to that length. Note that as the function normalises its output, the eventual length of the polynomial may be less than `trunc`.

```
void fmpz_poly_reverse(fmpz_poly_t output,
                       const fmpz_poly_t poly, unsigned long length)
```

This function considers the polynomial `poly` to be of length n, notionally truncating and zero padding if required, and reverses the result. Since this function normalises its result the eventual length of `output` may be less than `length`.

```
void fmpz_poly_normalise(fmpz_poly_t poly)
```

This function normalises `poly` so that the leading coefficient is non-zero (or the polynomial is the zero polynomial). As all functions in `fmpz_poly` expect and return normalised polynomials, this function is only used when manipulating the coefficients directly by making use of the functions in the `fmpz` module (described below).

7.8 Conversions

```
void fmpz_poly_to_zmod_poly(zmod_poly_t zpol, fmpz_poly_t fpol)
```

Reduce the coefficients of the `fmpz_poly_t fpol` mod the modulus of the `zmod_poly_t zpol` and store the result in `zpol`.

This function is provided to enable the implementation of multimodular algorithms.

```
void zmod_poly_to_fmpz_poly_unsigned(fmpz_poly_t fpol,
                                     zmod_poly_t zpol)
```

Convert the `zmod_poly_t zpol` to an `fmpz_poly_t`. The coefficients of the `fmpz_poly_t` will all be unsigned.

```
void zmod_poly_to_fmpz_poly(fmpz_poly_t fpol, zmod_poly_t zpol)
```

Convert the `zmod_poly_t zpol` to an `fmpz_poly_t`. If `p` is the modulus of `zpol` then coefficients which lie in $[0, p/2]$ are unchanged, however, coefficients a in the range $(p/2, p)$ become $a - p$.

This function is provided to enable the implementation of multimodular algorithms.

7.9 Chinese remaindering

```
int fmpz_poly_CRT_unsigned(fmpz_poly_t res, fmpz_poly_t fpol,
                zmod_poly_t zpol, fmpz_t newmod, fmpz_t oldmod)
```

Performs modular recombination using the Chinese Remainder Theorem. If `zpol` has modulus p, `newmod` is set equal to `oldmod*p` and each coefficient of `res` is set to the unique value modulo `newmod`, in the range $[0, \mathrm{newmod})$ which is a modulo `oldmod` and b modulo p, where a is the coefficient of `fpol` and b is the corresponding coefficient of `zpol`.

The coefficients of `fpol` are assumed to be unsigned.

```
int fmpz_poly_CRT(fmpz_poly_t res, fmpz_poly_t fpol,
                zmod_poly_t zpol, fmpz_t newmod, fmpz_t oldmod)
```

Performs modular recombination using the Chinese Remainder Theorem. If `zpol` has modulus p, `newmod` is set equal to `oldmod*p` and each coefficient of `res` is set to the unique value modulo `newmod`, in the range $(-\mathrm{newmod}/2, \mathrm{newmod}/2]$ which is a modulo `oldmod` and b modulo p, where a is the coefficient of `fpol` and b is the corresponding coefficient of `zpol`.

7.10 Comparison

```
int fmpz_poly_equal(const fmpz_poly_t poly1,
                                  const fmpz_poly_t poly2)
```

Return 1 if the two polynomials are equal, 0 otherwise.

7.11 Shifting

```
void fmpz_poly_left_shift(fmpz_poly_t output,
                    const fmpz_poly_t poly, unsigned long n)
```

Shift poly to the left by n coefficients (multiply by x^n) and write the result to `output`. Zero coefficients are inserted.

The parameter n must be non-negative, but can be zero.

```
void fmpz_poly_right_shift(fmpz_poly_t output,
                    const fmpz_poly_t poly, unsigned long n)
```

Shift poly to the right by n coefficients (divide by x^n and discard the remainder) and write the result to `output`.

The parameter n must be non-negative, but can be zero. Shifting right by more than the current length of the polynomial results in the zero polynomial.

7.12 Norms

```
void fmpz_poly_2norm(fmpz_t norm, fmpz_poly_t pol)
```

Sets `norm` to the euclidean norm of `pol`, i.e. the integer square root of the sum of the squares of the coefficients of `pol`.

7.13 Addition/subtraction

```
void fmpz_poly_add(fmpz_poly_t output, const fmpz_poly_t poly1,
                                        const fmpz_poly_t poly2)
```

Set the output to the sum of the input polynomials.

Note that if poly1 and poly2 have the same length, cancellation may occur (if the leading coefficients have the same absolute values but opposite signs) and so the result may have less coefficients than either of the inputs.

```
void fmpz_poly_sub(fmpz_poly_t output, const fmpz_poly_t poly1,
                                        const fmpz_poly_t poly2)
```

Set the output to poly1 - poly2.

Note that if poly1 and poly2 have the same length, cancellation may occur (if the leading coefficients have the same values) and so the result may have less coefficients than either of the inputs.

7.14 Scalar multiplication and division

```
void fmpz_poly_scalar_mul_ui(fmpz_poly_t output,
                          const fmpz_poly_t poly, unsigned long x)
```

Multiply poly by the unsigned long x and write the result to output.

```
void fmpz_poly_scalar_mul_si(fmpz_poly_t output,
                                   const fmpz_poly_t poly, long x)
```

Multiply poly by the long x and write the result to output.

```
void fmpz_poly_scalar_mul_fmpz(fmpz_poly_t output,
                           const fmpz_poly_t poly, const fmpz_t x)
```

Multiply poly by the fmpz_t x and write the result to output.

```
void fmpz_poly_scalar_mul_mpz(fmpz_poly_t output,
                            const fmpz_poly_t poly, const mpz_t x)
```

Multiply poly by the mpz_t x and write the result to output.

```
void fmpz_poly_scalar_div_ui(fmpz_poly_t output,
                          const fmpz_poly_t poly, unsigned long x)
```

Divide poly by the unsigned long x, round quotients towards minus infinity, discard remainders and write the result to output.

```
void fmpz_poly_scalar_div_si(fmpz_poly_t output,
                             const fmpz_poly_t poly, long x)
```

Divide `poly` by the `long x`, round quotients towards minus infinity, discard remainders and write the result to `output`.

```
void fmpz_poly_scalar_tdiv_ui(fmpz_poly_t output,
                    const fmpz_poly_t poly, unsigned long x)
```

Divide `poly` by the `unsigned long x`, round quotients towards zero, discard remainders and write the result to `output`.

```
void fmpz_poly_scalar_tdiv_si(fmpz_poly_t output,
                             const fmpz_poly_t poly, long x)
```

Divide `poly` by the `long x`, round quotients towards zero, discard remainders and write the result to `output`.

```
void fmpz_poly_scalar_div_exact_ui(fmpz_poly_t output,
                    const fmpz_poly_t poly, unsigned long x)
```

Divide `poly` by the `unsigned long x`. Division is assumed to be exact and the result is undefined otherwise.

```
void fmpz_poly_scalar_div_exact_si(fmpz_poly_t output,
                             const fmpz_poly_t poly, long x)
```

Divide `poly` by the `long x`. Division is assumed to be exact and the result is undefined otherwise.

```
void fmpz_poly_scalar_div_fmpz(fmpz_poly_t output,
                     const fmpz_poly_t poly, const fmpz_t x)
```

Divide `poly` by the `fmpz_t x`, round quotients towards minus infinity, discard remainders, and write the result to `output`.

```
void fmpz_poly_scalar_div_mpz(fmpz_poly_t output,
                      const fmpz_poly_t poly, const mpz_t x)
```

Divide `poly` by the `mpz_t x`, round quotients towards minus infinity, discard remainders, and write the result to `output`.

7.15 Polynomial multiplication

```
void fmpz_poly_mul(fmpz_poly_t output, const fmpz_poly_t poly1,
                                        const fmpz_poly_t poly2)
```

Multiply the two given polynomials and return the result in `output`.

The length of the output polynomial will be `poly1->length + poly2->length - 1`.

```
void fmpz_poly_mul_trunc_n(fmpz_poly_t output,
 const fmpz_poly_t poly1, const fmpz_poly_t poly2, unsigned long n)
```

Multiply the two given polynomials and truncate the result to n coefficients, storing the result in `output`. This is sometimes known as a short product.

The length of the output polynomial will be at most the minimum of n and the value `poly1->length + poly2->length - 1`. It is permissible to set n to any non-negative value, however the function is optimised for n about half of `poly1->length + poly2->length`.

This function is more efficient than multiplying the two polynomials then truncating. It is the operation used when multiplying power series.

```
void fmpz_poly_mul_trunc_left_n(fmpz_poly_t output,
 const fmpz_poly_t poly1, const fmpz_poly_t poly2, unsigned long n)
```

Multiply the two given polynomials storing the result in `output`. This function guarantees all the coefficients except the first n, which may be arbitrary. This is sometimes known as an opposite short product.

The length of the output polynomial will be `poly1->length + poly2->length - 1` unless n is greater than or equal to this value, in which case it will return the zero polynomial. It is permissible to set n to any non-negative value, however the function is optimised for n about half of `poly1->length + poly2->length`.

For short polynomials, this function is more efficient than computing the full product.

7.16 Polynomial division

```
void fmpz_poly_divrem(fmpz_poly_t Q, fmpz_poly_t R,
                      const fmpz_poly_t A, const fmpz_poly_t B)
```

Performs division with remainder in $\mathbb{Z}[x]$. Computes polynomials `Q` and `R` in $\mathbb{Z}[x]$ such that the equation `A = B*Q + R`, holds. All but the final `B->length - 1` coefficients of `R` will be positive and less than the absolute value of the lead coefficient of `B`.

Note that in the special cases where the leading coefficient of `B` is ± 1 or `A = B*Q` for some polynomial `Q`, the result of this function is the same as if the computation had been done over $\mathbb{Q}$.

```
void fmpz_poly_div(fmpz_poly_t Q, const fmpz_poly_t A,
                                   const fmpz_poly_t B)
```

Performs division without remainder in $\mathbb{Z}[x]$. The computation returns the same result as `fmpz_poly_divrem`, but no remainder is computed. This is in general faster than computing quotient and remainder.

Note that in the special cases where the leading coefficient of `B` is ± 1 or `A = B*Q` for some polynomial `Q`, the result of this function is the same as if the computation had been done over $\mathbb{Q}$. In particular it can be used efficiently for exact division in $\mathbb{Z}[x]$.

```
void fmpz_poly_div_series(fmpz_poly_t Q, const fmpz_poly_t A,
                              const fmpz_poly_t B, unsigned long n)
```

Performs power series division in $\mathbb{Z}[[x]]$. The function considers the polynomials `A` and `B` to be power series of length n starting with the constant terms. The function assumes that `B` is normalised, i.e. that the constant coefficient is ± 1. The result is truncated to length n regardless of the inputs.

```
int fmpz_poly_divides(fmpz_poly_t Q, fmpz_poly_t A, fmpz_poly_t B)
```

If the polynomial `A` is divisible by the polynomial `B` this function returns 1 and sets `Q` to the quotient, otherwise it returns 0.

At this point, this function is provided for convenience only; it is not efficient when `B` does not actually divide `A`.

7.17 Pseudo division

```
void fmpz_poly_pseudo_divrem(fmpz_poly_t Q, fmpz_poly_t R,
      unsigned long * d, const fmpz_poly_t A, const fmpz_poly_t B)
```

Performs division with remainder of two polynomials in $\mathbb{Z}[x]$, notionally returning the results in $\mathbb{Q}[x]$ (actually in $\mathbb{Z}[x]$ with a single common denominator).

Computes polynomials `Q` and `R` such that `lead(B)^d*A = B*Q + R` where `R` has degree less than that of `B`.

This function may be used to do division of polynomials in $\mathbb{Q}[x]$ as follows. Suppose polynomials `C` and `D` are given in $\mathbb{Q}[x]$.

1) Write `C = d1*A` and `D = d2*B` for some polynomials `A` and `B` in $\mathbb{Z}[x]$ and integers `d1` and `d2`.

2) Use pseudo-division to compute `Q` and `R` in $\mathbb{Z}[x]$ so that `l^d*A = B*Q + R` where `l` is the leading coefficient of `B`.

3) We can now write `C = (d1/d2*D*Q + d1*R)/l^d`.

```
void fmpz_poly_pseudo_div(fmpz_poly_t Q, unsigned long * d,
                          const fmpz_poly_t A, const fmpz_poly_t B)
```

Performs division without remainder of two polynomials in $\mathbb{Z}[x]$, notionally returning the results in $\mathbb{Q}[x]$ (actually in $\mathbb{Z}[x]$ with a single common denominator).

Notionally computes polynomials `Q` and `R` such that `lead(B)^d*A = B*Q + R` where `R` has degree less than that of `B`, but returns only `Q`. This is slightly more efficient than computing the quotient and remainder.

7.18 Powering

```
void fmpz_poly_power(fmpz_poly_t output, const fmpz_poly_t poly,
                                         unsigned long exp)
```

Raises `poly` to the power `exp` and writes the result in `output`.

```
void fmpz_poly_power_trunc_n(fmpz_poly_t output,
      const fmpz_poly_t poly, unsigned long exp, unsigned long n)
```

Notionally raises `poly` to the power `exp`, truncates the result to length n and writes the result in `output`. This is computed much more efficiently than simply powering the polynomial and truncating.

This function can be used to raise power series to a power in an efficient way.

7.19 Gaussian content

```
void fmpz_poly_content(fmpz_t c, fmpz_poly_t poly)
```

Set the `fmpz_t c` to the Gaussian content of the polynomial `poly`, i.e. to the greatest common divisor of its coefficients.

```
void _fmpz_poly_primitive_part(fmpz_poly_t prim, fmpz_poly_t poly)
```

Set `prim` to the primitive part of the polynomial `poly`, i.e. to `poly` divided by its Gaussian content.

7.20 Greatest common divisor and resultant

```
void fmpz_poly_gcd(fmpz_poly_t res, const fmpz_poly_t poly1,
                                    const fmpz_poly_t poly2)
```

Sets `res` to the greatest common divisor of the polynomials `poly1` and `poly2`.

```
unsigned long fmpz_poly_resultant_bound(fmpz_poly_t a,
                                                  fmpz_poly_t b)
void fmpz_poly_resultant(fmpz_t r, fmpz_poly_t a, fmpz_poly_t b)
```

Compute the resultant of the polynomials `a` and `b`. If `a` and `b` are monic with $a(x) = \prod_i (x-\alpha_i)$ and $b(x) = \prod_j (x-\beta_j)$, when factored over the complex numbers, then the resultant is given by the expression $r(x) = \prod_{i,j}(\alpha_i - \beta_j)$. If the polynomials are not monic, and `a` and `b` have leading coefficients l_1 and l_2 and degrees d_1 and d_2 respectively, then this quantity is multiplied by $l_1^{d_2-1} l_2^{d_1-1}$.

Note that the resultant is zero iff the polynomials share a root over the algebraic closure of $\mathbb{Q}$.

Currently it is necessary to ensure `r` has sufficient space to store the result. The function `fmpz_poly_resultant_bound` is used to determine a bit bound on the number of bits `b` required and `r` must have space for `b/FLINT_BITS + 2` limbs.

In a future version of FLINT, this computation will not be necessary.

```
void fmpz_poly_xgcd(fmpz_t r, fmpz_poly_t s, fmpz_poly_t t,
                                        fmpz_poly_t a, fmpz_poly_t b)
```

Given coprime polynomials `a` and `b` this function computes polynomials `s` and `t` and the resultant `r` of the polynomials such that `r = a*s + b*t`.

See the function `fmpz_poly_resultant` for information on how large `r` needs to be to hold the result.

7.21 Modular arithmetic

```
void fmpz_poly_invmod(fmpz_t d, fmpz_poly_t H, fmpz_poly_t poly1,
                                                    fmpz_poly_t poly2)
```

Computes a polynomial `H` and a denominator `d` such that `poly1*H` is `d` modulo `poly2`.

Assumes that `poly1` and `poly2` are coprime and that `poly2` is monic.

7.22 Subpolynomials

A number of functions are provided for attaching an `fmpz_poly_t` object to an existing polynomial or to a range of coefficients of an existing polynomial providing an alias for the original polynomial or part thereof.

Each of the functions in this section normalise the subpolynomials so that they can be used as inputs to `fmpz_poly` functions.

As FLINT has no way of reallocating space in subpolynomials, they should not be used for outputs of `fmpz_poly` functions, but only for inputs. In a later version of FLINT, this restriction will be lifted.

Note that FLINT may perform suboptimally if a polynomial and an alias of the polynomial are passed as inputs to the same function, as FLINT has no way to tell that it is dealing with aliases of the same polynomial.

```
void fmpz_poly_attach(fmpz_poly_t output, const fmpz_poly_t poly)
```

Attach the `fmpz_poly_t` object `output` to the polynomial `poly`. Any changes made to the `length` field of `output` then do not affect `poly`.

```
void fmpz_poly_attach_shift(fmpz_poly_t output,
                        const fmpz_poly_t input, unsigned long n)
```

Attach the `fmpz_poly_t` object `output` to `poly` but shifted to the left by n coefficients. This is equivalent to notionally shifting the original polynomial right (dividing by x^n) then attaching to the result.

```
void fmpz_poly_attach_truncate(fmpz_poly_t output,
                        const fmpz_poly_t input, unsigned long n)
```

Attach the `fmpz_poly_t` object `output` to the first n coefficients of the polynomial `poly`. This is equivalent to notionally truncating the original polynomial to n coefficients then attaching to the result.

8 The fmpz module

The `fmpz` module is designed for manipulation of the FLINT flat multiprecision integer format `fmpz_t`.

Internally, the data for an `fmpz_t` has first limb a sign/size limb. If it is 0 the integer represented by the `fmpz_t` is 0. The absolute value of the sign/size limb is the number of subsequent limbs that the absolute value of the integer being represented, takes up. The absolute value of the integer is then stored as limbs, least significant limb first, in the subsequent limbs after the sign/size limb. If the sign/size limb is positive, a positive integer is intended and if the sign/size limb is negative the negative integer with the stored absolute value is intended.

The `fmpz_t` type is not intended as a standalone integer type. It is intended to be used in composite types such as polynomials and matrices which consist of many integer entries.

Currently the user is responsible for memory management of `fmpz_t`'s, i.e. one must ensure that the output of a function in the `fmpz` module contains sufficient space to store the result. This will be changed in a later version of FLINT, where automatic memory management will be done for the user.

To ensure that the correct number of limbs are available in each `fmpz_t` of an `fmpz_poly_t` one must currently call `void fmpz_poly_fit_limbs(fmpz_poly_t pol, unsigned long limbs)`, which will then ensure that each coefficient of `pol` has space for at least the given number of limbs (referring to the absolute value of the coefficients). Again, in a later version of FLINT, this step will be unnecessary as automatic memory management will be done for all `fmpz_t`'s, including coefficients of `fmpz_poly_t`'s.

Note that `fmpz_t`'s are not currently guaranteed to allow aliasing between inputs or between inputs and outputs. However some optimised inplace functions are provided.

8.1 A simple example

We start with a simple example of the use of the `fmpz` module.

This example sets x to 3 and adds 5 to it.

```
#include "fmpz.h"
 ....
fmpz_t x = fmpz_init(1); // Allocate 1 limb of space
fmpz_set_ui(x, 3);
fmpz_add_ui_inplace(x, 5);
printf("3␣+␣5␣is␣"); fmpz_print(x); printf("\n");
fmpz_clear(x);
```

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

8.2 Memory management

```
fmpz_t fmpz_init(unsigned long limbs)
```

Allocates space for an `fmpz_t` with the given number of limbs (plus an additional limb for the sign/size) on the heap and return a pointer to the space.

```
fmpz_t fmpz_realloc(fmpz_t f, unsigned long limbs)
```

Reallocate the space used by the `fmpz_t f` so that it has space for the given number of limbs (plus a sign/size limb). The parameter `limbs` must be non-negative. The existing contents of `f` are not altered if they still fit in the new size.

```
void fmpz_clear(const fmpz_t f)
```

Free space used by the `fmpz_t f`.

8.3 Random numbers

```
void fmpz_random_limbs2(fmpz_t x, unsigned long n)
```

Set `x` to a random number of n limbs consisting of long strings of ones and zeroes.

8.4 String operations

```
void fmpz_print(const fmpz_t f)
```

Print the multiprecision integer `f`. A minus sign is prepended if the integer is negative.

8.5 fmpz properties

```
unsigned long fmpz_size(const fmpz_t f)
```

Return the number of limbs used to store the absolute value of the multiprecision integer `f`.

```
unsigned long fmpz_bits(const fmpz_t f)
```

Return the number of bits required to store the absolute value of the multiprecision integer `f`.

```
long fmpz_sgn(const fmpz_t f)
```

Return the sign/size limb of the multiprecision integer `f`. The sign of the sign/size limb is the sign of the multiprecision integer. The absolute value of the sign/size limb is the size in limbs of the absolute value of the multiprecision integer `f`.

8.6 Assignment

```
void fmpz_set_ui(fmpz_t res, unsigned long x)
```

Set the multiprecision integer `res` to the `unsigned long x`.

```
void fmpz_set_si(fmpz_t res, long x)
```

Set the multiprecision integer `res` to the `long x`.

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

Set the multiprecision integer `res` to equal the multiprecision integer `f`.

8.7 Comparison

```
int fmpz_equal(const fmpz_t f1, const fmpz_t f2)
```

Return 1 if `f1` is equal to `f2`, otherwise return 0.

```
int fmpz_is_one(const fmpz_t f)
```

Return 1 if `f` is one, otherwise return 0.

```
int fmpz_is_zero(const fmpz_t f)
```

Return 1 if `f` is zero, otherwise return 0.

```
int fmpz_cmpabs(const fmpz_t f1, const fmpz_t f2)
```

Compares the absolute values of `f1` and `f2`. If the absolute value of `f1` is less than that of `f2` then a negative value is returned. If the absolute value of `f1` is greater than that of `f2` then a positive value is returned. If the absolute values are equal, then zero is returned.

8.8 Conversions

```
void mpz_to_fmpz(fmpz_t res, const mpz_t x)
```

Convert the `mpz_t x` to the `fmpz_t res`.

```
void fmpz_to_mpz(mpz_t res, const fmpz_t f)
```

Convert the `fmpz_t f` to the `mpz_t res`.

8.9 Addition/subtraction

```
void fmpz_add(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to the sum of `f1` and `f2`.

```
void fmpz_add_ui_inplace(fmpz_t res, unsigned long x)
```

Set `res` to the sum of `res` and the `unsigned long x`.

```
void fmpz_add_ui(fmpz_t res, const fmpz_t f, unsigned long x)
```

Set `res` to the sum of `f` and the `unsigned long x`.

```
void fmpz_sub(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to `f1` minus `f2`.

```
void fmpz_sub_ui_inplace(fmpz_t res, unsigned long x)
```

Set `res` to `res` minus the `unsigned long x`.

```
void fmpz_sub_ui(fmpz_t res, const fmpz_t f, unsigned long x)
```

Set `res` to `f` minus the `unsigned long x`.

8.10 Multiplication

```
void fmpz_mul(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to `f1` times `f2`.

```
void fmpz_mul_ui(fmpz_t res, const fmpz_t f1, unsigned long x)
```

Set `res` to `f1` times the `unsigned long x`.

```
void fmpz_mul_2exp(fmpz_t output, fmpz_t x, unsigned long exp)
```

Multiply `x` by 2^{exp}.

```
void fmpz_addmul(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to `res + f1 * f2`.

8.11 Division

```
void fmpz_tdiv(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to the quotient of `f1` by `f2`. Round the quotient towards zero and discard the remainder.

```
void fmpz_fdiv(fmpz_t res, const fmpz_t f1, const fmpz_t f2)
```

Set `res` to the quotient of `f1` by `f2`. Round the quotient towards minus infinity and discard the remainder.

```
void fmpz_tdiv_ui(fmpz_t res, const fmpz_t f1, unsigned long x)
```

Set `res` to the quotient of `f1` by the unsigned long `x`. Round the quotient towards zero and discard the remainder.

```
void fmpz_div_2exp(fmpz_t output, fmpz_t x, unsigned long exp)
```

Divide `x` by 2^{exp}, returning the quotient and discarding the remainder.

```
unsigned long fmpz_mod_ui(const fmpz_t input,
                                          const unsigned long x)
```

Returns `f1` modulo the `unsigned long x`. Note that `f1` may be unsigned.

8.12 Powering

```
void fmpz_pow_ui(fmpz_t res, const fmpz_t f, unsigned long exp)
```

Set `res` to `f` raised to the power `exp`. This requires `exp` to be non-negative.

8.13 Root extraction

```
void fmpz_sqrtrem(fmpz_t sqrt, fmpz_t rem, fmpz_t x)
```

Computes the square root of `x` and returns the integer part of the square root, `sqrt`, and the remainder, `rem = x - sqrt^2`.

Note that `x` must be non-negative, else an exception is raised.

8.14 Number theoretical

```
void fmpz_gcd(fmpz_t output, fmpz_t x1, fmpz_t x2)
```

Compute the greatest common divisor of `x1` and `x2`. The result is always non-negative and will be zero if both of the inputs are zero.

8.15 Chinese remaindering

```
void fmpz_CRT_ui_precomp(fmpz_t x, fmpz_t r1, fmpz_t m1,
          unsigned long r2, unsigned long m2, unsigned long c,
                                                pre_inv_t pre)
void fmpz_CRT_ui2_precomp(fmpz_t x, fmpz_t r1, fmpz_t m1,
          unsigned long r2, unsigned long m2, unsigned long c,
                                               pre_inv2_t pre)
```

Computes the unique value `x` modulo `m1*m2` that is `r1` modulo `m1` and `r2` modulo `m2`. Requires `m1` and `m2` to be coprime, `c` to be set to the value `m1` modulo `m2` and `pre` to be a precomputed inverse of `m2` (computed using `z_precompute_inverse(m2)`).

The first version of the function requires that `m2` be no more than `FLINT_D_BITS` bits, whereas the second version requires `m2` to be no more than `FLINT_BITS - 1` bits.

9 The zmod_poly module

The `zmod_poly_t` data type represents elements of $\mathbb{Z}/n\mathbb{Z}[x]$ for some word sized integer n. Most of the functions work for an arbitrary n, however the division functions require the leading coefficient of the divisor polynomial to be invertible modulo n and the gcd and resultant functions require n to be prime.

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

Each coefficient of a `zmod_poly_t` is stored as an `unsigned long` and is assumed to be reduced modulo the modulus n.

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

9.1 Simple example

The following example computes the square of the polynomial $5x^3 + 1$, where the coefficients are understood to be in $\mathbb{Z}/7\mathbb{Z}$.

```
#include "zmod_poly.h"
 ....
zmod_poly_t x, y;
zmod_poly_init(x, 7);
zmod_poly_init(y);
zmod_poly_set_coeff_ui(x, 3, 5);
zmod_poly_set_coeff_ui(x, 0, 1);
zmod_poly_mul(y, x, x);
zmod_poly_print(x); printf("\n");
zmod_poly_print(y); printf("\n");
zmod_poly_clear(x);
zmod_poly_clear(y);
```

The output is:

```
4  1 0 0 5
7  1 0 0 3 0 0 4
```

9.2 Definition of the zmod_poly_t polynomial type

The `zmod_poly_t` type is a typedef for an array of length 1 of `zmod_poly_struct`'s. This permits passing parameters of type `zmod_poly_t` 'by reference'.

All `zmod_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 a `zmod_poly_t` or its coefficient data directly, but make use of the functions designed for this purpose (detailed below). The type has fields for the length of the polynomial, the number of coefficients allocated (the length is always less than or equal to this), a modulus n and possibly a precomputed inverse of n.

Functions in `zmod_poly` do all the memory management for the user. One does not need to specify the maximum length in advance before using a `zmod_poly_t` polynomial object, but it may be more efficient to do so. FLINT reallocates space automatically as the computation proceeds, if more space is required.

We now describe the functions available in `zmod_poly`.

9.3 Memory management

```
void zmod_poly_init(zmod_poly_t poly, unsigned long p)
```

Initialise `poly` as a polynomial over $\mathbb{Z}/p\mathbb{Z}$.

```
void zmod_poly_init2(zmod_poly_t poly, unsigned long p,
                                              unsigned long alloc)
```

Initialise `poly` as a polynomial over $\mathbb{Z}/p\mathbb{Z}$, allocating space for at least the given number of coefficients.

```
void zmod_poly_clear(zmod_poly_t poly)
```

Released the memory used by `poly`, which cannot then be used again until it is initialised again.

```
void zmod_poly_realloc(zmod_poly_t poly, unsigned long alloc)
```

Reallocate `poly` so that it has space for `alloc` coefficients. If alloc is greater than the current length of the polynomial, the existing coefficients are retained.

```
void zmod_poly_fit_length(zmod_poly_t poly, unsigned long alloc)
```

Reallocate `poly` so that it has space for at least `alloc` coefficients. This function will not reduce the number of allocated coefficients, so no data will be lost.

9.4 Setting/retrieving coefficients

```
unsigned long zmod_poly_get_coeff_ui(zmod_poly_t poly,
                                                  unsigned long n)
```

Return the n-th coefficient as an `unsigned long`. Coefficients are number from zero, starting with the constant coefficient. If n is greater than or equal to the current length of the polynomial, zero is returned.

```
void zmod_poly_set_coeff_ui(zmod_poly_t poly, unsigned long n,
                                                  unsigned long c)
```

Set the n-th coefficient to the `unsigned long c`. It is assumed that `c` is already reduced modulo the modulus of the polynomial. Coefficients are number from zero, starting with the constant coefficient. If n is greater than the current length of the polynomial, zeroes are inserted between the new coefficient and the existing coefficients if required.

9.5 String conversions and I/O

The functions in this section read/write a polynomial to/from a string representation. The representation starts with the length of the polynomial, a space and then the modulus of the polynomial. If the length is not zero, this is followed by a space and then a space separated list of the coefficients starting from the constant coefficient. Each coefficient is represented as an integer between zero and one less than the modulus.

The polynomial $3 * x^2 + 2$ in $\mathbb{Z}/7\mathbb{Z}[x]$ would be represented:

```
3 7 2 0 3
```

```
int zmod_poly_from_string(zmod_poly_t poly, char* s)
```

Load `poly` from the given string `s`.

```
char* zmod_poly_to_string(zmod_poly_t poly)
```

Return a pointer to a string representing `poly`. Space is allocated for the string and must be free'd after use.

```
void zmod_poly_print(zmod_poly_t poly)
```

Print the string representation of `poly` to `stdout`.

```
void zmod_poly_fprint(zmod_poly_t poly, FILE* f)
```

Print the string representation of `poly` to the given file/stream `f`.

```
int zmod_poly_read(zmod_poly_t poly)
```

Read a polynomial in string representation from `stdin`. The function returns 1 if the string represented a valid polynomial, otherwise it returns 0.

```
int zmod_poly_fread(zmod_poly_t poly, FILE* f)
```

Read a polynomial in string representation from the given file/stream `f`. The function returns 1 if the string represented a valid polynomial, otherwise it returns 0.

9.6 Polynomial parameters (length, degree, modulus, etc.)

```
unsigned long zmod_poly_length(zmod_poly_t poly)
```

Return the current length of the polynomial. The zero polynomial has length 0.

```
long zmod_poly_degree(zmod_poly_t poly)
```

Return the degree of the polynomial. The zero polynomial is defined to have length -1.

```
unsigned long zmod_poly_modulus(zmod_poly_t poly)
```

Return the modulus of the polynomial, i.e. if n is returned, the polynomial is an element of $\mathbb{Z}/n\mathbb{Z}$.

9.7 Assignment and basic manipulation

```
void zmod_poly_truncate(zmod_poly_t poly, unsigned long length)
```

Truncate `poly` to the given length and normalise.

```
void zmod_poly_set(zmod_poly_t res, zmod_poly_t poly)
```

Set `res` to equal `poly`.

```
void zmod_poly_zero(zmod_poly_t poly)
```

Set `poly` to be the zero polynomial.

```
void zmod_poly_swap(zmod_poly_t poly1, zmod_poly_t poly2)
```

Efficiently swap `poly1` and `poly2`. Data is not actually copied in memory. Instead, pointers are swapped.

```
void zmod_poly_neg(zmod_poly_t res, zmod_poly_t poly)
```

Negate the polynomial `poly`, i.e. set `res` to `-poly`.

```
void zmod_poly_reverse(zmod_poly_t output, zmod_poly_t input,
                                           unsigned long length)
```

Notionally zero padding or truncating if necessary, this function considers `input` to be a polynomial of the given length and reverses it, storing the result in `output`.

9.8 Subpolynomials

These functions allow one to attach a `zmod_poly_t` object to an existing polynomial or subpolynomial thereof. The subpolynomial is normalised if necessary.

Since FLINT cannot reallocate the attached polynomial object, these functions should only be used to construct polynomial objects to be used as inputs to other `zmod_poly` functions.

```
void zmod_poly_attach(zmod_poly_t poly1, zmod_poly_t poly2)
```

Attach `poly2` to the polynomial object `poly1`.

```
void zmod_poly_attach_shift(zmod_poly_t poly1,
                            zmod_poly_t poly2, unsigned long n)
```

This function notionally shifts `poly2` to the right by `n` coefficients and then attaches the polynomial object `poly1` to the result.

```
void zmod_poly_attach_truncate(zmod_poly_t output,
                               zmod_poly_t input, unsigned long n)
```

This function notionally truncates `poly2` to length `n` and then attaches the polynomial object `poly1` to the result.

9.9 Comparison

```
int zmod_poly_equal(zmod_poly_t poly1, zmod_poly_t poly2)
```

Returns 1 if the two polynomials are equal, otherwise returns 0.

```
int zmod_poly_is_one(zmod_poly_t poly1)
```

Returns 1 if the polynomial is equal to the constant polynomial 1, otherwise returns 0.

9.10 Scalar multiplication and division

```
void zmod_poly_scalar_mul(zmod_poly_t res, zmod_poly_t poly,
                                    unsigned long scalar)
```

Multiply the polynomial through by the given scalar. It is assumed that `scalar` is already reduced modulo the modulus of the polynomial.

```
void zmod_poly_make_monic(zmod_poly_t output, zmod_poly_t pol)
```

Divide the polynomial through by the inverse of the leading coefficient of the polynomial. It is assumed that the leading coefficient is invertible modulo the modulus of the polynomial. This function results in a monic polynomial if this condition is met, otherwise the results are undefined.

9.11 Addition/subtraction

```
void zmod_poly_add(zmod_poly_t res, zmod_poly_t poly1,
                                     zmod_poly_t poly2)
```

Set `res` to the sum of `poly1` and `poly2`. Note that if cancellation occurs, `res` may have a lesser length than either of the two input polynomials.

```
void zmod_poly_sub(zmod_poly_t res, zmod_poly_t poly1,
                                     zmod_poly_t poly2)
```

Set `res` to `poly1` minus `poly2`. Note that if cancellation occurs, `res` may have a lesser length than either of the two input polynomials.

9.12 Shifting

```
void zmod_poly_left_shift(zmod_poly_t res, zmod_poly_t poly,
                                        unsigned long k)
```

Shift the polynomial `poly` left by `k` coefficients, i.e. multiply the polynomial by x^k and store the result in `res`. The value of k must be non-negative.

```
void zmod_poly_right_shift(zmod_poly_t res, zmod_poly_t poly,
                                                         unsigned long k)
```

Shift the polynomial `poly` right by `k` coefficients, i.e. divide the polynomial by x^k, ignoring the remainder and store the result in `res`. The value of k must be non-negative. If k is greater than or equal to the current length of `poly`, `res` is set to equal the zero polynomial.

9.13 Polynomial multiplication

```
void zmod_poly_mul(zmod_poly_t res, zmod_poly_t poly1,
                                                       zmod_poly_t poly2)
```

Set `res` to `poly1` multiplied by `poly2`. The length of `res` will be at most one less than the sum of the lengths of `poly1` and `poly2`.

```
void zmod_poly_mul_trunc_n(zmod_poly_t res, zmod_poly_t poly1,
                                   zmod_poly_t poly2, unsigned long n)
```

Set `res` to `poly1` multiplied by `poly2` and truncate to length `n` if this is less than the length of the full product. This function is usually more efficient than simply doing the multiplication and then truncating. The function is tuned for `n` about half the length of a full product. This function is sometimes called a short product.

This function can be used for power series multiplication.

```
void zmod_poly_mul_trunc_left_n(zmod_poly_t res,
           zmod_poly_t poly1, zmod_poly_t poly2, unsigned long n)
```

Set `res` to `poly1` multiplied by `poly2` ignoring the least significant `n` terms of the result which may be set to anything. This function is more efficient than doing the full multiplication if the operands are relatively short. It is tuned for `n` about half the length of a full product. This function is sometimes called an opposite short product.

9.14 Polynomial division

```
void zmod_poly_newton_invert(zmod_poly_t Q_inv, zmod_poly_t Q,
                                                         unsigned long n)
```

Treat the polynomial `Q` as a series of length `n` (the constant coefficient of the series is taken to be the constant coefficient of the polynomial) and invert it, yielding a series `Q_inv` also given to precision `n`.

```
void zmod_poly_div_series(zmod_poly_t Q, zmod_poly_t A,
                                       zmod_poly_t B, unsigned long n)
```

Treat the polynomials `A` and `B` as series of length `n` and compute the quotient series `Q = A/B`.

```
void zmod_poly_div(zmod_poly_t Q, zmod_poly_t A, zmod_poly_t B)
```

Divide the polynomial A by the polynomial B and set Q to the result.

```
void zmod_poly_divrem(zmod_poly_t Q, zmod_poly_t R,
                                    zmod_poly_t A, zmod_poly_t B)
```

Divide the polynomial A by B and set Q to the quotient and R to the remainder.

9.15 Greatest common divisor and resultant

```
unsigned long zmod_poly_resultant(zmod_poly_t a, zmod_poly_t b)
```

Compute the resultant of the polynomials a and b.

If a and b are monic with $a(x) = \prod_i (x - \alpha_i)$ and $b(x) = \prod_j (x - \beta_j)$, when factored over an algebraic closure of the field of coefficients, then the resultant is given by the expression $r(x) = \prod_{i,j} (\alpha_i - \beta_j)$. If the polynomials are not monic, and a and b have leading coefficients l_1 and l_2 and degrees d_1 and d_2 respectively, then this quantity is multiplied by $l_1^{d_2-1} l_2^{d_1-1}$.

Note that the resultant is zero iff the polynomials share a root over an algebraic closure of the coefficient ring.

```
void zmod_poly_gcd(zmod_poly_t res, zmod_poly_t poly1,
                                          zmod_poly_t poly2)
```

Conmpute the greatest common divisor of the polynomials poly1 and poly2.

```
int zmod_poly_gcd_invert(zmod_poly_t res, zmod_poly_t poly1,
                                          zmod_poly_t poly2)
```

Compute a polynomial res such that res*poly1 is a constant modulo poly2. The two polynomials poly1 and poly2 are assumed to be coprime. If this is not the case, the function returns 0, otherwise it returns 1.

```
void zmod_poly_xgcd(zmod_poly_t res, zmod_poly_t s, zmod_poly_t t,
                             zmod_poly_t poly1, zmod_poly_t poly)
```

Compute polynomials s and t such that s*poly1+t*poly2 is the resultant of the polynomials poly1 and poly2. The polynomials poly1 and poly2 are assumed to be coprime.

10 The long_extras module

The `long_extras` module contains functions for doing arithmetic with integers which will fit into an `unsigned long`, including functions for modular arithmetic.

Many of the functions take a precomputed inverse, which increases performance. The functions which include 2 in the name support moduli up to `FLINT_BITS - 1` bits, i.e. 31 or 63 bits, and the remainder work with moduli up to and including `FLINT_D_BITS`.

On 64 bit machines, `FLINT_BITS` is 64 and `FLINT_D_BITS` is 53 bits. On a 32 bit machine the functions with 2 in the name are in fact macros aliasing the corresponding unadorned version. In this case `FLINT_BITS` is 32.

The functions which begin `z_ll_` generally take a parameter consisting of two `unsigned long`'s thought of as an integer of twice the normal size, e.g. on a 64 bit machine these functions would support an input of 128 bits.

Many of the functions in this module can be used to manipulate the individual coefficients of polynomials of type `zmod_poly_t`.

```
pre_inv_t z_precompute_inverse(unsigned long n)

pre_inv2_t z_precompute_inverse2(unsigned long n)

pre_inv_ll_t z_ll_precompute_inverse2(unsigned long n)
```

Return a precomputed inverse of the integer `n`. The first version returns a `pre_inv_t`, which is used with functions taking parameters up to `FLINT_D_BITS`. The second version returns a `pre_inv2_t` for use with function with second versions of functions taking a precomputed inverse, which support parameters up to `FLINT_BITS - 1` bits. The third version returns an inverse suitable for use with `z_ll_` functions which support an operand consisting of two `unsigned long`'s for twice the normal integer precision.

```
unsigned long z_addmod(unsigned long a, unsigned long b,
                                         unsigned long p)
```

Return the sum of `a` and `b` modulo `p`. Both `a` and `b` are assumed to be reduced modulo `p` when calling this function.

```
unsigned long z_submod(unsigned long a, unsigned long b,
                                         unsigned long p)
```

Return `a` minus `b` modulo `p`. Both `a` and `b` are assumed to be reduced modulo `p` when calling this function.

```
unsigned long z_negmod(unsigned long a, unsigned long p)
```

Return minus `a` modulo `p`. The value `a` is assumed to be reduced modulo `p` when calling this function.

```
unsigned long z_div2_precomp(unsigned long a, unsigned long n,
                                                  pre_inv2_t ninv)
```

Return the floor of the quotient of a by n. There are no restrictions on the size of a.

```
unsigned long z_mod_precomp(unsigned long a, unsigned long n,
                                                   pre_inv_t ninv)
```

```
unsigned long z_mod2_precomp(unsigned long a, unsigned long n,
                                                  pre_inv2_t ninv)
```

```
unsigned long z_ll_mod_precomp(unsigned long a_hi,
           unsigned long a_lo, unsigned long n, pre_inv_ll_t ninv)
```

Return a modulo n. The first version assumes that a is less than n^2. The second and third versions replaces no restrictions on a.

```
unsigned long z_mulmod_precomp(unsigned long a, unsigned long b,
                                  unsigned long n, pre_inv_t ninv)
```

```
unsigned long z_mulmod2_precomp(unsigned long a, unsigned long b,
                                 unsigned long n, pre_inv2_t ninv)
```

Return a times b modulo n. The first version assumes that a and b have been reduced modulo n before calling the function. The second version places no restrictions on a and b, i.e. their product may be up to two full limbs.

```
unsigned long z_powmod(unsigned long a, long exp, unsigned long n)
```

```
unsigned long z_powmod2(unsigned long a, long exp, unsigned long n)
```

```
unsigned long z_powmod_precomp(unsigned long a, long exp,
                                  unsigned long n, pre_inv_t ninv)
```

```
unsigned long z_powmod2_precomp(unsigned long a, long exp,
                                 unsigned long n, pre_inv2_t ninv)
```

Raise a to the power exp modulo n. All versions assume a is reduced modulo n, but there are no restrictions on exp, which may be negative (assuming a is invertible modulo n) or zero.

```
int z_jacobi_precomp(unsigned long a, unsigned long p,
                                                   pre_inv_t pinv)
```

Computes the Jacobi symbol of a modulo p for a prime p. Assumes that a is reduced modulo p.

```
unsigned long z_pow(unsigned long a, unsigned long exp)
```

Computes `a` to the power `exp` which must be non-negative. Assumes that the result will fit in an `unsigned long`.

```
unsigned long z_sqrtmod(unsigned long a, unsigned long p)
```

Returns a square root of `a` modulo `p`. Assumes `a` is reduced modulo `p`. The function returns 0 if `a` is not a quadratic residue modulo a prime `p`.

```
unsigned long z_cuberootmod(unsigned long * cuberoot1,
                                   unsigned long a, unsigned long p)
```

Returns a cube root of `a` modulo a prime `p`. Assumes `a` is reduced modulo `p`. If `a` is not 0, the function also sets `cuberoot1` to a cube root of unity modulo `p` if the cube roots of `a` are distinct, otherwise `cuberoot1` is set to 1. If `a` is not a cubic residue modulo `p` the function returns 0.

```
unsigned long z_gcd(long x, long y)
```

Returns the greatest common divisor of `x` and `y`, which may be signed.

```
unsigned long z_invert(unsigned long a, unsigned long n)
```

Returns a multiplicative inverse of `a` modulo `n`. Assumes `a` is reduced modulo `p`.

```
long z_gcd_invert(long* a, long x, long y)
```

Returns the greatest common divisor `d` of `x` and `y` (which may be signed) and sets `a` such that `a*x` is `d` modulo `y`. We ensure `a` is reduced modulo `y`.

```
long z_extgcd(long* a, long* b, long x, long y)
```

Returns the greatest common divisor `d` of `x` and `y` (which may be signed) and sets `a` and `b` such that `d = a*x+b*y`.

```
unsigned long z_CRT(unsigned long x1, unsigned long n1,
                                 unsigned long x2, unsigned long n2)
```

Returns the unique integer `d` reduced modulo `n1*n2` which is `x1` modulo `n1` and `x2` modulo `n2`. Assumes `x1` is reduced modulo `n1` and `x2` is reduced modulo `n2`. Also assumes `n1*n2` is no more than `FLINT_BITS - 1` bits and that `n1` and `n2` are coprime.

```
unsigned long z_randint(unsigned long limit)
```

Returns a random uniformly distributed integer in the range 0 to `limit - 1` inclusive. If `limit` is set to 0, the function returns a full random limb.

```
unsigned long z_randbits(unsigned long bits)
```

Returns a random uniformly distributed integer with (up to) the given number of bits. If `bits` is set to 0, the function returns a full random limb.

11 The mpn_extras module

The `mpn\_extras` module is designed to supplement the low level `mpn` functions provided in GMP. These functions are designed to operate on raw limbs of multiprecision integer data. Each such integer consists of a string of limbs representing an integer, with the least significant limb first. The integers may either be unsigned or signed in twos complement format.

```
void F_mpn_negate(mp_limb_t* dest, mp_limb_t* src,
                                                   unsigned long count)
```

Considering the data at the location `src` to be an integer of `count` limbs stored in twos complement format, this function negates the integer and stores the result at the location `dest`.

```
void F_mpn_copy(mp_limb_t* dest, const mp_limb_t* src,
                                                   unsigned long count)
```

Copy `count` raw limbs at `src` to the location `dest`. Copying begins with the most significant limb first, thus the destination limbs may overlap the source limbs only if `dest > src` in memory.

```
void F_mpn_copy_forward(mp_limb_t* dest, const mp_limb_t* src,
                                                   unsigned long count)
```

Copy `count` raw limbs at `src` to the location `dest`. Copying begins with the least significant limb first, thus the destination limbs may overlap the source limbs only if `dest < src` in memory.

```
void F_mpn_clear(mp_limb_t* dest, unsigned long count)
```

Set all bits of the `count` limbs starting at `dest` to binary zeros.

```
void F_mpn_set(mp_limb_t* dest, unsigned long count)
```

Set all bits of the `count` limbs starting at `dest` to binary ones.

```
pre_limb_t F_mpn_precompute_inverse(mp_limb_t d)
```

Returns a precomputed inverse of `d` for use in `F_mpn` functions which take a `pre_limb_t` precomputed inverse `dinv` of `d`.

One needs to normalise `d` before computing the precomputed inverse, however the original value of `d` itself is passed to the functions. This computation can be done as follows:

```
#include "flint.h"

unsigned long norm;
count_lead_zeros(norm, d);
pre_limb_t xinv = F_mpn_precompute_inverse(x<<norm);
```

```
mp_limb_t F_mpn_divrem_ui_precomp(mp_limb_t * quot,
    mp_limb_t * x, unsigned long xn, mp_limb_t d, pre_limb_t dinv)
```

Compute the quotient of the unsigned multiprecision integer of `xn` limbs at `x` by the limb `d`, placing the quotient at `quot` and returning the remainder. The location `quot` needs space for `count` limbs. The function takes a precomputed inverse of `d`.

```
mp_limb_t F_mpn_mul(mp_limb_t * rn, mp_limb_t * s1p,
             unsigned long s1n, mp_limb_t * s2p, unsigned long s2n)
```

Set `rn` to `s1p*s2p) where \code{s1p` has `s1n` limbs and `s2p` has `s2n` limbs. The number of limbs written is `s1n + s2n`. The most significant limb of the result (which may be zero) is returned by the function.

This function simply calls the GMP `mpn_mul` function for small operands, however for integers of FFT size (larger than about 1300 limbs for multiplication and 1000 limbs for squares) the function is significantly faster than GMP 4.2.2.

```
mp_limb_t F_mpn_mul_trunc(mp_limb_t * rn, mp_limb_t * s1p,
       unsigned long s1n, mp_limb_t * s2p, unsigned long s2n,
                                           unsigned long tn)
```

Set `rn` to `s1p*s2p) where \code{s1p` has `s1n` limbs and `s2p` has `s2n` limbs. The output is truncated to `tn` limbs, where `tn` must be at most `s1n+s2n`. The most significant limb of the result (i.e. limb `tn`) is returned by the function.

The location `rn` must have space for `s1n + s2n` limbs, regardless of the value of `tn`.

This function simply calls the GMP `mpn_mul` function for small operands, however for integers of FFT size the function is significantly faster than GMP 4.2.2. and slightly faster than doing a full multiplication.

```
void F_mpn_mul_precomp_init(F_mpn_precomp_t precomp,
                         mp_limb_t * s1p, unsigned long s1n, s2n)
```

When multiplying a single large integer `s1p` of `s1n` limbs (usually hundreds or more), by many other integers whose maximum size is `s2n` limbs, one can cache the FFT of `s1p` to speed up the multiplications. The precomputed data is attached to an `F_mpn_precomp_t precomp` by this function for use in the functions below.

```
void F_mpn_mul_precomp_clear(F_mpn_precomp_t precomp)
```

Release the memory allocated for the data attached to the F_mpn_precomp_t precomp once it is finished with.

```
mp_limb_t F_mpn_mul_precomp(mp_limb_t * rp, mp_limb_t * s2p,
                        unsigned long s2n, F_mpn_precomp_t precomp)
```

Multiply the integer s2p of s2n limbs by the integer whose FFT has been cached and attached to the F_mpn_precomp_t precomp, computed previously with F_mpn_mul_precomp_init.

The total number of limbs written is s1n + s2n (even if the final limb is zero) where s1n is the size of the integer whose FFT was cached. The most significant limb of the product is returned by the function.

12 NTL interface

Various functions are provided for converting between FLINT objects and NTL objects. To make use of these functions one must type:

#include "NTL-interface.h"

In each case the functions provided for conversion expect the output objects, whether NTL or FLINT objects, to be initialised. However the functions are managed, in that a reallocation automatically occurs if insufficient space was allocated by the user.

```
void ZZ_to_fmpz(fmpz_t output, const ZZ& z)
```

Convert an NTL ZZ integer object to a FLINT fmpz_t integer object.

```
void fmpz_to_ZZ(ZZ& output, const fmpz_t z)
```

Convert a FLINT fmpz_t integer object to an NTL ZZ integer object.

```
void fmpz_poly_to_ZZX(ZZX& output, const fmpz_poly_t poly)
```

Convert an NTL ZZX polynomial object to a FLINT fmpz_poly_t polynomial object.

```
void fmpz_poly_to_ZZX(ZZX& output, const fmpz_poly_t poly)
```

Convert a FLINT fmpz_poly_t polynomial object to an NTL ZZX polynomial object.

13 The quadratic sieve

Currently the quadratic sieve is a standalone program which can be built by typing:

make QS

in the main FLINT directory.

The program is called mpQS. Upon running it, one enters the number to be factored at the prompt.

The quadratic sieve requires that the number entered not be a prime, not be a perfect power and it must not have very small factors. Trial division and the elliptic curve method should be run before making a call to the quadratic sieve, to remove small factors. The sieve may fail silently if the conditions are not met or if the number is too small to be factored by the quadratic sieve (currently about 20 decimal digits or below).

14 Large integer multiplication

In the module mpn_extras and mpz_extras are functions F_mpn_mul and F_mpz_mul respectively which are drop in replacements for GMP's mpn_mul and mpz_mul respectively.

These replacement functions are substantially faster than GMP 4.2.1 when multiplying integers which are thousands of limbs in size. For smaller multiplications these functions call their respective GMP counterparts.