FLINT 1.6: Fast Library for Number Theory

William B. Hart and Andy Novocin
and previously David Harvey

July 29, 2012

Contents

1 Introduction **1**

2 Building and using FLINT **1**

3 Test code **2**

4 Reporting bugs **2**

5 Example programs **2**

6 FLINT macros **3**

7 The fmpz_poly module **3**

7.1 Simple example 4
7.2 Definition of the fmpz_poly_t polynomial type 4
7.3 Initialisation and memory management 4
7.4 Setting/retrieving coefficients 5
7.5 String conversions and I/O 8
7.6 Polynomial parameters (length, degree, max limbs, etc.) 9
7.7 Assignment and basic manipulation 10
7.8 Conversions 11
7.9 Chinese remaindering 11
7.10 Comparison 12
7.11 Shifting 12
7.12 Norms 12
7.13 Addition/subtraction 12
7.14 Scalar multiplication and division 13
7.15 Polynomial multiplication 14

7.16 Polynomial division . . . 15
7.17 Pseudo division . . . 16
7.18 Powering . . . 17
7.19 Gaussian content . . . 17
7.20 Greatest common divisor and resultant . . . 18
7.21 Modular arithmetic . . . 18
7.22 Derivative . . . 18
7.23 Evaluation . . . 19
7.24 Polynomial composition . . . 19
7.25 Polynomial signature . . . 19
7.26 Squarefree . . . 19
7.27 Factorisation . . . 20
7.28 Subpolynomials . . . 20

8 The F_mpz_poly module **21**
8.1 Simple example . . . 22
8.2 Definition of the F_mpz_poly_t polynomial type . . . 22
8.3 Memory management . . . 22
8.4 Normalisation . . . 23
8.5 Subpolynomials . . . 23
8.6 Coefficient operations . . . 24
8.7 Attributes . . . 25
8.8 Truncation . . . 26
8.9 Conversions . . . 26
8.10 String conversions and I/O . . . 27
8.11 Assignment . . . 28
8.12 Comparison . . . 28
8.13 Reverse . . . 28
8.14 Negation . . . 28
8.15 Addition/subtraction . . . 29
8.16 Shifting . . . 29
8.17 Scalar multiplication . . . 29
8.18 Scalar division . . . 30
8.19 Polynomial multiplication . . . 30
8.20 Powering . . . 30
8.21 Polynomial division . . . 30
8.22 Derivative . . . 31
8.23 Content . . . 31
8.24 Evaluation . . . 31
8.25 GCD . . . 32
8.26 Hensel lifting . . . 32
8.27 Factoring . . . 34

9 The fmpz module **35**
9.1 A simple example . . . 35
9.2 Memory management . . . 35
9.3 String operations . . . 36
9.4 fmpz properties . . . 36
9.5 Assignment . . . 36
9.6 Comparison . . . 37
9.7 Conversions . . . 37
9.8 Addition/subtraction . . . 38
9.9 Multiplication . . . 38
9.10 Division . . . 39
9.11 Modular arithmetic . . . 39
9.12 Powering . . . 40
9.13 Root extraction . . . 40
9.14 Number theoretical . . . 40
9.15 Chinese remaindering . . . 40
9.16 Montgomery format . . . 41

10 The F_mpz module **43**
10.1 Simple example . . . 43
10.2 Memory Management . . . 44
10.3 Random generation . . . 44
10.4 Assignment and basic manipulation . . . 44
10.5 Comparison . . . 46
10.6 Properties of integers . . . 47
10.7 Input/output . . . 48
10.8 Basic arithmetic . . . 48
10.9 Addition/subtraction . . . 48
10.10 Multiplication . . . 49
10.11 Division and remainder . . . 50
10.12 Precomputed inverse . . . 51
10.13 Powering . . . 51
10.14 GCD and inverse . . . 51
10.15 Multimodular reduction and CRT . . . 52

11 The zmod_poly module **53**
11.1 Simple example . . . 53
11.2 Definition of the zmod_poly_t polynomial type . . . 53
11.3 Memory management . . . 54
11.4 Setting/retrieving coefficients . . . 54
11.5 String conversions and I/O . . . 55
11.6 Polynomial parameters (length, degree, modulus, etc.) . . . 56
11.7 Assignment and basic manipulation . . . 56
11.8 Subpolynomials . . . 57
11.9 Comparison . . . 57
11.10Scalar multiplication and division . . . 58
11.11Addition/subtraction . . . 58
11.12Shifting . . . 58
11.13Polynomial multiplication . . . 59
11.14Polynomial division . . . 61
11.15Greatest common divisor and resultant . . . 61
11.16Differentiation . . . 62
11.17Arithmetic modulo a polynomial . . . 62
11.18Composition and evaluation . . . 63
11.19Polynomial Factorization . . . 63

12 The F_mpz_mod_poly module **64**
12.1 Simple example . . . 65
12.2 Definition of the F_mpz_mod_poly_t polynomial type . . . 65
12.3 Memory management . . . 66
12.4 Normalisation and truncation . . . 67
12.5 Conversions . . . 67
12.6 Subpolynomials . . . 68
12.7 Reduction . . . 68
12.8 Assignment and swap . . . 69
12.9 Comparison . . . 69
12.10Input/output . . . 69
12.11Shifting . . . 69
12.12Addition/subtraction . . . 70
12.13Scalar multiplication . . . 70
12.14Multiplication . . . 70
12.15Division . . . 70

13 The F_mpz_mat module **71**
13.1 Simple example . . . 71
13.2 Definition of the F_mpz_mat_t matrix type . . . 71
13.3 Memory management . . . 72
13.4 Input/output . . . 72
13.5 Row export . . . 73
13.6 Assignment . . . 74
13.7 Comparison . . . 74
13.8 Negation . . . 74
13.9 Vector memory management . . . 74
13.10 Vector copy and comparison . . . 75
13.11 Addition and subtraction . . . 75
13.12 Scalar multiplication . . . 76
13.13 Scalar addmul/submul . . . 76
13.14 Basic properties . . . 77
13.15 Matrix by scalar operations . . . 78
13.16 Scalar product . . . 78
13.17 Column operations . . . 78
13.18 Matrix windows . . . 78
13.19 Symmetric modulus . . . 79
13.20 Matrix transpose . . . 79

14 The F_mpz_LLL module **79**
14.1 Simple example . . . 79
14.2 Two varieties of LLL . . . 80
14.3 Some special randomized matrices . . . 80

15 The long_extras module **81**
15.1 Precomputed inverses . . . 81
15.2 Fast division . . . 81
15.3 Modulo arithmetic . . . 81
15.4 Powering . . . 82
15.5 Legendre and Jacobi symbols . . . 83
15.6 Primality testing . . . 83
15.7 Roots . . . 84
15.8 GCD and inverses . . . 85
15.9 CRT . . . 85
15.10 Factoring . . . 85
15.11 Random numbers . . . 86

16 The mpn_extras module **86**

17 NTL interface **89**

18 The quadratic sieve **89**

19 Large integer multiplication **90**

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.

As of version 1.1.0 FLINT supports 32 and 64 bit processors including x86, PPC, Alpha and Itanium processors, though in theory it compiles on any machine with GCC version 3.4 or later and with GMP version 5.0 or MPIR 2.0 or later and with MPFR 3.0 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, 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.

FLINT requires GMP version 5.0 or later or MPIR version 2.0 or later (in GMP compatibility mode). Set the environment variables `FLINT_GMP_LIB_DIR` and `FLINT_GMP_INCLUDE_DIR` to point to your GMP or MPIR library and include directories respectively and `FLINT_MPFR_LIB_DIR` and `FLINT_MPFR_INCLUDE_DIR` to point to your MPFR library and include directories respectively. Alternatively you can set default values for these environment variables in the `flint_env` file.

The `NTL-interface` module of FLINT requires NTL version 5.4.1 or later. However NTL is not required to build FLINT if this interface module is not required. To build with NTL set the environment variables `FLINT_NTL_LIB_DIR` and `FLINT_NTL_INCLUDE_DIR` to point to your NTL library and include directories respectively.

Once the environment variables are set or defaults are set in `flint_env` simply 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/MPIR and MPFR with the options `-lflint -lmpfr -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 and run the test programs, simply type:

```
make check
```

in the main FLINT directory.

To test the `NTL-interface` module simply:

```
make NTL-interface-test
```

```
./NTL-interface-test
```

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.

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

Note that FLINT needs to be linked against version 5.0 or later of GMP or version 2.0 or later of MPIR (in GMP compatibility mode) and version 3.0 or later of MPFR and must be compiled with gcc version 3.4 or later. In particular the compiler must be fully C99 compatible.

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 $x^3+y^3+z^3 = k$. This program outputs a file output.log containing parameters for reconstructing the first solution it finds, and then aborts.

`bernoulli_zmod` Compute many bernoulli numbers modulo a prime. If no command line input is supplied it merely checks that the bernoulli_zmod function works for the first 2000 primes. If you specify an integer argument `n` on the command line, it computes the Bernoulli numbers $B_0, B_2, ..., B_{p-1}$ modulo `p`, where `p` is the next prime from `n`.

`expmod` Computes a very large modular exponentiation. This is actually a basic pseudo primality test.

`Zmul` Compares the output of the FLINT FFT with that of GMP for ever larger operands.

`thetaproduct` Computes the congruent number theta function. To run this you need to have OpenMP on your machine, you need a recent version of gcc (4.4.2 or later) and you need to export the environment declaration OMP_NUM_THREADS=16 or some factor of 16, depending on how many cores your machine has. The code also expects a directory /storage with PLENTY of space where temporary files will be created. Be warned that this code multiplies HUGE integers which do not fit into memory and much disk space is used. You also need a significant amount of memory on your machine, which must also be a 64 bit linux platform. Parameters can be changed at the top of the file thetaproduct.c. Primitive (squarefree) zeroes of the congruent number theta function curve will be computed up to MOD∗LIMIT in the class K (mod MOD). At present FILES1 and FILES2 must be equal. LIMIT must also be divisible by BLOCK and by BUNDLE∗FILES1. The code is not currently designed to correctly handle small problems.

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` 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`.

`ulong` is a synonym for `unsigned long`.

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 leading coefficient of the polynomial 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 function, but not as an output. Its contents may be inspected, but not alterered. This function will in general be much faster than the function `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 the 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 white space. 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. If n is greater than or equal to the length of `poly` then `poly` is set to the zero polynomial.

`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`. If `trunc` is not less than the current length of the polynomial, this function does nothing.

```
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`. Note that the supplied `length` may be smaller or larger than the current length of `poly` if required.

`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)
void fmpz_poly_to_zmod_poly_no_red(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`.

If the modulus of `zpol` is p, the `no_red` version of this function assumes that the coefficients of `fmpz_poly_t fpol` are in the range $[-p, p)$ and the computation is done more efficiently.

These functions are 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 $[0, \text{newmod})$ 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 $[-(\text{newmod}-1)/2, \text{newmod}/2]$ 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 greater than or equal to 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 (discarding the remainder) 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 $\mathbb{Q}$.

```
void fmpz_poly_invert_series(fmpz_poly_t Q_inv,
                         const fmpz_poly_t Q, const unsigned long n)
```

Sets `Q_inv` to n terms of the inverse of `Q`. Calling this function is equivalent to calling the function below, `fmpz_poly_div_series`, with `A` equal to 1. Assumes that the constant term of Q is 1.

```
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.

This function can be used for efficient exact division.

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.

```
void fmpz_poly_pseudo_rem(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 $\mathbb{Z}[x]$, without returning the quotient, 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 `R`. This is more efficient than computing the quotient and remainder.

Note that at present this function is not asymptotically fast. Use `fmpz_poly_pseudo_divrem` if large operands will be supplied (e.g. of length greater than 32).

```
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 `lead(B)^d*A = B*Q + R`. However the value `d` is fixed at `A->length - B->length + 1`.

This function is faster when the remainder is not well behaved, i.e. where it is not expected to be zero or close to it. Note that this function is not asymptotically fast. It is efficient only for short polynomials (e.g. `B->length < 32`).

```
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 also notionally computes polynomials `Q` and `R` such that `lead(B)^d*A = B*Q + R`, but returns only `R`. However the value `d` is fixed at `A->length - B->length + 1`.

This function is faster when the remainder is not well behaved, i.e. where it is not expected to be zero or close to it. Note that this function is not asymptotically fast. It is efficient only for short polynomials (e.g. `B->length < 32`).

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.

If `exp` is zero then the result will be the constant polynomial equal to 1, unless `poly` is zero, in which case the output will be zero.

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.

This function is useful for computing inverses in number field arithmetic.

7.22 Derivative

```
void fmpz_poly_derivative(fmpz_poly_t der, fmpz_poly_t poly)
```

Sets `der` to the derivative of `poly`.

7.23 Evaluation

```
void fmpz_poly_evaluate(fmpz_t output,
                        const fmpz_poly_t poly, const fmpz_t val)
```

Evaluates `poly` at the value `val` and sets `output` to the result.

```
unsigned long fmpz_poly_evaluate_mod(const fmpz_poly_t poly,
              unsigned long p, unsigned long val, pre_inv_t pinv)
```

Evaluates `poly` at the value `val` modulo `p` and returns the result. The last argument `pinv` must be set to the precomputed inverse of `p`, which can be obtained using the function `z_precompute_inverse`.

7.24 Polynomial composition

```
void fmpz_poly_compose(fmpz_poly_t output,
                       const fmpz_poly_t f, const fmpz_poly_t g)
```

Sets `output` to the polynomial composition of f with g, i.e. computes $f(g(x))$.

```
void fmpz_poly_translate_mod_horner(zmod_poly_t output,
                       const fmpz_poly_t f, const zmod_poly_t g)
```

Sets `output` to the polynomial composition of f with g where g is of the form $x + c$ for some $c \in \mathbb{Z}_p$ with p the modulus of g, i.e. computes $f(x + c) \mod p$.

7.25 Polynomial signature

```
void fmpz_poly_signature(ulong * r1, ulong * r2, fmpz_poly_t poly)
```

Determines the signature r1, r2 (where r1 + 2*r2 = degree(poly) and r1 is the number of real roots of poly). The input polynomial must be squarefree, otherwise the result is undefined and an exception may be raised. The zero polynomial is allowed, for convenience, and the number of real and complex roots are both set to 0 in that case.

7.26 Squarefree

```
void fmpz_poly_is_squarefree(ulong * r1, ulong * r2, fmpz_poly_t poly)
```

Returns 1 if `poly` is squarefree, otherwise returns 0.

7.27 Factorisation

The fmpz_poly module has a factorisation function. This is a wrapper of the very sophisticated F_mpz_poly_factor function (see below).

A factorisation is returned in an fmpz_poly_factor_t. This is a simple structure to hold an array of irreducible factors and their exponents.

The structure itself contains three user accessible fields, (i) a num_factors field contains the number of irreducible factors, (ii) factors, an array of num_factors irreducible factors, each of which is an fmpz_poly_t and (iii) exponents, an array of corresponding exponents for each irreducible factor. Whilst the user may access these fields, they should not be modified except via the provided functions.

```
void fmpz_poly_factor_init(fmpz_poly_factor_t fac)
```

Initialise an fmpz_poly_factor_t for use.

```
void fmpz_poly_factor_clear(fmpz_poly_factor_t fac)
```

Clear an fmpz_poly_factor_t after use. This releases any memory used by the structure and clears any polynomials it contains.

```
void fmpz_poly_factor_insert(fmpz_poly_factor_t fac,
                         fmpz_poly_t poly, ulong exp)
```

Copy the polynomial poly into the factor structure fac, with the given exponent exp. No attempt is made to merge factors with existing factors. A new slot is created.

```
void fmpz_poly_factor_print(fmpz_poly_factor_t fac)
```

This convenience function prints all the polynomial factors, with their exponents, that are contained in the factor structure.

```
void fmpz_poly_factor(fmpz_poly_factor_t fac, fmpz_poly_t pol)
```

Removes any Gaussian content from pol and discards it (including any factor of -1) and factors the polynomial into irreducible factors, placing the factors with their corresponding exponents into the initialised factor structure. The polynomial being factored may not have length zero.

7.28 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` 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 without affecting the original polynomial.

```
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 without affecting the original polynomial.

8 The F_mpz_poly module

The `F_mpz_poly` module is a new FLINT polynomial module, based on the new `F_mpz_t` integer type (see below). Thus the new `F_mpz_poly_t` data type represents elements of $\mathbb{Z}[x]$. The `F_mpz_poly` module provides routines for memory management, basic arithmetic, and conversions to/from other types.

Each coefficient of an `F_mpz_poly_t` is an integer of the FLINT `F_mpz_t` type. There are two advantages to this model. Firstly, the `F_mpz_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 `F_mpz_poly_t` can be changed without changing the size of any of the other coefficients.

The `F_mpz_poly_t` type will become the main polynomial type in FLINT 2.0 and the `fmpz_poly_t` type will be unavailable. In most cases changing from one module to the other simply means renaming the functions from `fmpz_poly` to `F_mpz_poly`. To ease the transition, FLINT 2.0 will call the new `F_mpz_poly` functions `fmpz_poly`, though there will be some minor differences in naming of some functions in comparison with both the old `fmpz_poly` and `F_mpz_poly` modules. Of course the `F_mpz` functions will also be named `fmpz` in FLINT 2.0 to match.

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

8.1 Simple example

The following example computes the square of the polynomial $5x^3 - 1$. Note the extreme similarity to the example for the `fmpz_poly` module (above).

```
#include "F_mpz_poly.h"
 ....
F_mpz_poly_t x, y;
F_mpz_poly_init(x);
F_mpz_poly_init(y);
F_mpz_poly_set_coeff_ui(x, 3, 5);
F_mpz_poly_set_coeff_si(x, 0, -1);
F_mpz_poly_mul(y, x, x);
F_mpz_poly_print(x); printf("\n");
F_mpz_poly_print(y); printf("\n");
F_mpz_poly_clear(x);
F_mpz_poly_clear(y);
```

The output is:

```
4  -1 0 0 5
7  1 0 0 -10 0 0 25
```

8.2 Definition of the F_mpz_poly_t polynomial type

The `F_mpz_poly_t` type is a typedef for an array of length 1 of `F_mpz_poly_struct`'s. This permits passing parameters of type `F_mpz_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 `F_mpz_poly_t`. For simplicity we will think of an `F_mpz_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 `F_mpz_poly_t` called `poly1` one writes `poly1->length`.

An `F_mpz_poly_t` is said to be *normalised* if either `length == 0`, or if the leading coefficient of the polynomial is nonzero. All `F_mpz_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 `F_mpz_poly_t` or its coefficient data directly, but make use of the functions designed for this purpose (detailed below).

Functions in `F_mpz_poly` do all the memory management for the user. One does not need to specify the maximum length or any coefficient sizes 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 `F_mpz_poly`.

8.3 Memory management

`void F_mpz_poly_init(F_mpz_poly_t poly)`

Initialise a polynomial of length zero with zero allocated coefficients.

```
void F_mpz_poly_init2(F_mpz_poly_t poly, const ulong alloc)
```

Initialise a polynomial of length zero with the given number of allocated coefficients. It is not generally necessary to use this function, however it may be faster if the maximum number of coefficients a polynomial will have is known in advance.

```
void F_mpz_poly_realloc(F_mpz_poly_t poly, const ulong alloc)
```

Reallocates `poly` to have space for precisely the given number of coefficients. If `alloc = 0`, then the polynomial is cleared. If `alloc` is smaller than the current length of the polynomial the polynomial is truncated and normalised. It is generally not necessary to use this function, however it may be faster if the maximum number of coefficients is known in advance, and memory may be saved if a polynomial can be reallocated to a shorter length.

```
void F_mpz_poly_fit_length(F_mpz_poly_t poly, const ulong length)
```

Expands `poly`, if necessary, so that it has space for the given number of coefficients. This function never shrinks the polynomial, only expands it. It is not generally necessary to use this function, but if the maximum length of a polynomial is known in advance, this can save some time in allocations.

```
void F_mpz_poly_clear(F_mpz_poly_t poly)
```

Clear the polynomial `poly`, releasing any memory it was using. The polynomial may not be used again until it is reinitialised.

8.4 Normalisation

```
void _F_mpz_poly_normalise(F_mpz_poly_t poly)
```

Normalise `poly` so that the leading coefficient is nonzero or the polynomial has length zero. This function is mainly used internally and should only be used by the user when manually modifying the internals of a `F_mpz_poly_t`.

8.5 Subpolynomials

```
void _F_mpz_poly_attach(F_mpz_poly_t poly1, const F_mpz_poly_t poly2)
```

Make `poly1` an alias for `poly2`. Note `poly1` must not be reallocated whilst `poly2` is attached to it. Thus `poly1` may not be used as the output of an `F_mpz_poly` function.

```
void _F_mpz_poly_attach_shift(F_mpz_poly_t poly1,
                              const F_mpz_poly_t poly2, const ulong n)
```

Make `poly1` an alias for `poly2`, but starting at the given coefficient, i.e. as though `poly2` had been shifted right by `n`. Note `poly1` must not be reallocated whilst `poly2` is attached to it. Thus `poly1` may not be used as the output of an `F_mpz_poly` function. If `n > poly2->length` then `poly1` will have length zero.

```
void _F_mpz_poly_attach_truncate(F_mpz_poly_t poly1,
                                 const F_mpz_poly_t poly2, const ulong n)
```

Make `poly1` an alias for `poly2`, but as though `poly2` had been truncated to the given number of coefficients `n`. Note `poly1` must not be reallocated whilst `poly2` is attached to it. Thus `poly1` may not be used as the output of an `F_mpz_poly` function. If `n > poly2->length` then `poly1->length` is set to `poly2->length`. The polynomial `poly1` is normalised after truncation.

8.6 Coefficient operations

```
void F_mpz_poly_set_coeff_si(F_mpz_poly_t poly,
                                          ulong n, const long x)
```

Set coefficient `n` of `poly` to the signed `long` value `x`. Coefficients are numbered from the constant coefficient, starting at zero.

```
void F_mpz_poly_set_coeff_ui(F_mpz_poly_t poly,
                                         ulong n, const ulong x)
```

Set coefficient `n` of `poly` to the `ulong` value `x`. Coefficients are numbered from the constant coefficient, starting at zero.

```
void F_mpz_poly_set_coeff_mpz(F_mpz_poly_t poly,
                                         ulong n, const mpz_t x)
```

Set coefficient `n` of `poly` to the `mpz_t` value `x`. Coefficients are numbered from the constant coefficient, starting at zero.

```
void F_mpz_poly_set_coeff(F_mpz_poly_t poly,
                                       ulong n, const F_mpz_t x)
```

Set coefficient `n` of `poly` to the `F_mpz_t` value `x`. Coefficients are numbered from the constant coefficient, starting at zero.

```
long F_mpz_poly_get_coeff_si(const F_mpz_poly_t poly,
                                                 const ulong n)
```

Return coefficient `n` of `poly` as a signed `long`. If `n` is greater than the degree of poly, then zero is returned. The return value is undefined if the coefficient does not fit in a `long`.

```
long F_mpz_poly_get_coeff_ui(const F_mpz_poly_t poly,
                                                  const ulong n)
```

Return coefficient `n` of `poly` as a `ulong`. If `n` is greater than the degree of poly, then zero is returned. The return value is undefined if the coefficient does not fit in a `ulong`.

```
void F_mpz_poly_get_coeff_mpz(mpz_t x,
                        const F_mpz_poly_t poly, const ulong n)
```

Return coefficient `n` of `poly` as an `mpz_t`. If `n` is greater than the degree of poly, then zero is returned.

```
F_mpz * F_mpz_poly_get_coeff_ptr(F_mpz_poly_t poly, ulong n)
```

Return a pointer to coefficient `n` of `poly`. Coefficients are numbered from the constant coefficient, starting at zero. No check is made to verify that `n` is in range. The returned pointer can be used in any function which accepts an `F_mpz_t` as the latter is merely a pointer to an `F_mpz`. Thus coefficients of an `F_mpz_poly_t` can be modified in situ. Note that modifying a coefficient of `poly` may leave the polynomial unnormalised. One should call `_F_mpz_poly_normalise` in this case.

8.7 Attributes

```
long F_mpz_poly_degree(const F_mpz_poly_t poly)
```

Returns the degree of the polynomial `poly`. If the polynomial is zero, then minus one is returned for the degree.

```
ulong F_mpz_poly_length(const F_mpz_poly_t poly)
```

Returns the length of the polynomial `poly`. If the polynomial is zero, then zero is returned for the length.

```
long F_mpz_poly_max_bits1(const F_mpz_poly_t poly)
```

Returns zero if any of the coefficients of `poly` are `mpz_t`'s. Otherwise, computes the largest number of bits `n` that any coefficient has and returns `-n` if a negative coefficient exists in `poly`, else it returns `n`. Zero is returned for the zero polynomial.

```
long F_mpz_poly_max_bits(const F_mpz_poly_t poly)
```

Computes the largest number of bits `n` that any coefficient of `poly` has and returns `-n` if a negative coefficient exists in `poly`, else it returns `n`. Zero is returned for the zero polynomial.

```
ulong F_mpz_poly_max_limbs(const F_mpz_poly_t poly)
```

Returns the largest number of limbs required to store the absolute value of coefficients of `poly`. Zero is returned for the zero polynomial.

8.8 Truncation

```
void _F_mpz_poly_set_length(F_mpz_poly_t poly, const ulong length)
```

In general it is unsafe to set the length of a polynomial by modifying its `length` attribute. However, assuming that it is being set to a length which does not exceed its allocated length (e.g. `F_mpz_poly_fit_length` has been called first), this function will safely set the length of poly to the given length. This function does not normalise the `poly` if truncation occurs. If that functionality is required, call `F_mpz_poly_truncate`, which is always safe.

```
void F_mpz_poly_truncate(F_mpz_poly_t poly, const ulong length)
```

Truncate `poly` to the given length and normalise. If `length` is greater than or equal to the current length of `poly` nothing will happen.

8.9 Conversions

```
void F_mpz_poly_to_fmpz_poly(fmpz_poly_t f_poly,
                                    const F_mpz_poly_t F_poly)
```

Convert an `F_mpz_poly_t F_poly` to an `fmpz_poly_t f_poly`.

```
void fmpz_poly_to_F_mpz_poly(F_mpz_poly_t F_poly,
                                     const fmpz_poly_t f_poly)
```

Convert an `fmpz_poly_t f_poly` to an `F_mpz_poly_t F_poly`.

```
void F_mpz_poly_to_zmod_poly(zmod_poly_t z_poly,
                                    const F_mpz_poly_t F_poly)
```

Convert an `F_mpz_poly_t F_poly` to a `zmod_poly_t z_poly` by reducing modulo the modulus of `z_poly`.

```
void zmod_poly_to_F_mpz_poly(F_mpz_poly_t F_poly,
                                     const zmod_poly_t z_poly)
```

Convert a `zmod_poly_t z_poly` to an `F_mpz_poly_t F_poly` by lifting the coefficients from $\mathbb{Z}/n\mathbb{Z}$ to $\mathbb{Z}$, where `n` is the modulus of `z_poly`.

8.10 String conversions and I/O

The various string formats for polynomials in the `F_mpz_poly` module are identical to those of the `fmpz_poly` module (see the previous section for a description).

```
int F_mpz_poly_from_string(F_mpz_poly_t poly, const char * s)
```

Read a polynomial from a string. The return value will be one if a validly formatted string was read, otherwise zero will be returned.

```
char * F_mpz_poly_to_string(const F_mpz_poly_t poly)
```

Return a string representing `poly` in standard FLINT format.

```
char * F_mpz_poly_to_string_pretty(const F_mpz_poly_t poly,
                                                  const char * x)
```

Return a string representing `poly` in pretty format (see the `fmpz_poly` module for details of the pretty format. The function takes a string representing the variable symbol used to print the polynomial.

```
void F_mpz_poly_fprint(const F_mpz_poly_t poly, FILE * f)
```

Prints the `F_mpz_poly_t poly` to a file stream `f` in standard FLINT format.

```
void F_mpz_poly_fprint_pretty(const F_mpz_poly_t poly,
                                     FILE * f, const char * x)
```

Prints the `F_mpz_poly_t poly` to a file stream `f` in pretty format (see the `fmpz_poly` module for details of the pretty format. The function takes a string representing the variable symbol used to print the polynomial.

```
int F_mpz_poly_fread(F_mpz_poly_t poly, FILE * f)
```

Reads an `F_mpz_poly_t poly` from a file stream `f` in standard FLINT format. If a validly formatted polynomial string is read the function will return one, otherwise it will return zero.

```
void F_mpz_poly_print(F_mpz_poly_t poly)
```

Prints the `F_mpz_poly_t poly` to `stdout` in standard FLINT format. No carriage return or line feed is issued by the function.

8.11 Assignment

```
void F_mpz_poly_zero(F_mpz_poly_t poly)
```

Set `poly` to the zero polynomial.

```
void F_mpz_poly_zero_coeffs(F_mpz_poly_t poly, const ulong n)
```

Zero the first `n` coefficient of `poly`. If `n` is greater than or equal to the current length of `poly`, the polynomial is set to the zero polynomial.

```
void F_mpz_poly_set(F_mpz_poly_t poly1, const F_mpz_poly_t poly2)
```

Set `poly1` to equal `poly2`. Unless `poly1` and `poly2` are the same polynomial this function makes a copy of the data.

```
void F_mpz_poly_swap(F_mpz_poly_t poly1, F_mpz_poly_t poly2)
```

Swap `poly1` and `poly2`. Note that this function efficiently swaps pointers and does not copy any actual data when performing the swap.

8.12 Comparison

```
int F_mpz_poly_equal(const F_mpz_poly_t poly1,
                                  const F_mpz_poly_t poly2)
```

Return one if `poly1` and `poly2` are equal (arithmetically), otherwise returns zero.

8.13 Reverse

```
void F_mpz_poly_reverse(F_mpz_poly_t res,
               const F_mpz_poly_t poly, const ulong length)
```

Treats `poly` as though it were the given length (with leading zeroes if necessary) and sets `res` to the reverse polynomial, i.e. the polynomial of the given length with coefficients in the reverse order to `poly`.

8.14 Negation

```
void F_mpz_poly_neg(F_mpz_poly_t res, const F_mpz_poly_t poly)
```

Set `res` to the negative of `poly`. The zero polynomial remains unchanged.

8.15 Addition/subtraction

```
void F_mpz_poly_add(F_mpz_poly_t res,
           const F_mpz_poly_t poly1, const F_mpz_poly_t poly2)
```

Set `res` to the sum of `poly1` and `poly2`.

```
void F_mpz_poly_sub(F_mpz_poly_t res,
           const F_mpz_poly_t poly1, const F_mpz_poly_t poly2)
```

Set `res` to the difference of `poly1` and `poly2`.

8.16 Shifting

```
void F_mpz_poly_left_shift(F_mpz_poly_t res,
                       const F_mpz_poly_t poly, const ulong n)
```

Multiplies `poly` by x^n and sets `res` to the result.

```
void F_mpz_poly_right_shift(F_mpz_poly_t res,
                       const F_mpz_poly_t poly, const ulong n)
```

Divides `poly` by x^n, discards any remainder and sets `res` to the result.

8.17 Scalar multiplication

```
void F_mpz_poly_scalar_mul_ui(F_mpz_poly_t poly1,
                                  F_mpz_poly_t poly2, ulong x)
```

Multiply `poly2` by the `ulong x` and set `poly1` to the result.

```
void F_mpz_poly_scalar_mul_ui(F_mpz_poly_t poly1,
                                  F_mpz_poly_t poly2, long x)
```

Multiply `poly2` by the signed `long x` and set `poly1` to the result.

```
void F_mpz_poly_scalar_mul(F_mpz_poly_t poly1,
                   const F_mpz_poly_t poly2, const F_mpz_t x)
```

Multiply `poly2` by the `F_mpz_t x` and set `poly1` to the result.

8.18 Scalar division

```
void F_mpz_poly_scalar_divexact(F_mpz_poly_t res,
                                F_mpz_poly_t f, F_mpz_t d)
```

Divides polynomial `f` by the scalar `fmpz_t d`, assuming the division is exact and sets `res` to the resulting quotient.

```
void F_mpz_poly_scalar_smod(F_mpz_poly_t res,
                                F_mpz_poly_t f, F_mpz_t p)
```

Reduce each coefficient of `f` modulo `p` and set `res` to the result, but normalise each coefficient to be in the range `(-p/2, p/2]`, i.e. perform a symmetric modular reduction of the coefficients.

8.19 Polynomial multiplication

```
void F_mpz_poly_mul(F_mpz_poly_t res,
          const F_mpz_poly_t poly1, const F_mpz_poly_t poly2)
```

Multiply `poly1` by `poly2` and set `res` to the result. An attempt is made to choose an optimal algorithm.

```
void F_mpz_poly_mul_trunc_left(F_mpz_poly_t res,
   const F_mpz_poly_t poly1, const F_mpz_poly_t poly2,
                                        const ulong trunc)
```

Multiply `poly1` by `poly2` and set `res` to the result. An attempt is made to choose the optimal algorithm. The lower `trunc` coefficients of `res` will either be correct or set to zero.

8.20 Powering

```
void F_mpz_poly_pow_ui(F_mpz_poly_t res,
                   const F_mpz_poly_t poly1, const ulong exp)
```

Set `res` to `poly1`$^e xp$.

8.21 Polynomial division

```
void F_mpz_poly_divrem(F_mpz_poly_t Q, F_mpz_poly_t R,
                   const F_mpz_poly_t A, const F_mpz_poly_t B)
```

Return polynomials `Q` and `R` such that $B \times Q + R = A$, where each of the coefficients of R beyond coefficient `B->length - 1` are reduced modulo the leading coefficient of `B`. If the division is exact or the leading coefficient of `B` is plus or minus one, the division is as per division over $\mathbb{Q}$ and the length of `R` will be at most `B->length - 1`.

```
void F_mpz_poly_div(F_mpz_poly_t Q,
                    const F_mpz_poly_t A, const F_mpz_poly_t B)
```

Return a polynomial `Q` such that $B \times Q + R = A$ for some polynomial `R` which is not computed and where each of the coefficients of R beyond coefficient `B->length - 1` would be reduced modulo the leading coefficient of `B`. If the division is exact or the leading coefficient of `B` is plus or minus one, the division is as per division over $\mathbb{Q}$ and the length of `R` would be at most `B->length - 1`.

```
void F_mpz_poly_divexact(F_mpz_poly_t Q,
                    const F_mpz_poly_t A, const F_mpz_poly_t B)
```

Return `Q` assuming `A = B\times Q`, i.e. perform a division assuming that it is exact. The result is undefined if the division is not exact.

8.22 Derivative

```
void F_mpz_poly_derivative(F_mpz_poly_t der, F_mpz_poly_t poly)
```

Set `der` to the derivative of `poly`.

8.23 Content

```
void F_mpz_poly_content(F_mpz_t c, const F_mpz_poly_t poly)
```

Set `c` to the content of `poly`.

8.24 Evaluation

```
double F_mpz_poly_eval_horner_d(F_mpz_poly_t poly, double val)
```

Evaluate `poly` at the `double val` and return the result. No cancellation checks are done. The routine is naive and will return an approximate result which will not be anywhere near correct unless there has been no cancellation.

```
double F_mpz_poly_eval_horner_d_2exp(long * exp,
                                     F_mpz_poly_t poly, double val)
```

Computes the evaluation of `poly` at the `double val`, returning the result as a normalised mantissa and an exponent. The result should be extremely close to the correct value, regardless of cancellations, etc. However, it does not guarantee exact rounding (it uses `mpf_t`'s, not `mpfr_t`'s internally). If `d` is the return value of the function, the evaluation is given by $d * 2^{e}xp$. If `d == 0` then `exp` is undefined.

8.25 GCD

```
void F_mpz_poly_gcd(F_mpz_poly_t res,
                                F_mpz_poly_t f, F_mpz_poly_t g)
```

Set `res` to the polynomial GCD of `f` and `g`. At the present moment this is merely a wrapper around the `fmpz_poly` GCD function for convenience.

8.26 Hensel lifting

Let's suppose we have polynomials F, G, H such that $F = GH$. But let's suppose we only know $F, f = F \pmod{p^k} and g = G \pmod{p_0^k}$ for some k_0. We'd like to "lift" this to a solution $\pmod{p^k}$ for some ultimate $k > k_0$, (in the hope that we can actually recover the factors F and G).

FLINT has restartable Hensel lifting, meaning that one can lift something $\pmod{p_0^k}$ to something $\pmod{p_1^k}$ for some $k_1 > k_0$, then at a later stage lift it even further from the polynomials $\pmod{p_1^k}$ up to an even higher power, e.g. to polynomials $\pmod{p^k}$. This can all be done with essentially no loss in time, i.e. as though we had lifted all the way to polynomials $\pmod{p^k}$ in the first place. (There may be some loss, depending on what the powers happen to be, but it is not considered significant compared to lifting again all the way from the start.)

The easiest function to use for Hensel lifting just does all the Hensel lifting in one go, without the option to restart:

```
void F_mpz_poly_hensel_lift_once(F_mpz_poly_factor_t fac,
   F_mpz_poly_t F, zmod_poly_factor_t local_fac, ulong exp)
```

Let `F` be a polynomial whose factors are known modulo a small prime `p` (supplied in `local_fac`). We refer to these factors modulo `p` as "local factors" and require that the product of local factors is squarefree.

This function lifts the local factors all the way up to a specified limit $\pmod{p^{exp}}$ and returns them as polynomials whose coefficients are symmetric mods modulo p^{exp}, i.e. with coefficients in the range $(-p^{exp}/2, p^{exp}/2]$. It is required that `exp > 0`.

The resulting factors are placed in `fac`. We assume `F` has Gaussian content 1.

If one requires restartable Hensel lifting then a "state" needs to be retained. This is a Hensel "tree" and currently it consists of three parts, (i) an array holding permutations, called `link`, (ii) an array of polynomials called `v` and (iii) an array of polynomials called `w`. In future versions of FLINT, these three pieces of state may be encapsulated in a single custom type. For now they need to all be passed separately to the Hensel lifting functions.

In all cases we assume that the number of local factors, `r >= 2`. We also assume `link`, `v` and codew are array with $2r - 2$ entries, and that in the case of the polynomial entries, these have been initialised.

```
ulong _F_mpz_poly_start_hensel_lift(F_mpz_poly_factor_t fac,
        long * link, F_mpz_poly_t * v, F_mpz_poly_t * w,
     F_mpz_poly_t F, zmod_poly_factor_t local_fac, ulong exp)
```

Let `F` be a polynomial whose factors are known modulo a small prime `p` (supplied in `local_fac`). We refer to these factors modulo `p` as "local factors" and require that the product of local factors is squarefree.

This function lifts the local factors up to a specified limit $\pmod{p^{exp}}$ and returns them as polynomials whose coefficients are symmetric mods modulo p^{exp}, i.e. with coefficients in the range $(-p^{exp}/2, p^{exp}/2]$. It is required that `exp > 0`.

The resulting factors are placed in `fac`. We assume `F` has Gaussian content 1.

A Hensel tree `[link, v, w]` is constructed, for restarting this Hensel lift. In addition, an exponent is returned (it gives details about how far Hensel lifting has really been taken internally) and should be passed as `prev_exp` if the Hensel lifting is continued.

```
ulong _F_mpz_poly_continue_hensel_lift(F_mpz_poly_factor_t fac,
 long * link, F_mpz_poly_t * v, F_mpz_poly_t * w, F_mpz_poly_t F,
    ulong prev_exp, ulong current_exp, ulong exp, ulong p, ulong r)
```

This function continues a Hensel lift which has been started by the previous function, or indeed by a previous call to this function at a lower exponent. Three exponents are supplied. The new exponent we'd like to lift to is given by `exp`. The exponent we last lifted to is given by `current_exp`, (this would have been `exp` in the last Hensel lift we did) and `prev_exp` is the exponent returned by the previous Hensel lift we did that we are now continuing.

This function returns a new exponent and an updated Hensel tree `[link, v, w]` in case the current Hensel lift is again restarted.

Note that this function also requires `r`, the number of local factors and `p`, the small prime whose powers we are lifting mod (note that this information is given in lieu of `local_fac`).

The factors at the current exponent are supplied in `fac` and once they have been lifted to the requested `exp`, they are again returned in `fac`.

Note that a Hensel lift can be resumed as many times as one likes, however we always require that `exp > current_exp`.

The Hensel lifting machinery can all be accessed more directly if one requires. The main component of the machinery is a routine which lifts from polynomials $\pmod{p_1^k}$ to at most p^{2k_1}.

```
void F_mpz_poly_hensel_lift(F_mpz_poly_t G, F_mpz_poly_t H,
     F_mpz_poly_t A, F_mpz_poly_t B, F_mpz_poly_t F,
        F_mpz_poly_t g, F_mpz_poly_t h, F_mpz_poly_t a,
           F_mpz_poly_t b, F_mpz_t p1, F_mpz_t p2, F_mpz_t P)
```

We suppose that $F = gh \pmod{p_1}$ where $p_1 = p^{k_1}$ and that we'd like to lift this to $F = GH \pmod{P}$ where $P = p_1 p_2$ where $p_2 = p^{k_2}$ for some $k_2 <= k_1$ (so that $P = p^k$ for some $k <= 2k_1$). In order for this to work, we need to provide polynomials a, b such that $ag + bh = 1 \pmod{p_1}$.

Upon return, A, B will be supplied such that $AG + BH = 1 \pmod{P}$, where $g = G \pmod{p_1}$ and $h = H \pmod{p_1}$.

Note that the powers of the primes, not the exponents are supplied to this and the following two functions.

Under normal circumstances, one would lift all the way from a $p_1 = p$ for a small prime p. Then the same function can be called over and over to get to the final precision. The polynomials a, b can then be initially computed using `zmod_poly_xgcd` and simply converted to `F_mpz_poly`'s, using `zmod_poly_to_F_mpz_poly` for example. Subsequent lifts will then simply use the values of `A, B` supplied by the previous iteration.

As the final lift in a series of lifts will take some time to compute a new G, H, the final lift can be done without this step:

```
void F_mpz_poly_hensel_lift_without_inverse(F_mpz_poly_t G,
 F_mpz_poly_t H, F_mpz_poly_t F, F_mpz_poly_t g, F_mpz_poly_t h,
  F_mpz_poly_t a, F_mpz_poly_t b, F_mpz_t p1, F_mpz_t p2, F_mpz_t P)
```

As per `F_mpz_poly_hensel_lift`, but `A, B` are not computed.

If it is decided that a Hensel lift needs to be continued, but no suitable `A, B` has been computed, the following function computed just `A, B` (usually without significant loss of time versus doing the computation from the start).

```
void F_mpz_poly_hensel_lift_only_inverse(F_mpz_poly_t A,
 F_mpz_poly_t B, F_mpz_poly_t F, F_mpz_poly_t G, F_mpz_poly_t H,
  F_mpz_poly_t a, F_mpz_poly_t b, F_mpz_t p1, F_mpz_t p2, F_mpz_t P)
```

As per `F_mpz_poly_hensel_lift`, but only `A, B` are computed.

8.27 Factoring

```
int F_mpz_poly_is_squarefree(F_mpz_poly_t poly)
```

Return one if `poly` is squarefree, otherwise return zero.

```
void F_mpz_poly_factor_squarefree(F_mpz_poly_factor_t fac,
                           F_mpz_t content, F_mpz_poly_t poly)
```

Factor `poly` into a product of squarefree factors with Gaussian content one and return the Gaussian content of `poly`.

```
void F_mpz_poly_factor_zassenhaus(F_mpz_poly_factor_t fac,
                                  ulong exp, F_mpz_poly_t poly)
```

Given a squarefree polynomial `poly` and an exponent `exp`, this function will factor `poly` using the Zassenhaus algorithm and merge the factors into `fac` raised to the exponent `exp`. The Zassenhaus implementation is not highly optimised and will struggle with more than about ten actual factors, and not too many more local factors.

```
void F_mpz_poly_factor(F_mpz_poly_factor_t fac,
                               F_mpz_t content, F_mpz_poly_t poly)
```

Factor `poly` into irreducible factors with Gaussian content one and return the content of `poly`. No assumptions are made about `poly` except that it be nonzero, and the algorithm is extremely efficient when there are a large number of local factors. It uses an algorithm due to Mark van Hoeij and Andy Novocin.

9 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.

9.1 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.

9.2 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`.

9.3 String operations

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

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

9.4 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`.

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

Return 1 if the sign of f is positive, -1 if it is negative and 0 if f is zero.

9.5 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`.

```
double fmpz_get_d(fmpz_t x)
```

Returns a `double` floating point approximation to the multiprecision integer x. Note that the exponent of a double is limited to strictly less that 1024, thus the absolute value of the integer x must be less than 2^{1024}.

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

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

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

Set the multiprecision integer `res` to the absolute value of the multiprecision integer `f`.

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

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

9.6 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_m1(const fmpz_t f)
```

Return 1 if `f` is minus 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.

9.7 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`.

9.8 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`.

9.9 Multiplication

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

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

```
void fmpz_mul_trunc(fmpz_t res, fmpz_t a,
                                fmpz_t b, unsigned long trunc)
```

Set `res` to `f1` times `f2` truncated to `trunc` limbs. This is in general faster than doing a full multiplication then truncating.

```
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)
```

Set `output` to `x` multiplied by 2^{exp}.

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

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

9.10 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. Rounding occurs towards zero.

`int fmpz_divides(fmpz_t q, const fmpz_t a, const fmpz_t b)`

If b divides a then set q to the quotient and return 1, else return 0.

9.11 Modular arithmetic

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

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

`void fmpz_mod(fmpz_t res, const fmpz_t input, const fmpz_t x)`

Sets `res` to `input` modulo `x`. Note that `input` may be signed but `x` must be unsigned.

`void fmpz_mulmod(fmpz_t res, fmpz_t a, fmpz_t b, fmpz_t m)`

Sets `res` to a multiplied by b modulo m. Note m must be unsigned and both a and b are assumed to be reduced modulo m.

`void fmpz_invert(fmpz_t res, fmpz_t x, fmpz_t m)`

Sets `res` to the inverse of x modulo m. Note m must be unsigned, x and m must be coprime and x reduced modulo m.

`void fmpz_divmod(fmpz_t res, fmpz_t a, fmpz_t b, fmpz_t m)`

Sets `res` to a divided by b modulo m. Note m must be unsigned, b and m must be coprime and both a and b are assumed to be reduced modulo m.

9.12 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.

9.13 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.

9.14 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.

9.15 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.

Multiple modular reductions or Chinese remainders can be done at once with the following functions. An `fmpz_comb_t` type holds information which is used to speed up the modular reductions and modular recombinations. The first two functions are for initialising and clearing such a structure.

```
void fmpz_comb_init(fmpz_comb_t comb, ulong * primes, ulong num_primes)
```

Initialise 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_clear(fmpz_comb_t comb)
```

Clear the given comb structure, releasing any memory it uses.

```
fmpz_t ** fmpz_comb_temp_init(fmpz_comb_t comb)
```

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

```
void fmpz_comb_temp_clear(fmpz_t ** temp, fmpz_comb_t comb)
```

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

```
void fmpz_multi_mod_ui(unsigned long * out,
                fmpz_t in, fmpz_comb_t comb, fmpz_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 array 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,
        unsigned long * residues, fmpz_comb_t comb,
                                    fmpz_t ** comb_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 array 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,
    unsigned long * residues, fmpz_comb_t comb,
                                     fmpz_t ** comb_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 `output` 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`.

9.16 Montgomery format

In this section a number of functions are described which deal with numbers in Montgomery format. In cases where multiple multiplicative functions need to be applied, Montgomery format provides a speed increase over manipulating the integers in ordinary multiprecision format.

```
void fmpz_montgomery_init(fmpz_montgomery_t mont, fmpz_t m)
```

Convert the multiprecision integer to Montgomery format for use with the `fmpz_montgomery_redc` function.

```
void fmpz_montgomery_clear(fmpz_montgomery_t mont)
```

Clear the Montgomery structure, releasing any memory used.

```
void fmpz_montgomery_redc(fmpz_t res, fmpz_t x,
                                        fmpz_montgomery_t mont)
```

Compute the product of `x` and the integer stored in Montgomery format in `mont` and store the result in Montgomery format in `res`.

```
void fmpz_montgomery_mulmod_init(fmpz_montgomery_t mont,
                                            fmpz_t b, fmpz_t m)
```

Compute the Montgomery format of a precomputed multiplication by `b` modulo `m`.

```
void fmpz_montgomery_mulmod(fmpz_t res, fmpz_t a,
                                        fmpz_montgomery_t mont)
```

Compute the product of `a` by `b` modulo `m` where the precomputed data `b` and `m` are stored in the Montgomery structure `mont` by the previous function. Set `res` to the result, which is in ordinary integer format, not Montgomery format.

```
void fmpz_montgomery_divmod_init(fmpz_montgomery_t mont,
                                            fmpz_t b, fmpz_t m)
```

Compute the Montgomery format of a precomputed division by `b` modulo `m`, assuming `b` is coprime with and reduced modulo `m`.

```
void fmpz_montgomery_mulmod(fmpz_t res, fmpz_t a,
                                        fmpz_montgomery_t mont)
```

Compute `a` divided by `b` modulo `m` where the precomputed data `b` and `m` are stored in the Montgomery structure `mont` by the previous function. Set `res` to the result, which is in ordinary integer format, not Montgomery format.

```
void fmpz_montgomery_mod_init(fmpz_montgomery_t mont, fmpz_t m)
```

Compute the Montgomery format for a precomputed reduction modulo `m`.

```
void fmpz_montgomery_mod(fmpz_t res, fmpz_t a,
                                        fmpz_montgomery_t mont)
```

Compute `a` modulo `m` where the precomputed data `m` is stored in the Montgomery structure `mont` by the previous function. Set `res` to the result, which is in ordinary integer format, not Montgomery format.

10 The F_mpz module

The F_mpz module introduces a new FLINT integer format, the F_mpz_t. By default an F_mpz_t is implemented as an array of F_mpz's of length one to allow passing by reference as one can do with GMP/MPIR's mpz_t type. The F_mpz 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, F_mpz_t's act precisely like GMP/MPIR mpz_t's, with automatic memory management, however in the first place only one limb is used to implement them. Once an F_mpz_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, F_mpz_t's will be more efficient than mpz_t's for single limb operations (strictly speaking 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 F_mpz_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 F_mpz_t's. As for mpz_t's there is an underlying type (an F_mpz) which can be used to create the array, e.g.:

```
F_mpz myarr[100];
```

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

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

The F_mpz 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.

10.1 Simple example

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

```
#include "F_mpz.h"
 ....
F_mpz_t x, y;
F_mpz_init(x);
F_mpz_init(y);
F_mpz_set_ui(x, 7);
F_mpz_mul(y, x, x);
F_mpz_print(x);
printf("^2␣=␣");
F_mpz_print(y);
printf("\n");
F_mpz_clear(x);
F_mpz_clear(y);
```

The output is:

```
7^2 = 49
```

We now describe the functions available in the F_mpz module.

10.2 Memory Management

`void F_mpz_init(F_mpz_t f)`

Initialise an `F_mpz_t` for use. It starts as a small `F_mpz_t` (i.e. one not representing an `mpz_t`).

`void F_mpz_init2(F_mpz_t f, ulong limbs)`

Allocate an `F_mpz_t` with the given number of limbs. If limbs is zero then a small `F_mpz_t` results (i.e. not representing an `mpz_t`).

`void F_mpz_clear(F_mpz_t f)`

Clear the given `F_mpz_t`.

`void _F_mpz_cleanup()`

The `F_mpz` module creates a global cache of `F_mpz`'s which get saved by the memory manager for resuse. In order to clean up the cache (say at the end of a program) one calls this function with no arguments. All memory used to allocate such `F_mpz`'s will be freed.

10.3 Random generation

At the present moment the following random generation functions are provided for convenience only. They are not intended to be efficient and their prototypes may change in a later version of FLINT.

`void F_mpz_random(F_mpz_t f, const ulong bits)`

Generate a random `F_mpz_t` with the given number of bits.

`void F_mpz_randomm(F_mpz_t f, const mpz_t n)`

Generate a random `F_mpz_t` in $[0, n)$ where n is an `mpz_t`.

10.4 Assignment and basic manipulation

`void F_mpz_zero(F_mpz_t f)`

Set the given `F_mpz_t` to zero.

`void F_mpz_set_si(F_mpz_t f, const long val)`

Set `f` to a signed long value `val`.

```
void F_mpz_set_ui(F_mpz_t f, const ulong val)
```

Set `f` to an unsigned long value `val`.

```
long F_mpz_get_si(const F_mpz_t f)
```

Return the value of `f` as a long.

```
long F_mpz_get_ui(const F_mpz_t f)
```

Return the value of `f` as an unsigned long.

```
void F_mpz_get_mpz(mpz_t x, const F_mpz_t f)
```

Returns `f` as an `mpz_t`.

```
double F_mpz_get_d_2exp(long * exp, const F_mpz_t f)
```

Return `f` as a signed normalised double and a long exponent.

```
double F_mpz_get_d(const F_mpz_t f)
```

Return `f` as a signed double precision floating point number. The value is truncated (rounded towards zero). The exponent of the double is limited to the range -1023 to 1023 inclusive as usual.

```
void F_mpz_get_mpf(mpf_t m, const F_mpz_t f)
```

Sets `m` to the `mpf_t` value of `f`.

```
void F_mpz_get_mpfr(mpfr_t m, const F_mpz_t f)
```

Sets `m` to the `mpfr_t` value of `f` to the current precision of `m`.

```
void F_mpz_set_d(F_mpz_t f, double d)
```

Sets `f` to the integer part of the double `d`.

```
void F_mpz_set_d_2exp(F_mpz_t f, double d, long exp)
```

Sets `f` to the integer part of $\texttt{d} \times 2^{exp}$.

```
void F_mpz_set_mpz(F_mpz_t f, const mpz_t x)
```

Sets `f` to the given `mpz_t`.

```
void F_mpz_set_mpfr(F_mpz_t f, const mpfr_t x)
```

Sets `f` to the value of the `mpfr_t x` rounded down.

```
int F_mpz_set_mpfr_2exp(F_mpz_t f, const mpfr_t x)
```

Sets `f` to the stored mantissa of the `mpfr_t x` and return an exponent `exp` so that $x = f \times 2^{exp}$.

```
void F_mpz_set_limbs(F_mpz_t f, const mp_limb_t * x, const ulong limbs)
```

Sets `f` to the array of limbs `x` which is the given number of limbs in length and where the least significant limb is stored first in `x`.

```
ulong F_mpz_get_limbs(const mp_limb_t * x, F_mpz_t f)
```

Sets the array of limbs `x` to the absolute value of `f`. The array is assumed to be stored with least significant limb first. The number of limbs written is returned.

```
void F_mpz_set(F_mpz_t f, F_mpz_t g)
```

Sets `f` to the value of `g`.

```
void F_mpz_swap(F_mpz_t f, F_mpz_t g)
```

Efficiently swaps the two `F_mpz_t`'s `f` and `g`.

10.5 Comparison

```
int F_mpz_equal(const F_mpz_t f, const F_mpz_t g)
```

Returns 1 if the values `f` and `g` are equal, otherwise returns 0.

```
int F_mpz_cmpabs(const F_mpz_t f, const F_mpz_t g)
```

Returns a negative value if `abs(f) < abs(g)`, positive if `abs(f) > abs(g)` and returns 0 if the two values are equal.

```
int F_mpz_cmp(const F_mpz_t f, const F_mpz_t g)
```

Returns a negative value if `f < g`, positive if `f > g` and returns 0 if the two values are equal.

```
int F_mpz_is_zero(const F_mpz_t f)
```

Returns 1 if `f` is zero, 0 otherwise.

```
int F_mpz_is_one(const F_mpz_t f)
```

Returns 1 if `f` is one, 0 otherwise.

```
int F_mpz_is_m1(const F_mpz_t f)
```

Returns 1 if `f` is minus one, 0 otherwise.

10.6 Properties of integers

```
ulong F_mpz_size(F_mpz_t f)
```

Returns the number of limbs required to store the absolute value of `f`. Returns 0 if `f` is zero.

```
int F_mpz_sgn(const F_mpz_t f)
```

Returns 1 if `f` is positive, -1 if it is negative and 0 if `f` is zero.

```
ulong F_mpz_bits(F_mpz_t f)
```

Returns the number of bits required to store the absolute value of `f`. Returns 0 if `f` is zero.

```
__mpz_struct * F_mpz_ptr_mpz(F_mpz f)
```

Returns a pointer to the `mpz_t` associated with the coefficient `f`. Assumes `f` is actually associated with an `mpz_t` and not a long. To determine if `g` is actually an `mpz_t` one can use the macro `COEFF_IS_MPZ(*g)`.

Users generally do not need to use this function and it is mainly used internally by FLINT. However it can be useful when one wishes to read an `F_mpz_t` as an `mpz_t` without making a copy of the data.

If `g` is an `F_mpz_t` one must first dereference it before passing it to this function.

To get the value of `g` as a long when it is not associated with an `mpz_t` simply dereference `g`, i.e. the value is given by `*g`.

10.7 Input/output

`void F_mpz_print(F_mpz_t x)`

Print the given `F_mpz_t` to stdout.

`void F_mpz_read(F_mpz_t x)`

Read an `F_mpz_t` from stdin. The integer can be a signed multiprecision integer in decimal format.

`void F_mpz_sscanf(F_mpz_t x, char * str)`

Read an `F_mpz_t` from the given string. The integer can be a signed multiprecision integer in decimal format.

10.8 Basic arithmetic

`void F_mpz_neg(F_mpz_t f, F_mpz_t g)`

Set `f` to minus `g`.

`void F_mpz_abs(F_mpz_t f, F_mpz_t g)`

Set `f` to the absolute value of `g`.

10.9 Addition/subtraction

`void F_mpz_add_ui(F_mpz_t f, const F_mpz_t g, const ulong x)`

Add the unsigned long `x` to `g` and set `f` to the result.

`void F_mpz_sub_ui(F_mpz_t f, const F_mpz_t g, const ulong x)`

Subtract the unsigned long `x` from `g` and set `f` to the result.

`void F_mpz_add_mpz(F_mpz_t f, const F_mpz_t g, mpz_t h)`

Set `f` to `g` plus `h`, where `h` is an `mpz_t`.

`void F_mpz_add(F_mpz_t f, const F_mpz_t g, F_mpz_t h)`

Set `f` to `g` plus `h`.

`void F_mpz_sub(F_mpz_t f, const F_mpz_t g, F_mpz_t h)`

Set `f` to `g` minus `h`.

10.10 Multiplication

`void F_mpz_mul_ui(F_mpz_t f, const F_mpz_t g, const ulong x)`

Multiply `g` by the unsigned long `x` and set `f` to the result.

`void F_mpz_mul_si(F_mpz_t f, const F_mpz_t g, const long x)`

Multiply `g` by the signed long `x` and set `f` to the result.

`void F_mpz_mul2(F_mpz_t f, const F_mpz_t g, const F_mpz_t h)`

Multiply `g` by `h` and set `f` to the result. The function is called `mul2` rather than `mul` due to a conflict in naming with the `mpn_extras` module in FLINT. This conflict will be removed in a later version of FLINT.

`void F_mpz_mul_2exp(F_mpz_t f, const F_mpz_t g, const ulong exp)`

Multiply `g` by `2^exp` and set `f` to the result.

`void F_mpz_addmul_ui(F_mpz_t f, const F_mpz_t g, const ulong x)`

Multiply `g` by the unsigned long `x` and add the result to `f`, in place.

`void F_mpz_submul_ui(F_mpz_t f, const F_mpz_t g, const ulong x)`

Multiply `g` by the unsigned long `x` and subtract the result from `f`, in place.

`void F_mpz_addmul(F_mpz_t f, const F_mpz_t g, const F_mpz_t h)`

Multiply `g` by `h` and add the result to `f`, in place.

`void F_mpz_submul(F_mpz_t f, const F_mpz_t g, const F_mpz_t h)`

Multiply `g` by `h` and subtract the result from `f`, in place.

10.11 Division and remainder

```
void F_mpz_div_2exp(F_mpz_t f, const F_mpz_t g, const ulong exp)
```

Divide `g` by `2^exp` and set `f` to the result. Rounding is towards zero.

```
void F_mpz_mod_ui(F_mpz_t f, const F_mpz_t g, const ulong h)
```

Set `f` to `g` modulo `h`.

```
void F_mpz_mod(F_mpz_t f, const F_mpz_t g, const F_mpz_t h)
```

Set `f` to `g` modulo `h`.

```
void F_mpz_smod(F_mpz_t f, const F_mpz_t g, const F_mpz_t h)
```

Set `f` to the symmetric modulus of `g` modulo `h`, i.e. `f` modulo `g` with values chosen in the range `(-g/2, g/2]`..

```
void F_mpz_divexact(F_mpz_t f, const F_mpz_t g, const F_mpz_t h)
```

Set `f` to `g` divided by `h`, assuming the division is exact.

```
void F_mpz_fdiv_q(F_mpz_t f, const F_mpz_t g, const F_mpz_t h)
```

Set `f` to `g` divided by `h`, rounded down towards minus infinity.

```
void F_mpz_fdiv_qr(F_mpz_t q, F_mpz_t r,
                              const F_mpz_t g, const F_mpz_t h)
```

Set `q` to `g` divided by `h`, rounded down towards minus infinity and set `r` to the remainder.

```
void F_mpz_cdiv_q(F_mpz_t f, const F_mpz_t g, const F_mpz_t h)
```

Set `f` to `g` divided by `h`, rounded up towards infinity.

```
void F_mpz_rdiv_q(F_mpz_t f, const F_mpz_t g, const F_mpz_t h)
```

Set `f` to `g` divided by `h`, rounded to nearest, ties rounded towards positive infinity.

10.12 Precomputed inverse

```
mp_ptr F_mpz_precompute_inverse(F_mpz_t p)
```

Returns a pointer to a precomputed inverse of `p` for use with preinv functions. The value `p` must be positive.

```
void F_mpz_mod_preinv(F_mpz_t f,
       const F_mpz_t g, const F_mpz_t p, mp_srcptr pinv)
```

Set `f` to `g` modulo `p` given a precomputed inverse `pinv`.

```
void F_mpz_smod_preinv(F_mpz_t f,
        const F_mpz_t g, const F_mpz_t h, mp_srcptr pinv)
```

Set `f` to the symmetric modulus of `g` modulo `h` given a precomputed inverse `pinv`, i.e. `f` modulo `g` with values chosen in the range `(-g/2, g/2]`..

```
void F_mpz_preinv_clear(mp_ptr pinv)
```

Free the memory allocated for a precomputed inverse.

10.13 Powering

```
void F_mpz_pow_ui(F_mpz_t f, const F_mpz_t g, const ulong exp)
```

Set `f` to `g` to the power `exp`. If 0 is raised to the power 0, the result will be 1.

10.14 GCD and inverse

```
void F_mpz_gcd(F_mpz_t f, const F_mpz_t g, const F_mpz_t h)
```

Set `f` to the greatest common divisor of `g` and `h`. If both inputs are negative, the output will be negative.

```
int F_mpz_invert(F_mpz_t f, const F_mpz_t g, const F_mpz_t h)
```

Set `f` to the inverse of `g` modulo `|h|` if it exists and return 1. If the inverse does not exist, return 0. We normalise with $0 \leq f < |h|$.

10.15 Multimodular reduction and CRT

```
void F_mpz_comb_init(F_mpz_comb_t comb, ulong * primes, ulong num_primes)
```

Initialise a comb for use with multimodular reduction and CRT. The array `primes` is a list of `num_primes` primes which will be used in the multimodular reduction and CRT.

```
void F_mpz_comb_clear(F_mpz_comb_t comb)
```

Free a comb, releasing any memory allocated for it.

```
F_mpz ** F_mpz_comb_temp_init(F_mpz_comb_t comb)
```

Allocate and initialise temporary space for use with a comb, required by the multimodular reduction and CRT routines.

```
void F_mpz_comb_temp_free(F_mpz_comb_t comb, F_mpz ** comb_temp)
```

Free temporary space allocated for use with a comb.

```
void F_mpz_multi_mod_ui(ulong * out, F_mpz_t f,
                  F_mpz_comb_t comb, F_mpz ** comb_temp, F_mpz_t temp)
```

Reduce the `F_mpz_t f` modulo each of the primes associated with the comb and output the residues in an array. Requires a comb, temporary space allocated by the function above and a temporary `F_mpz_t` to be passed in.

```
void F_mpz_multi_CRT_ui_unsigned(F_mpz_t output, ulong * residues,
    F_mpz_comb_t comb, F_mpz ** comb_temp, F_mpz_t temp, F_mpz_t temp2)
```

Given an array of residues mod primes associated with a comb, compute an integer which has those residues mod those primes. Requires a comb, temporary space for a comb, as allocated by the function above, and two temporary `F_mpz_t`'s. The output integer is assumed to be unsigned.

```
void F_mpz_multi_CRT_ui(F_mpz_t output, ulong * residues,
    F_mpz_comb_t comb, F_mpz ** comb_temp, F_mpz_t temp, F_mpz_t temp2)
```

Given an array of residues mod primes associated with a comb, compute an integer which has those residues mod those primes. Requires a comb, temporary space for a comb, as allocated by the function above, and two temporary `F_mpz_t`'s. The output integer is assumed to be signed.

11 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 factoring, 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 return polynomials whose coefficients are reduced modulo n.

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

11.1 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
```

11.2 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. Data is also stored for manipulation of the

polynomials by `zn_poly` which is included in FLINT for efficient computation with polynomials in this module.

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`.

11.3 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)
```

Release the memory used by `poly`, which cannot then be used 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, otherwise the polynomial is truncated and normalised.

```
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.

11.4 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 numbered 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 numbered 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.

11.5 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 two spaces 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.

11.6 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 $\mathbb{Z}/n\mathbb{Z}[x]$.

`unsigned long zmod_poly_bits(zmod_poly_t poly)`

Return the maximum number of bits used in the coefficients of `poly`, i.e. if n is returned, then no coefficient of the polynomial uses more than n bits.

11.7 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`.

```
void __zmod_poly_normalise(zmod_poly_t poly)
```

Normalises the given polynomial. The polynomial will then either be of length zero or its leading coefficient will be non-zero. As all functions in the `zmod_poly` module expect and return normalised polynomials, this function is only used when manipulating coefficients directly rather than through the functions provided.

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

```
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 poly1,
                             zmod_poly_t poly2, unsigned long n)
```

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

11.9 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.

```
int zmod_poly_is_zero(zmod_poly_t poly1)
```

Returns 1 if the polynomial is the zero polynomial, otherwise returns 0.

11.10 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 result is undefined.

11.11 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.

11.12 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 the zero polynomial.

11.13 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 `poly1->length + poly2->length - 1`.

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

Set `res` to `poly` squared. The length of `res` will be `2*poly->length - 1`.

```
void zmod_poly_mul_precache_init(zmod_poly_precache_t pre,
               zmod_poly_t poly2, unsigned long bits_input,
                                  unsigned long length1)
```

This function precaches an FFT of the polynomial `input2` for (usually multiple) subsequent multiplications by the polynomial `input2`, with up to the given number of bits per output coefficient (0 if this is to be computed automatically). One must set `length1` to the maximum length of any polynomials `poly1` that `poly2` will be multiplied by.

```
void zmod_poly_mul_precache(zmod_poly_t output,
               zmod_poly_t poly1, zmod_poly_precache_t pre)
```

Multiply the polynomial `poly1` by the polynomial whose precached FFT has been stored in `pre` by `zmod_poly_mul_precache_init`, i.e. sets `output` to the product of `poly1` by `poly2`.

```
void zmod_poly_mul_precache_clear(zmod_poly_precache_t pre)
```

Free any memory used by the `zmod_poly_mul_precache_t pre`.

```
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.

```
void zmod_poly_mul_trunc_n_precache_init(zmod_poly_precache_t pre,
      zmod_poly_t poly2, unsigned long bits, unsigned long trunc)
```

This function precaches an FFT of a polynomial `poly2` to be used (usually multiple times) for truncated multiplications by `input2`, with up to the given number of bits per output coefficient (0 if this is to be computed automatically), where the output will be truncated to the given length.

This function is also used for initialising a precached middle product.

```
void zmod_poly_mul_trunc_n_precache(zmod_poly_t output,
  zmod_poly_t poly1, zmod_poly_precache_t pre, unsigned long trunc)
```

Performs a truncated multiplication by a polynomial whose FFT has been precached using `zmod_poly_mul_trunc_n_precache_init`, i.e. `output` is set to `poly1` multiplied by `poly2` and truncated to length `trunc` (and normalised).

```
void zmod_poly_mul_middle(zmod_poly_t output,
               zmod_poly_t poly1, zmod_poly_t poly22,
                                  unsigned long trunc)
```

Performs a middle product of the polynomial `poly1` by the polynomial `poly2`.

The middle product is the product of `poly1` by `poly2` truncated to length `trunc` and with the first `trunc/2` coefficients set to zero. Note that for this function to return a correct result one must ensure that if the full product were wrapped around after the first `trunc` terms then no more than `trunc/2` terms would be affected by the wraparound.

The typical situation to apply this function is when multiplying a polynomial of length $2n$ by one of length n. Ordinarily the product would have $3n - 1$ terms, however if `trunc` is set to $2n$ the first n terms will be set to zero and the product truncated at $2n$ terms. Note that $n - 1$ terms would be wrapped around and $n - 1$ is less than the n terms that will be set to zero.

```
void zmod_poly_mul_middle_precache(zmod_poly_t output,
               zmod_poly_t poly1, zmod_poly_precache_t pre,
                                        unsigned long trunc)
```

Performs a middle product of the polynomial `poly1` by the precached polynomial `poly2` stored in `pre` by the function `zmod_poly_mul_trunc_n_precache_init`.

The middle product is the product of `poly1` by `poly2` truncated to length `trunc` with the first `trunc/2` coefficients set to zero. Note that for this function to return a correct result one must ensure that if the full product were wrapped around after the first `trunc` terms then no more than `trunc/2` terms would be affected by the wraparound.

The typical situation to apply this function is when multiplying a polynomial of length $2n$ by one of length n. Ordinarily the product would have $3n - 1$ terms, however if `trunc` is set to $2n$ the first n terms will be set to zero and the product truncated at $2n$ terms.

11.14 Polynomial division

```
void zmod_poly_invert_series(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, which must be invertible modulo the modulus of `Q`) 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_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. The leading coefficient of `B` must be invertible modulo the modulus of `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 quotient. The leading coefficient of `B` must be invertible modulo the modulus of `B`. This function is slightly faster than computing the quotient and remainder as per `zmod_poly_divrem`.

```
void zmod_poly_rem(zmod_poly_t R, zmod_poly_t A, zmod_poly_t B)
```

Divide the polynomial `A` by `B` and set `R` to the remainder. The leading coefficient of `B` must be invertible modulo the modulus of `B`. This function is more efficient than computing the quotient and remainder as per `zmod_poly_divrem`.

11.15 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)
```

Compute the greatest common divisor of the polynomials `poly1` and `poly2`. The result that is returned will be monic.

```
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 1 modulo `poly2`. The two polynomials `poly1` and `poly2` are assumed to be coprime. If this is not the case, the function returns 0 and the result is undefined, 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. The resultant that is returned will be monic.

11.16 Differentiation

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

Set `res` equal to the derivative of `poly` and reduce all the coefficients modulo the modulus of `poly`.

11.17 Arithmetic modulo a polynomial

```
void zmod_poly_mulmod(zmod_poly_t res, zmod_poly_t poly1,
                           zmod_poly_t poly2, zmod_poly_t f)
```

Set `res` equal to the product of `poly1` and `poly2` modulo `f`. Assumes that `poly1` and `poly2` are reduced modulo `f`.

```
void zmod_poly_powmod(zmod_poly_t res, zmod_poly_t pol,
                                    long exp, zmod_poly_t f)
```

Sets `res` equal to `pol` raised to the power `exp` modulo `f`. Assumes `pol` is reduced modulo `f`. There are no restrictions on `exp`, i.e. it can be zero, positive or negative. The leading coefficient of `f` must be invertible modulo the modulus.

11.18 Composition and evaluation

```
ulong zmod_poly_evaluate(zmod_poly_t poly, ulong c)
```

Evaluate the polynomial `poly` at the value `c` and return the result. It is assumed that `c` is already reduced modulo the modulus of `poly`.

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

Compute the composition `poly1(poly2(x))` and set `res` to the result.

11.19 Polynomial Factorization

```
void zmod_poly_factor_init(zmod_poly_factor_t fac)
```

Initializes an array for storing factors resulting from a factorisation.

```
void zmod_poly_factor_clear(zmod_poly_factor_t fac)
```

Clear an array of factors, releasing any memory used by the struct.

```
void zmod_poly_factor_add(zmod_poly_factor_t fac,
                          zmod_poly_t poly)
```

Adds an extra element, `poly`, to the array of factors, `fac`.

```
void zmod_poly_factor_concat(zmod_poly_factor_t res,
                             zmod_poly_factor_t fac)
```

Concatenates the two arrays, `res` and `fac`, into a single array of factors, `res`.

```
void zmod_poly_factor_print(zmod_poly_factor_t fac)
```

Prints to stdout each factor in the array `fac` each with their corresponding exponent.

```
void zmod_poly_factor_pow(zmod_poly_factor_t fac,
                                  unsigned long exp)
```

Raises each factor in the array `fac` to the power `exp`.

```
ulong zmod_poly_deflation(const zmod_poly_t f)
```

Returns the maximum positive integer s such that $f(x) = g(x^s)$ for some polynomial $g(x)$.

```
void zmod_poly_deflate(zmod_poly_t g, const zmod_poly_t f, ulong d)
```

Computes $f(x) = f(x^d)$.

```
void zmod_poly_inflate(zmod_poly_t g, const zmod_poly_t f, ulong d)
```

Computes $f(x) = f(x^d)$.

```
void zmod_poly_factor_square_free(zmod_poly_factor_t res,
                                  zmod_poly_t f)
```

Sets `res` to a square-free factorization of `f`.

```
void zmod_poly_factor_berlekamp(zmod_poly_factor_t factors,
                                zmod_poly_t f)
```

Performs the Berlekamp factoring algorithm on `f`. Sets `factors` to the factors of `f`. Assumes `f` is squarefree.

```
unsigned long zmod_poly_factor(zmod_poly_factor_t result,
                               zmod_poly_t input)
```

Sets `result` to be a complete factorization of `input`. There are no restrictions on `input`.

```
int zmod_poly_isirreducible(zmod_poly_t f)
```

Returns 1 if the polynomial f is irreducible, otherwise it returns 0.

12 The F_mpz_mod_poly module

The `F_mpz_mod_poly` module is a new FLINT polynomial module, based on the new `F_mpz_t` integer type for polynomials over $\mathbb{Z}/P\mathbb{Z}$ for a multiprecision modulus `P`. Thus the new `F_mpz_mod_poly_t` data type represents elements of $\mathbb{Z}/P\mathbb{Z}[x]$.

Each coefficient of an `F_mpz_mod_poly_t` is an integer of the FLINT `F_mpz_t` type, which is reduced modulo `P`.

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

12.1 Simple example

The following example computes the square of the polynomial $5x^3 + 1 \pmod 7$.

```
#include "F_mpz_poly.h"
#include "F_mpz_mod_poly.h"
 ....
F_mpz_t P;
F_mpz_mod_poly_t x, y;
F_mpz_poly u, v;
F_mpz_init(P);
F_mpz_poly_init(u);
F_mpz_set_ui(P, 7);
F_mpz_mod_poly_init(x, P);
F_mpz_mod_poly_init(y, P);
F_mpz_poly_set_coeff_ui(u, 3, 5);
F_mpz_poly_set_coeff_ui(u, 0, 1);
F_mpz_poly_to_F_mpz_mod_poly(x, u);
F_mpz_mod_poly_mul(y, x, x);
F_mpz_mod_poly_print(x); printf("\n");
F_mpz_mod_poly_print(y); printf("\n");
F_mpz_clear(P);
F_mpz_poly_clear(u);
F_mpz_mod_poly_clear(x);
F_mpz_mod_poly_clear(y);
```

The output is:

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

12.2 Definition of the F_mpz_mod_poly_t polynomial type

The `F_mpz_mod_poly_t` type is a typedef for an array of length 1 of `F_mpz_mod_poly_struct`'s. This permits passing parameters of type `F_mpz_mod_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 `F_mpz_mod_poly_t`. For simplicity we will think of an `F_mpz_mod_poly_t` as a struct, though in practice to access fields of this struct, one needs to dereference first, e.g. to access the `length` field of an `F_mpz_mod_poly_t` called `poly1` one writes `poly1->length`.

An `F_mpz_mod_poly_t` is said to be *normalised* if either `length == 0`, or if the leading coefficient of the polynomial is nonzero. All `F_mpz_mod_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 `F_mpz_mod_poly_t` or its coefficient data directly, but make use of the functions designed for this purpose. At present, this is most easily achieved by converting to and from `F_mpz_poly_t`'s.

Functions in `F_mpz_mod_poly` do all the memory management for the user. One does not need to specify the maximum length or any coefficient sizes 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 `F_mpz_mod_poly`.

12.3 Memory management

```
void F_mpz_mod_poly_init(F_mpz_mod_poly_t poly,
                                    const F_mpz_t P)
```

Given a positive modulus `P`, initialises an `F_mpz_mod_poly_t` object for use.

```
void F_mpz_mod_poly_init2(F_mpz_mod_poly_t poly,
                          const F_mpz_t P, ulong alloc)
```

Given a positive modulus `P`, initialises an `F_mpz_mod_poly_t` object for use, ensuring that it has space for at least `alloc` coefficients. It is not necessary to use this function in practice, as `F_npz_mod_poly` functions automatically resize their outputs as required. However, if the maximum length of a polynomial is known in advance, it may be possible to save some time on memory allocation.

```
void F_mpz_mod_poly_clear(F_mpz_mod_poly_t poly)
```

Clear an `F_mpz_mod_poly_t` object, releasing any memory it was using.

```
void F_mpz_mod_poly_realloc(F_mpz_mod_poly_t poly,
                                  const ulong alloc)
```

Reallocate an `F_mpz_mod_poly_t poly` so that it has space for `alloc` coefficients. It is not necessary to use this function in practice, however, if the maximum length of a polynomial is known in advance, it may be possible to save some time on memory allocation. If the current length of `poly` is greater than `alloc` the extra data will be lost.

```
void F_mpz_mod_poly_fit_length(F_mpz_mod_poly_t poly,
                                  const ulong length)
```

Reallocate an `F_mpz_mod_poly_t poly` so that it has space for at least `alloc` coefficients. It is not necessary to use this function in practice, however, if the maximum length of a polynomial is known in advance, it may be possible to save some time on memory allocation. This function will never shrink a polynomial, so data cannot be lost.

12.4 Normalisation and truncation

```
void _F_mpz_mod_poly_normalise(F_mpz_mod_poly_t poly)
```

By definition an `F_mpz_mod_poly_t poly` is normalised if it is length zero or if the coefficient at index `poly->length - 1` is nonzero. This function normalises a polynomial. All functions in the `F_mpz_mod_poly` module normalise their outputs, so this function is mainly used internally and if the user modifies the internals of an `F_mpz_mod_poly_t` directly.

```
void _F_mpz_mod_poly_set_length(F_mpz_mod_poly_t poly,
                                const ulong length)
```

This function provides a safe way to set the length of a polynomial. If the current polynomial length is greater than `length` the additional coefficients are released and set to zero. Note that this function does not normalise the result and assumes that the polynomial already has space allocated for at least `length` coefficients. This function is mainly used internally.

```
void F_mpz_mod_poly_truncate(F_mpz_mod_poly_t poly,
                             const ulong length)
```

This function truncates `poly` to the given length and normalises the result. If `length` is greater than or equal to the current polynomial length, nothing happens.

12.5 Conversions

```
void F_mpz_poly_to_F_mpz_mod_poly(F_mpz_mod_poly_t F_poly,
                                  const F_mpz_poly_t poly)
```

Convert from an `F_mpz_poly_t poly` to an `F_mpz_mod_poly_t F_poly`. This function does not attempt to reduce the coefficients modulo the modulus `F_poly->P`. This can be achieved with the function `_F_mpz_mod_poly_reduce_coeffs` (see below).

```
void F_mpz_mod_poly_to_F_mpz_poly(F_mpz_poly_t poly,
                            const F_mpz_mod_poly_t F_poly)
```

Convert from an `F_mpz_mod_poly_t F_poly` to an `F_mpz_poly_t poly`.

```
void zmod_poly_to_F_mpz_mod_poly(F_mpz_mod_poly_t fpol,
                                 const zmod_poly_t zpol)
```

Convert from a `zmod_poly_t zpol` to an `F_mpz_mod_poly_t fpol`. This function does not check that the modulus `zpol->p` matches the modulus `fpol->P`.

```
void F_mpz_mod_poly_to_zmod_poly(zmod_poly_t zpol,
                            const F_mpz_mod_poly_t fpol)
```

Convert from an `F_mpz_mod_poly_t fpol` to a `zmod_poly_t zpol`. This function does not check that the modulus `zpol->p` matches the modulus `fpol->P`.

12.6 Subpolynomials

```
void _F_mpz_poly_attach_F_mpz_mod_poly(F_mpz_poly_t out,
                                const F_mpz_mod_poly_t in)
```

Attach a temporary `F_mpz_poly_t` object to an existing `F_mpz_mod_poly_t` object. The `F_mpz_mod_poly_t` can then be treated as though it were an `F_mpz_poly_t`, e.g. by passing it to `F_mpz_poly` functions. Note that such manipulation should not change the contents of the polynomial, i.e. the alias should be passed only as the input to `F_mpz_poly` functions.

```
void _F_mpz_mod_poly_attach_F_mpz_poly(F_mpz_mod_poly_t out,
                                    const F_mpz_poly_t in)
```

Attach a temporary `F_mpz_mod_poly_t` object to an existing `F_mpz_poly_t` object. The `F_mpz_poly_t` can then be treated as though it were an `F_mpz_mod_poly_t`, e.g. by passing it to `F_mpz_mod_poly` functions. Note that such manipulation should not change the contents of the polynomial, i.e. the alias should be passed only as the input to `F_mpz_mod_poly` functions. Note that the `F_mpz_mod_poly_t` must be initialised with `F_mpz_mod_poly_init` (but subsequently unusued) to set the modulus and must not be cleared after use.

```
void _F_mpz_mod_poly_attach_shift(F_mpz_mod_poly_t poly1,
              const F_mpz_mod_poly_t poly2, const ulong n)
```

Attach a temporary `F_mpz_mod_poly_t poly1` to an existing `F_mpz_mod_poly_t poly2` object, but ignore the first `n` coefficients, i.e. notionally equivalent to first shifting `poly2` to the right by `n` coefficients and then attaching `poly1`.

```
void _F_mpz_mod_poly_attach_truncate(F_mpz_mod_poly_t poly1,
               const F_mpz_mod_poly_t poly2, const ulong n)
```

Attach a temporary `F_mpz_mod_poly_t poly1` to an existing `F_mpz_mod_poly_t poly2` object, but notionally truncate `poly2` to length `n` before attaching `poly1`. The result is then normalised.

12.7 Reduction

```
void _F_mpz_mod_poly_reduce_coeffs(F_mpz_mod_poly_t poly)
```

Reduce all the coefficients of an `F_mpz_mod_poly_t poly` modulo the modulus `poly->P`. This function is mainly used internally, as all functions in this module automatically reduce their output unless explicitly stated.

```
void _F_mpz_poly_reduce_coeffs(F_mpz_poly_t poly,
                                        const F_mpz_t P)
```

Reduce all the coefficients of an `F_mpz_poly_t poly` modulo the modulus `P`. This function might be used before converting an `F_mpz_poly_t` to an `F_mpz_mod_poly_t` with the modulus `P`.

12.8 Assignment and swap

```
void F_mpz_mod_poly_set(F_mpz_mod_poly_t poly1,
                        const F_mpz_mod_poly_t poly2)
```

Set `poly1` to equal `poly2`. No check is made to ensure both polynomials have the same modulus.

```
void F_mpz_mod_poly_swap(F_mpz_mod_poly_t poly1,
                         F_mpz_mod_poly_t poly2)
```

Efficiently swap the contents of `poly1` and `poly2` by swapping pointers. No check is made to ensure both polynomials have the same modulus, and the moduli are not swapped.

```
void F_mpz_mod_poly_zero(F_mpz_mod_poly_t poly)
```

Set `poly` to the zero polynomial.

12.9 Comparison

```
int F_mpz_mod_poly_equal(const F_mpz_mod_poly_t poly1,
                         const F_mpz_mod_poly_t poly2)
```

Return 1 if `poly1` and `poly2` are (arithmetically) equal, otherwise return 0.

12.10 Input/output

```
void F_mpz_mod_poly_print(F_mpz_mod_poly_t poly)
```

Print the given polynomial to `stdout`. The format is as per the `F_mpz_poly` module, except that a space and then ": P =" is appended, followed by a space, then the integer modulus `poly->P`.

12.11 Shifting

```
void F_mpz_mod_poly_left_shift(F_mpz_mod_poly_t res,
            const F_mpz_mod_poly_t poly, const ulong n)
```

Shift the polynomial `poly` to the left by `n` coefficients, i.e. multiply by x^n.

```
void F_mpz_mod_poly_right_shift(F_mpz_mod_poly_t res,
            const F_mpz_mod_poly_t poly, const ulong n)
```

Shift the polynomial `poly` to the right by `n` coefficients, i.e. divide by x^n and discard any remainder. If `n` is greater than the current length of `poly` the result is the zero polynomial.

12.12 Addition/subtraction

```
void F_mpz_mod_poly_add(F_mpz_mod_poly_t res,
 const F_mpz_mod_poly_t poly1, const F_mpz_mod_poly_t poly2)
```

Set `res` to the sum of `poly1` and `poly2`.

```
void F_mpz_mod_poly_sub(F_mpz_mod_poly_t res,
 const F_mpz_mod_poly_t poly1, const F_mpz_mod_poly_t poly2)
```

Set `res` to the difference of `poly1` and `poly2`.

12.13 Scalar multiplication

```
void F_mpz_mod_poly_scalar_mul(F_mpz_mod_poly_t res,
            const F_mpz_mod_poly_t poly1, const F_mpz_t x)
```

Set `res` to the sum of `poly1` multiplied by the scalar `x`.

12.14 Multiplication

```
void F_mpz_mod_poly_mul(F_mpz_mod_poly_t res,
 const F_mpz_mod_poly_t poly1, const F_mpz_mod_poly_t poly2)
```

Set `res` to the product of `poly1` and `poly2`.

```
void F_mpz_mod_poly_mul_trunc_left(F_mpz_mod_poly_t res,
 const F_mpz_mod_poly_t poly1, const F_mpz_mod_poly_t poly2,
                                      const ulong trunc)
```

Set `res` to the product of `poly1` and `poly2` where only the most significant `trunc` coefficients are necessarily computed. The remaining coefficients will either be zero or correct.

12.15 Division

```
void F_mpz_mod_poly_divrem(F_mpz_mod_poly_t Q,
    F_mpz_mod_poly_t R, const F_mpz_mod_poly_t A,
                            const F_mpz_mod_poly_t B)
```

Set `Q` and `R` to the quotient and remainder of `A` divided by `B`. This function requires that the leading coefficient of `B` be coprime to the modulus `A->P`, e.g. the modulus is prime.

```
void F_mpz_mod_poly_rem(F_mpz_mod_poly_t R,
             F_mpz_mod_poly_t A, F_mpz_mod_poly_t B)
```

Set `R` to the remainder of `A` divide `B`. This function is for convenience only, it is not faster than `F_mpz_mod_poly_divrem` at the current time. This function requires that the leading coefficient of `B` be coprime to the modulus `A->P`, e.g. the modulus is prime.

13 The F_mpz_mat module

The F_mpz_mat module is for matrices whose entries are the new F_mpz_t integer type. The data type F_mpz_mat_t represents a matrix in $\mathbb{Z}^{n\times m}$.

Unless otherwise specified or if there is dimension mismatch, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

13.1 Simple example

The following example computes the square of

```
[[10 21]
[16 3]]

#include F_mpz_mat.h
....
F_mpz_mat_t A;
F_mpz_mat_init(A, 2, 2);
F_mpz_set_ui(A->rows[0] + 0, 10);
F_mpz_set_ui(A->rows[0] + 1, 21);
F_mpz_set_ui(A->rows[1] + 0, 16);
F_mpz_set_ui(A->rows[1] + 1, 3);
F_mpz_mat_mul_classical(A, A, A);
F_mpz_mat_print_pretty(A);
F_mpz_mat_clear(A);
```

The output is:

```
[[436 273]
[208 345]]
```

13.2 Definition of the F_mpz_mat_t matrix type

The F_mpz_mat_t type is a typedef for an array of length 1 of F_mpz_mat_struct's. This permits passing parameters of type F_mpz_mat_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 F_mpz_mat_t. For simplicity we will think of an F_mpz_mat_t as a struct, though in practice to access fields of this struct, one needs to dereference first, e.g. to access the r field of an F_mpz_mat_t called A one writes A->r (this is the number of rows of A).

The entries of an F_mpz_mat_t are stored as an array of arrays of F_mpz_t's, thus to access the F_mpz_t in the i^{th} row and j^{th} column of the F_mpz_mat_t A use A->rows[i] + j.

Functions in F_mpz_mat do all the memory management for the user. One does not need to specify the maximum length or any coefficient sizes 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 F_mpz_mat.

13.3 Memory management

```
void F_mpz_mat_init(F_mpz_mat_t mat, ulong r, ulong c)
```

Initialise a matrix `mat` with `r` rows and `c` columns, for use.

```
void F_mpz_mat_init_identity(F_mpz_mat_t mat, ulong n)
```

Initialise an `n x n` matrix and set it to the `n x n` identity matrix.

```
void F_mpz_mat_clear(F_mpz_mat_t mat)
```

Clear a matrix after use, clearing any entries and releasing the memory used by it. The matrix cannot be used again until it is initialised afresh.

13.4 Input/output

```
int F_mpz_mat_from_string(F_mpz_mat_t mat,
                                        const char * s)
```

Reads a matrix from a string `s`. The format is a number of rows followed by a whitespace (space, tab or new line), followed by a number of columns, followed by a whitespace, followed by the entries of the matrix given row by row, each entry separated by a space. If a valid matrix is read, 1 is returned, otherwise 0 is returned.

```
char * F_mpz_mat_to_string(F_mpz_mat_t mat)
```

Reads a matrix `mat` and stores its string representation (as described by the previous function) in a string. To free the string one must use `free`.

```
int F_mpz_mat_from_string_pretty(F_mpz_mat_t mat,
                                              char * s)
```

Reads a matrix from a string `s`. The format is a left square bracket followed by a set of rows (delimited in any manner desired, except by numerals or the minus sign), followed by a closing right square bracket. The format of each row is an opening square bracket followed by a list of entries, which may be signed multiprecision integers (delimited in whatever manner desired, except with numerals or minus signs), followed by a closing square bracket.

```
char * F_mpz_mat_to_string_pretty(F_mpz_mat_t mat)
```

Reads a matrix `mat` and stores its pretty string representation (as described by the previous function) in a string. To free the string one must use `free`.

```
void F_mpz_mat_print(F_mpz_mat_t mat)
```

Print the standard string representation of `mat` (as described above) to `stdout`.

```
void F_mpz_mat_print_pretty(F_mpz_mat_t mat)
```

Print the pretty string representation of `mat` (as described above) to `stdout`.

```
void F_mpz_mat_fprint(F_mpz_mat_t mat, FILE * f)
```

Print the standard string representation of `mat` (as described above) to the given file stream.

```
void F_mpz_mat_fprint_pretty(F_mpz_mat_t mat,
                                          FILE * f)
```

Print the pretty string representation of `mat` (as described above) to the given file stream.

```
int F_mpz_mat_fread(F_mpz_mat_t mat, FILE * f)
```

Read from the given file stream, a string representation of a matrix, in standard form (as described above) and store it in `mat`. If a valid matrix is read, 1 is returned, otherwise 0 is returned.

```
int F_mpz_mat_fread_pretty(F_mpz_mat_t mat,
                                        FILE * f)
```

Read from the given file stream, a string representation of a matrix, in pretty form (as described above) and store it in `mat`. If a valid matrix is read, 1 is returned, otherwise 0 is returned.

13.5 Row export

As each row of an `F_mpz_mat` is an array of `F_mpz`'s, the rows can be accessed separately, e.g. given a matrix `mat`, the first row would be `mat->rows[0]`. Various functions are provided to export such a row (or part thereof) to an array of a different kind.

```
long _F_mpz_vec_to_d_vec_2exp(double * appv,
             const F_mpz * vec, const ulong n)
```

Export the array of `n` entries starting at the pointer `vec` to an array of doubles `appv`, each entry of which is notionally multiplied by a single returned exponent to give the original entry. The returned exponent is set to be the maximum exponent of all the original entries so that all the doubles in the returned array have a maximum absolute value of 1.0.

```
void _F_mpz_vec_to_mpfr_vec(__mpfr_struct * appv,
              const F_mpz * vec, const ulong n)
```

Export the array of `n` entries starting at the pointer `vec` to an array of `mpfr`'s. Note that an array of `__mpfr_struct`'s is returned, instead of an array of `mpfr_t`'s. This means that `appv + i` for some integer `i` can be used in lieu of an `mpfr_t` as an input or output of an `mpfr` function. This makes access to entries in arrays of `mpfr`'s similar in syntax to access to entries in arrays of `fmpz`'s.

13.6 Assignment

```
void F_mpz_mat_set(F_mpz_mat_t mat1,
                         const F_mpz_mat_t mat2)
```

Set `mat1` equal to `mat2`. It is required that the number of rows of both matrices is the same and similarly for the number of columns.

```
void F_mpz_mat_swap(F_mpz_mat_t mat1,
                               F_mpz_mat_t mat2)
```

Efficiently swap `mat1` and `mat2` by swapping pointers.

13.7 Comparison

```
int F_mpz_mat_equal(const F_mpz_mat_t mat1,
                         const F_mpz_mat_t mat2)
```

Return 1 if `mat1` and `mat2` are (arithmetically) equal. No check is made that the matrices have the same numbers of rows or columns. It is therefore assumed that they are the same size.

13.8 Negation

```
void F_mpz_mat_neg(F_mpz_mat_t mat1,
                         const F_mpz_mat_t mat2)
```

Set `mat1` to the matrix whose entries are the negation of those in `mat2`.

13.9 Vector memory management

It is sometimes necessary to create vectors independently of matrices and manipulate them directly. As rows of `F_mpz_mat_t`'s are arrays of `F_mpz`'s, this is also the natural format for independent vectors.

```
F_mpz * _F_mpz_vec_init(ulong n)
```

Create and initialise a vector of length `n` of `fmpz`'s. If `vec` is the return value, then `vec + i` can be used in lieu of an `F_mpz_t` in an `F_mpz` function and thus the entry at that index can be individually manipulated.

```
void _F_mpz_vec_clear(F_mpz * vec, ulong n)
```

Clear a previously allocated vector of length `n`. It is necessary to pass the length as each individual entry is cleared and the memory it used is freed.

13.10 Vector copy and comparison

```
void _F_mpz_vec_copy(F_mpz * vec1,
                const F_mpz * vec2, ulong n)
```

Copies the vector of length `n` at `vec2` to the vector at `vec1`. Note that operations of this type can be used to copy partial vectors by simply offsetting the start of each vector by pointer arithmetic and using a lesser length `n`.

```
int _F_mpz_vec_equal(F_mpz * vec1,
                 const F_mpz * vec2, ulong n)
```

Return 1 if the vector at `vec2` of length `n` is (arithmetically) equal to the vector of the same length at `vec1`, otherwise return 0.

13.11 Addition and subtraction

```
void F_mpz_mat_add(F_mpz_mat_t res,
 const F_mpz_mat_t mat1, const F_mpz_mat_t mat2)
```

Set the matrix `res` to the sum of the matrices `mat1` and `mat2`. It is assumed all three matrices have the same dimensions.

```
void F_mpz_mat_sub(F_mpz_mat_t res,
 const F_mpz_mat_t mat1, const F_mpz_mat_t mat2)
```

Set the matrix `res` to the difference of the matrices `mat1` and `mat2`. It is assumed all three matrices have the same dimensions.

```
void _F_mpz_vec_add(F_mpz * res,
 const F_mpz * vec1, const F_mpz * vec2, ulong n)
```

Set the vector `res` of length `n` to the sum of the vectors `vec1` and `vec2` of the same length.

```
void _F_mpz_vec_sub(F_mpz * res,
 const F_mpz * vec1, const F_mpz * vec2, ulong n)
```

Set the vector `res` of length `n` to the difference of the vectors `vec1` and `vec2` of the same length.

13.12 Scalar multiplication

```
void _F_mpz_vec_mul_ui(F_mpz * vec1,
                   F_mpz * vec2, ulong n, ulong x)
```

Multiply the vector `vec2` of length `n` by the scalar `ulong x` and set the vector `vec1`, of the same length, to the result.

```
void _F_mpz_vec_mul_si(F_mpz * vec1,
                    F_mpz * vec2, ulong n, long x)
```

Multiply the vector `vec2` of length `n` by the scalar `long x` and set the vector `vec1`, of the same length, to the result.

```
void _F_mpz_vec_mul_F_mpz(F_mpz * vec1,
                  F_mpz* vec2, ulong n, F_mpz_t x)
```

Multiply the vector `vec2` of length `n` by the scalar `F_mpz_t x` and set the vector `vec1`, of the same length, to the result.

13.13 Scalar addmul/submul

```
void _F_mpz_vec_addmul_ui(F_mpz * vec1,
                   F_mpz * vec2, ulong n, ulong x)
```

Multiply the vector `vec2` of length `n` by the scalar `ulong x` and add the result in-place to the vector `vec1` of the same length.

```
void _F_mpz_vec_addmul_F_mpz(F_mpz * vec1,
                 F_mpz * vec2, ulong n, F_mpz_t x)
```

Multiply the vector `vec2` of length `n` by the scalar `F_mpz_t x` and add the result in-place to the vector `vec1` of the same length.

```
void _F_mpz_vec_submul_ui(F_mpz * vec1,
                   F_mpz * vec2, ulong n, ulong x)
```

Multiply the vector `vec2` of length `n` by the scalar `ulong x` and subtract the result in-place from the vector `vec1` of the same length.

```
void _F_mpz_vec_submul_F_mpz(F_mpz * vec1,
                 F_mpz * vec2, ulong n, F_mpz_t x)
```

Multiply the vector `vec2` of length `n` by the scalar `F_mpz_t x` and subtract the result in-place from the vector `vec1` of the same length.

```
void _F_mpz_vec_addmul_2exp_ui(F_mpz * vec1,
       F_mpz * vec2, ulong n, ulong c, ulong exp)
```

Multiply the vector `vec2` of length `n` by the scalar $c \times 2^{exp}$ and add the result in-place to the vector `vec1` of the same length, where `c` is a `ulong`.

```
void _F_mpz_vec_submul_2exp_ui(F_mpz * vec1,
       F_mpz * vec2, ulong n, ulong c, ulong exp)
```

Multiply the vector `vec2` of length `n` by the scalar $c \times 2^{exp}$ and subtract the result in-place from the vector `vec1` of the same length, where `c` is a `ulong`.

```
void _F_mpz_vec_submul_2exp_F_mpz(F_mpz * vec1,
     F_mpz * vec2, ulong n, F_mpz_t c, ulong exp)
```

Multiply the vector `vec2` of length `n` by the scalar $c \times 2^{exp}$ and subtract the result in-place from the vector `vec1` of the same length, where `c` is an `F_mpz_t`.

```
void F_mpz_mat_swap_rows(F_mpz_mat_t mat,
                                ulong r1, ulong r2)
```

Efficiently swap rows `r1` and `r2` of the matrix `mat` by swapping pointers.

```
void _F_mpz_vec_neg(F_mpz * vec1,
                           F_mpz * vec2, ulong n)
```

Set the vector `vec1` of length `n` to the vector of the same length whose entries are the negative of the entries in the vector `vec2`.

13.14 Basic properties

```
long F_mpz_mat_max_bits(const F_mpz_mat_t M)
```

Determine the maximum number of bits `b` required to store the absolute value of the largest coefficient of the matrix `M`. If `M` has signed coefficients, return `-b`, else return `b`.

```
long F_mpz_mat_max_bits2(ulong * row,
           ulong * col, const F_mpz_mat_t M)
```

Determine the maximum number of bits `b` required to store the absolute value of the largest coefficient of the matrix `M`. If `M` has signed coefficients, return `-b`, else return `b`. Set `row` and `col` to the row and column of an entry which consumes this many bits. Note that the function need not return the largest such element, just one such.

13.15 Matrix by scalar operations

```
void F_mpz_mat_mul_2exp(F_mpz_mat_t res,
                        F_mpz_mat_t M, ulong n)
```

Set the matrix `res` to the matrix `M`, of the same dimensions, multiplied by 2^n.

```
void F_mpz_mat_div_2exp(F_mpz_mat_t res,
                        F_mpz_mat_t M, ulong n)
```

Set the matrix `res` to the matrix `M`, of the same dimensions, divided by 2^n. Each entry is truncated towards zero and the remainder is discarded.

13.16 Scalar product

```
void _F_mpz_vec_scalar_product(F_mpz_t sp,
        F_mpz * vec1, F_mpz * vec2, ulong n)
```

Set `sp` to the vector scalar product of the vectors `vec1` and `vec2` of length `n`.

13.17 Column operations

```
int F_mpz_mat_col_equal(F_mpz_mat_t M,
                        ulong c1, ulong c2)
```

Return 1 if columns `c1` and `c2` of the matrix `M` are (arithmetically) equal, otherwise return 0.

```
void F_mpz_mat_col_copy(F_mpz_mat_t M,
                        ulong c1, ulong c2)
```

Set column `c1` of the matrix `M` to be equal to column `c2`.

13.18 Matrix windows

```
void F_mpz_mat_window_init(F_mpz_mat_t U,
         F_mpz_mat_t M, ulong r0, ulong c0,
                       ulong rows, ulong cols)
```

Attach an `F_mpz_mat_t` structure `U` to an existing matrix `M` starting at the position given by row `r0` and column `c0` and extending over a block with the given number of rows and columns. The window alias `U` may only be used as the input of `F_mpz_mat` functions (including vector functions). The matrix structure `U` must not be initialised before being set up as a window.

```
void F_mpz_mat_window_clear(F_mpz_mat_t U)
```

Free any memory allocated to set up a matrix window `U`. The matrix `U` cannot be used again unless it it initialised or set up as a matrix window afresh.

13.19 Symmetric modulus

```
void F_mpz_mat_smod(F_mpz_mat_t res,
                    F_mpz_mat_t M, F_mpz_t P)
```

Set matrix `res` of the same dimensions as matrix `M` to be the matrix whose entries are the symmetric mod modulo `P` of the entries in `M`, i.e. to the values mod `P` that are in the range `(-P/2, P/2]`.

13.20 Matrix transpose

```
void F_mpz_mat_transpose(F_mpz_mat_t res,
                                F_mpz_mat_t M)
```

Set the matrix `res` with `c` rows and `r` columns to the transpose of the matrix `M` with `r` rows and `c` columns. Currently this operation is performed naively in a cache inefficient manner and is provided for convenience.

14 The F_mpz_LLL module

The `F_mpz_LLL` module introduces lattice reduction functionality to the FLINT `F_mpz_mat_t` type for matrices with `F_mpz_t` entries. The rows of the $n \times m$ `F_mpz_mat_t` are treated as a basis of a lattice, a $\mathbb{Z}$-module in $\mathbb{R}^m$. Integer combinations of the rows are found which remain a basis of the same lattice while having better properties such as near orthogonality and generally shorter length.

14.1 Simple example

The following example computes a reduced basis from the rows of

```
[[1 0 16884]
[0 1 714]]
```

```
#include "F_mpz_mat.h"
#include "F_mpz_LLL.h"
 ....
F_mpz_mat_t L;
F_mpz_mat_init(L, 2, 3);
F_mpz_set_ui(L->rows[0] + 0, 1);
F_mpz_set_ui(L->rows[1] + 1, 1);
F_mpz_set_ui(L->rows[0] + 2, 16884);
F_mpz_set_ui(L->rows[1] + 2, 714);
LLL(L);
F_mpz_mat_print_pretty(L);
F_mpz_mat_clear(L);
```

The output is:

```
[[3 -71 -42]
[-5 118 -168]]
```

14.2 Two varieties of LLL

```
void LLL (F_mpz_mat_t B)
```

Given a matrix `B`, LLL-reduces the rows of `B`, in place. The strength of the reduction is determined by the macros `DELTA` and `ETA` defined in `F_mpz_LLL.h`. It is assumed that DELTA $\in (1/4, 1]$ and ETA $\in (1/2, \sqrt{\text{DELTA}}]$. For details on these parameters see the paper "An LLL algorithm with quadratic complexity" by Nguyen and Stehlé.

```
int U_LLL_with_removal(F_mpz_mat_t FM,
                            long new_size, F_mpz_t gs_B)
```

A new type of LLL which tends to be numerically stable. The given matrix `FM` is shifted down by a power of 2 and truncated until the `new_size` most significant bits of `FM` are found. An identity matrix is augmented to this truncation and the augmented matrix is reduced. This gives a unimodular transformation, `U`, which is then applied to `FM`. If the bit size of `FM` decreases after multiplication by `U` then the algorithm repeats, otherwise standard floating-point LLL is performed.

The parameter `gs_B` is used when searching for only short vectors in the lattice. Let `gs_B` be an upper bound for the squared ℓ_2 norm of targeted vector or group of vectors in the lattice, then the output of `U_LLL_with_removal` is an index, call it `d`, such that all targeted vectors are within the span of the first `d` rows of the output `FM`. The computation of `d` is proven and performed after all other computations so there is no speed-up from using any value of `gs_B`.

14.3 Some special randomized matrices

```
void F_mpz_mat_randintrel(F_mpz_mat_t mat, ulong bits)
```

Given an `F_mpz_mat_t mat` with `r` rows and `r+1` columns we populate `mat` with the identity matrix in the first `r` columns and random `bits`-bit integers in the final column.

```
void F_mpz_mat_randintrel_little_big(F_mpz_mat_t mat,
                         ulong n, ulong bits1, ulong bits)
```

Given an `F_mpz_mat_t mat` with `r` rows and `r+1` columns we populate `mat` with the identity matrix in the first `r` columns and random `bits1`-bit integers in the first `n` entries of the final column and random `bits`-bit integers in the remaining entries of the final column.

```
void F_mpz_mat_rand_unimodular(F_mpz_mat_t U, ulong bits)
```

Given a square `F_mpz_mat_t U` this procedure populates it with a randomized unimodular matrix (determinant 1) whose entries are `bits`-bit `F_mpz_t`'s.

```
void F_mpz_mat_rand_unimodular_little_big(F_mpz_mat_t U,
                        ulong n, ulong bits1, ulong bits2)
```

Given a square `F_mpz_mat_t U` this procedure populates it with a randomized unimodular matrix (determinant 1) where the first `n` rows have `bits1`-bit `F_mpz_t` entries and the remaining rows have `bits2`-bit entries.

15 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. Unless otherwise specified, 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.

15.1 Precomputed inverses

```
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.

15.2 Fast division

```
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.

15.3 Modulo arithmetic

```
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_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 place 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.

15.4 Powering

```
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_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.

15.5 Legendre and Jacobi symbols

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

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

```
int z_jacobi(long x, unsigned long y)
```

Calculates the Jacobi symbol of `x` mod `y`. Assumes that gcd(`x`,`y`) = 1 and `y` is odd.

15.6 Primality testing

```
int z_ispseudoprime_fermat(unsigned long const n,
                           unsigned long const b)
```

Checks to see if `n` is a Fermat pseudoprime with base `b`. Assumes that `n` does not divide `b`.

```
int z_isprime(unsigned long n)

int z_isprime_precomp(unsigned long n, pre_inv_t ninv)
```

Returns 1 if n is proved prime, otherwise it returns 0 in which case n is composite. In the precomp version of the function it is assumed that `n` is greater than 2 and odd. The function takes a precomputed inverse of n.

```
int z_isprobab_prime(unsigned long n)

int z_isprobab_prime_precomp(unsigned long n, pre_inv_t ninv)
```

This is a deterministic prime test up to 10^{16}. Requires `n` to be at most `FLINT_BITS-1` bits. For numbers greater than 10^{16} there are no known counterexamples to the conjecture that a composite will never be declared prime. Primes are always declared prime by this test.

```
unsigned long z_nextprime(unsigned long n, int proved)
```

Returns the next prime after `n`. Assumes the result will fit in an unsigned long. If `proved` is 0 the prime is not proven prime, otherwise it is.

```
int z_isprime_pocklington(unsigned long const n,
                    unsigned long const iterations)
```

Proves that `n` is prime using a Pocklington-Lehmer test. Returns 0 if composite, 1 if prime and -1 if it failed to prove either way. The number of iterations can be increased for a more thorough check but will take longer. Setting `iterations` to `-1L` will cause it to continue until the number is proven prime or composite.

```
int z_ispseudoprime_lucas_ab(unsigned long n, int a, int b)
```

Tests to see if `n` is an `a,b`-Lucas pseudoprime. Returns 0 if `n` is composite or fails gcd(`n`, 2*`a`*`b`*(`a`*`a` - 4*`b`)) = 1. Returns 1 if `n` is a Lucas pseudoprime with respect to $x^2 - ax + b$. Returns -1 if the discriminant of the quadratic is square. Assumes `n` has been checked for primality using trial factoring up to 256. The absolute values of `a` and `b` should be < 128. For details of this function see the book "Primes : a computational perspective" by Pomerance and Crandall.

```
int z_ispseudoprime_lucas(unsigned long const n)
```

Tests if n is a Lucas pseudoprime as per the algorithm of Baillie and Wagstaff (see Math. Comp. vol 35, no. 152, 1980, pp. 1391–1417). Assumes `n` has been checked for primality using trial factoring up to 256.

15.7 Roots

```
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_intsqrt(unsigned long r)
```

Returns the integer part of the square root of `r`.

```
int z_issquare(long n)
```

Returns 1 if n is a square, otherwise returns 0. There are no restrictions on n, which may be signed and negative numbers will not be declared square.

15.8 GCD and inverses

```
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 `n`.

```
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_xgcd(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`.

15.9 CRT

```
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.

15.10 Factoring

```
int z_remove(unsigned long * n,
                unsigned long p)

int z_remove_precomp(unsigned long * n,
          unsigned long p, pre_inv_t pinv)
```

Removes the highest power of `p` possible from `n` and returns the exponent to which it appeared in `n`. In the second function `n` can only be up to `FLINT_BITS-1` bits.

```
void z_factor(factor_t * factors, unsigned long n, int proved)
```

Find the factors of `n`. If `proved` is set to 0 then the factors are not proved prime, otherwise the result is proved.

The `factor_t` struct contains three fields. The first is the `num` field, which is an int containing the number of factors. Then `p` is an array of unsigned long's containing the actual factors, and the respective exponents are given by the array of unsigned long's comprising the `exp` field of the struct.

```
unsigned long z_factor_partial(factor_t * factors,
               unsigned long n, unsigned long limit, int proved)
```

Factors `n` until the product of the factor found is > `limit`. It puts the factors in `factors` and returns the cofactor. If `proved` is set to 0 then the factors are not proved prime, otherwise the result is proved.

```
int z_issquarefree(unsigned long n, int proved)
```

Returns 1 if n is squarefree, otherwise returns 0. If `proved` is set to 1 then the result is guaranteed, and if set to 0 then internal factoring may declare some composites prime. Note that `n` must be at most `FLINT_BITS - 1` bits.

15.11 Random numbers

```
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.

```
unsigned long z_randprime(unsigned long bits, int proved)
```

Returns a random prime integer with up to the given number of bits. Assumes `bits` > 1. If `proved` is 0 then the prime is not proven prime, otherwise it is.

16 The mpn_extras module

The `mpn_extras` module is designed to supplement the low level `mpn` functions provided in GMP/MPIR. 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. 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(d<<norm);
```

Note that although one must normalise `d` before precomputing its inverse, the actual value of `d`, not its normalisation, is passed to the functions below.

```
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 `xn` 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 `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`.

```
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.

17 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"
```

If one is linking against `libflint` then one must also build `NTL-interface.o` in the top level FLINT source tree as follows:

```
g++ -c NTL-interface -o NTL-interface.o -O2 -fPIC
```

One must then include `NTL-interface.o` in the list of files to link when compiling your program and linking against `libflint`, e.g.

```
g++ myprog.cpp NTL-interface.o -o myprog -O2 -I$FLINT_GMP_INCLUDE_DIR \
-I$FLINT_NTL_INCLUDE_DIR -L$FLINT_GMP_LIB_DIR -L$FLINT_NTL_LIB_DIR \
-lflint -lntl -lgmp
```

In each case the functions provided for conversion expect the output objects, whether NTL or FLINT objects, to be initialised. The first function is unmanaged in that the user must ensure that sufficient space is allocated in the `fmpz_t` to hold the integer contained in the `ZZ`.

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

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

The following functions are managed, in that a reallocation automatically occurs if insufficient space was allocated by the user.

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

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

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

18 The quadratic sieve

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

```
make mpQS
```

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 and not be a perfect power. 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 26 binary bits or below).

19 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/MPIR's `mpn_mul` and `mpz_mul` respectively.

These replacement functions are substantially faster than GMP 4.3.1 and somewhat faster than MPIR 1.2.0 when multiplying integers which are thousands of limbs in size. For smaller multiplications these functions call their respective GMP/MPIR counterparts.