CLN, a Class Library for Numbers

by Bruno Haible

Published by Bruno Haible, `<haible@clisp.cons.org>` and Richard Kreckel, `<kreckel@ginac.de>`.

Table of Contents

1 Introduction **1**

2 Installation **3**

2.1 Prerequisites 3
2.1.1 C++ compiler 3
2.1.2 Make utility 3
2.1.3 Sed utility 3
2.2 Building the library 3
2.2.1 Using the GNU MP Library 5
2.3 Installing the library 5
2.4 Cleaning up 5

3 Ordinary number types **6**

3.1 Exact numbers 6
3.2 Floating-point numbers 7
3.3 Complex numbers 8
3.4 Conversions 8

4 Functions on numbers **10**

4.1 Constructing numbers 10
4.1.1 Constructing integers 10
4.1.2 Constructing rational numbers 10
4.1.3 Constructing floating-point numbers 10
4.1.4 Constructing complex numbers 10
4.2 Elementary functions 11
4.3 Elementary rational functions 12
4.4 Elementary complex functions 12
4.5 Comparisons 13
4.6 Rounding functions 13
4.7 Roots 16
4.8 Transcendental functions 17
4.8.1 Exponential and logarithmic functions 17
4.8.2 Trigonometric functions 17
4.8.3 Hyperbolic functions 19
4.8.4 Euler gamma 20
4.8.5 Riemann zeta 20
4.9 Functions on integers 20
4.9.1 Logical functions 20
4.9.2 Number theoretic functions 23
4.9.3 Combinatorial functions 23
4.10 Functions on floating-point numbers 24
4.11 Conversion functions 25

4.11.1 Conversion to floating-point numbers 25
4.11.2 Conversion to rational numbers 26
4.12 Random number generators 26
4.13 Obfuscating operators 27

5 Input/Output 29
5.1 Internal and printed representation 29
5.2 Input functions 30
5.3 Output functions 31

6 Rings 33

7 Modular integers 34
7.1 Modular integer rings 34
7.2 Functions on modular integers 34

8 Symbolic data types 37
8.1 Strings 37
8.2 Symbols 37

9 Univariate polynomials 38
9.1 Univariate polynomial rings 38
9.2 Functions on univariate polynomials 40
9.3 Special polynomials 42

10 Internals 43
10.1 Why C++ ? 43
10.2 Memory efficiency 43
10.3 Speed efficiency 43
10.4 Garbage collection 44

11 Using the library 45
11.1 Compiler options 45
11.2 Compatibility to old CLN versions 46
11.3 Include files 46
11.4 An Example 49
11.5 Debugging support 50

12 Customizing 52
12.1 Error handling 52
12.2 Floating-point underflow 52
12.3 Customizing I/O 52
12.4 Customizing the memory allocator 52

Index 53

1 Introduction

CLN is a library for computations with all kinds of numbers. It has a rich set of number classes:

- Integers (with unlimited precision),
- Rational numbers,
- Floating-point numbers:
  - Short float,
  - Single float,
  - Double float,
  - Long float (with unlimited precision),
- Complex numbers,
- Modular integers (integers modulo a fixed integer),
- Univariate polynomials.

The subtypes of the complex numbers among these are exactly the types of numbers known to the Common Lisp language. Therefore `CLN` can be used for Common Lisp implementations, giving '`CLN`' another meaning: it becomes an abbreviation of "Common Lisp Numbers".

The CLN package implements

- Elementary functions (`+`, `-`, `*`, `/`, `sqrt`, comparisons, . . .),
- Logical functions (logical `and`, `or`, `not`, . . .),
- Transcendental functions (exponential, logarithmic, trigonometric, hyperbolic functions and their inverse functions).

CLN is a C++ library. Using C++ as an implementation language provides

- efficiency: it compiles to machine code,
- type safety: the C++ compiler knows about the number types and complains if, for example, you try to assign a float to an integer variable.
- algebraic syntax: You can use the `+`, `-`, `*`, `=`, `==`, . . . operators as in C or C++.

CLN is memory efficient:

- Small integers and short floats are immediate, not heap allocated.
- Heap-allocated memory is reclaimed through an automatic, non-interruptive garbage collection.

CLN is speed efficient:

- The kernel of CLN has been written in assembly language for some CPUs (`i386`, `m68k`, `sparc`, `mips`, `arm`).
- On all CPUs, CLN may be configured to use the superefficient low-level routines from GNU GMP version 3.
- It uses Karatsuba multiplication, which is significantly faster for large numbers than the standard multiplication algorithm.
- For very large numbers (more than 12000 decimal digits), it uses Schönhage-Strassen multiplication, which is an asymptotically optimal multiplication algorithm, for multiplication, division and radix conversion.

CLN aims at being easily integrated into larger software packages:

- The garbage collection imposes no burden on the main application.
- The library provides hooks for memory allocation and exceptions.
- All non-macro identifiers are hidden in namespace `cln` in order to avoid name clashes.

2 Installation

This section describes how to install the CLN package on your system.

2.1 Prerequisites

2.1.1 C++ compiler

To build CLN, you need a C++ compiler. Actually, you need GNU `g++ 2.95` or newer.

The following C++ features are used: classes, member functions, overloading of functions and operators, constructors and destructors, inline, const, multiple inheritance, templates and namespaces.

The following C++ features are not used: `new`, `delete`, virtual inheritance, exceptions.

CLN relies on semi-automatic ordering of initializations of static and global variables, a feature which I could implement for GNU g++ only. Also, it is not known whether this semi-automatic ordering works on all platforms when a non-GNU assembler is being used.

2.1.2 Make utility

To build CLN, you also need to have GNU `make` installed.

Only GNU `make` 3.77 is unusable for CLN; other versions work fine.

2.1.3 Sed utility

To build CLN on HP-UX, you also need to have GNU `sed` installed. This is because the libtool script, which creates the CLN library, relies on `sed`, and the vendor's `sed` utility on these systems is too limited.

2.2 Building the library

As with any autoconfiguring GNU software, installation is as easy as this:

```
$ ./configure
$ make
$ make check
```

If on your system, '`make`' is not GNU `make`, you have to use '`gmake`' instead of '`make`' above.

The `configure` command checks out some features of your system and C++ compiler and builds the `Makefile`s. The `make` command builds the library. This step may take about an hour on an average workstation. The `make check` runs some test to check that no important subroutine has been miscompiled.

The `configure` command accepts options. To get a summary of them, try

```
$ ./configure --help
```

Some of the options are explained in detail in the '`INSTALL.generic`' file.

You can specify the C compiler, the C++ compiler and their options through the following environment variables when running `configure`:

`CC` Specifies the C compiler.

`CFLAGS` Flags to be given to the C compiler when compiling programs (not when linking).

`CXX` Specifies the C++ compiler.

`CXXFLAGS` Flags to be given to the C++ compiler when compiling programs (not when linking).

Examples:

```
$ CC="gcc" CFLAGS="-O" CXX="g++" CXXFLAGS="-O" ./configure
$ CC="gcc -V egcs-2.91.60" CFLAGS="-O -g" \
  CXX="g++ -V egcs-2.91.60" CXXFLAGS="-O -g" ./configure
$ CC="gcc -V 2.95.2" CFLAGS="-O2 -fno-exceptions" \
  CXX="g++ -V 2.95.2" CFLAGS="-O2 -fno-exceptions" ./configure
$ CC="gcc -V 3.0.4" CFLAGS="-O2 -finline-limit=1000 -fno-exceptions" \
  CXX="g++ -V 3.0.4" CFLAGS="-O2 -finline-limit=1000 -fno-exceptions" \
  ./configure
```

Note that for these environment variables to take effect, you have to set them (assuming a Bourne-compatible shell) on the same line as the `configure` command. If you made the settings in earlier shell commands, you have to `export` the environment variables before calling `configure`. In a `csh` shell, you have to use the '`setenv`' command for setting each of the environment variables.

Currently CLN works only with the GNU `g++` compiler, and only in optimizing mode. So you should specify at least `-O` in the CXXFLAGS, or no CXXFLAGS at all. (If CXXFLAGS is not set, CLN will use `-O`.)

If you use `g++` 3.x, I recommend adding '`-finline-limit=1000`' to the CXXFLAGS. This is essential for good code.

If you use `g++` gcc-2.95.x or gcc-3.x , I recommend adding '`-fno-exceptions`' to the CXXFLAGS. This will likely generate better code.

If you use `g++` from gcc-3.0.4 or older on Sparc, add either '`-O`', '`-O1`' or '`-O2 -fno-schedule-insns`' to the CXXFLAGS. With full '`-O2`', `g++` miscompiles the division routines. If you use `g++` older than 2.95.3 on Sparc you should also specify '`--disable-shared`' because of bad code produced in the shared library. Also, do not use gcc-3.0 on Sparc for compiling CLN, it won't work at all.

If you use `g++` on OSF/1 or Tru64 using gcc-2.95.x, you should specify '`--disable-shared`' because of linker problems with duplicate symbols in shared libraries. If you use `g++` from gcc-3.0.n, with n larger than 1, you should *not* add '`-fno-exceptions`' to the CXXFLAGS, since that will generate wrong code (gcc-3.1 is okay again, as is gcc-3.0).

Also, please do not compile CLN with `g++` using the `-O3` optimization level. This leads to inferior code quality.

If you use `g++` from gcc-3.1, it will need 235 MB of virtual memory. You might need some swap space if your machine doesn't have 512 MB of RAM.

By default, both a shared and a static library are built. You can build CLN as a static (or shared) library only, by calling `configure` with the option '`--disable-shared`' (or '`--disable-static`'). While shared libraries are usually more convenient to use, they may not work on all architectures. Try disabling them if you run into linker problems. Also, they are generally somewhat slower than static libraries so runtime-critical applications should be linked statically.

If you use `g++` from gcc-3.1 with option '`-g`', you will need some disk space: 335 MB for building as both a shared and a static library, or 130 MB when building as a shared library only.

2.2.1 Using the GNU MP Library

Starting with version 1.1, CLN may be configured to make use of a preinstalled `gmp` library. Please make sure that you have at least `gmp` version 3.0 installed since earlier versions are unsupported and likely not to work. Enabling this feature by calling `configure` with the option '`--with-gmp`' is known to be quite a boost for CLN's performance.

If you have installed the `gmp` library and its header file in some place where your compiler cannot find it by default, you must help `configure` by setting `CPPFLAGS` and `LDFLAGS`. Here is an example:

```
$ CC="gcc" CFLAGS="-O2" CXX="g++" CXXFLAGS="-O2 -fno-exceptions" \
  CPPFLAGS="-I/opt/gmp/include" LDFLAGS="-L/opt/gmp/lib" ./configure --with-gmp
```

2.3 Installing the library

As with any autoconfiguring GNU software, installation is as easy as this:

```
$ make install
```

The '`make install`' command installs the library and the include files into public places ('`/usr/local/lib/`' and '`/usr/local/include/`', if you haven't specified a `--prefix` option to `configure`). This step may require superuser privileges.

If you have already built the library and wish to install it, but didn't specify `--prefix=...` at configure time, just re-run `configure`, giving it the same options as the first time, plus the `--prefix=...` option.

2.4 Cleaning up

You can remove system-dependent files generated by `make` through

```
$ make clean
```

You can remove all files generated by `make`, thus reverting to a virgin distribution of CLN, through

```
$ make distclean
```

3 Ordinary number types

CLN implements the following class hierarchy:

```
                        Number
                      cl_number
                    <cln/number.h>
                          |
                          |
                 Real or complex number
                        cl_N
                    <cln/complex.h>
                          |
                          |
                     Real number
                        cl_R
                     <cln/real.h>
                          |
      +-------------------+-------------------+
      |                                       |
Rational number                    Floating-point number
    cl_RA                                   cl_F
<cln/rational.h>                       <cln/float.h>
      |                                       |
      |                +--------------+--------------+--------------+
   Integer             |              |              |              |
    cl_I          Short-Float    Single-Float   Double-Float     Long-Float
<cln/integer.h>      cl_SF          cl_FF          cl_DF           cl_LF
                 <cln/sfloat.h> <cln/ffloat.h> <cln/dfloat.h> <cln/lfloat.h>
```

The base class `cl_number` is an abstract base class. It is not useful to declare a variable of this type except if you want to completely disable compile-time type checking and use run-time type checking instead.

The class `cl_N` comprises real and complex numbers. There is no special class for complex numbers since complex numbers with imaginary part `0` are automatically converted to real numbers.

The class `cl_R` comprises real numbers of different kinds. It is an abstract class.

The class `cl_RA` comprises exact real numbers: rational numbers, including integers. There is no special class for non-integral rational numbers since rational numbers with denominator `1` are automatically converted to integers.

The class `cl_F` implements floating-point approximations to real numbers. It is an abstract class.

3.1 Exact numbers

Some numbers are represented as exact numbers: there is no loss of information when such a number is converted from its mathematical value to its internal representation. On exact numbers, the elementary operations (`+`, `-`, `*`, `/`, comparisons, ...) compute the completely correct result.

In CLN, the exact numbers are:

- rational numbers (including integers),

- complex numbers whose real and imaginary parts are both rational numbers.

Rational numbers are always normalized to the form *numerator*/*denominator* where the numerator and denominator are coprime integers and the denominator is positive. If the resulting denominator is `1`, the rational number is converted to an integer.

Small integers (typically in the range `-2^29...2^29-1`, for 32-bit machines) are especially efficient, because they consume no heap allocation. Otherwise the distinction between these immediate integers (called "fixnums") and heap allocated integers (called "bignums") is completely transparent.

3.2 Floating-point numbers

Not all real numbers can be represented exactly. (There is an easy mathematical proof for this: Only a countable set of numbers can be stored exactly in a computer, even if one assumes that it has unlimited storage. But there are uncountably many real numbers.) So some approximation is needed. CLN implements ordinary floating-point numbers, with mantissa and exponent.

The elementary operations (`+`, `-`, `*`, `/`, ...) only return approximate results. For example, the value of the expression `(cl_F) 0.3 + (cl_F) 0.4` prints as '`0.70000005`', not as '`0.7`'. Rounding errors like this one are inevitable when computing with floating-point numbers.

Nevertheless, CLN rounds the floating-point results of the operations `+`, `-`, `*`, `/`, `sqrt` according to the "round-to-even" rule: It first computes the exact mathematical result and then returns the floating-point number which is nearest to this. If two floating-point numbers are equally distant from the ideal result, the one with a `0` in its least significant mantissa bit is chosen.

Similarly, testing floating point numbers for equality '`x == y`' is gambling with random errors. Better check for '`abs(x - y) < epsilon`' for some well-chosen `epsilon`.

Floating point numbers come in four flavors:

- Short floats, type `cl_SF`. They have 1 sign bit, 8 exponent bits (including the exponent's sign), and 17 mantissa bits (including the "hidden" bit). They don't consume heap allocation.
- Single floats, type `cl_FF`. They have 1 sign bit, 8 exponent bits (including the exponent's sign), and 24 mantissa bits (including the "hidden" bit). In CLN, they are represented as IEEE single-precision floating point numbers. This corresponds closely to the C/C++ type '`float`'.
- Double floats, type `cl_DF`. They have 1 sign bit, 11 exponent bits (including the exponent's sign), and 53 mantissa bits (including the "hidden" bit). In CLN, they are represented as IEEE double-precision floating point numbers. This corresponds closely to the C/C++ type '`double`'.
- Long floats, type `cl_LF`. They have 1 sign bit, 32 exponent bits (including the exponent's sign), and n mantissa bits (including the "hidden" bit), where n >= 64. The precision of a long float is unlimited, but once created, a long float has a fixed precision. (No "lazy recomputation".)

Of course, computations with long floats are more expensive than those with smaller floating-point formats.

CLN does not implement features like NaNs, denormalized numbers and gradual underflow. If the exponent range of some floating-point type is too limited for your application, choose another floating-point type with larger exponent range.

As a user of CLN, you can forget about the differences between the four floating-point types and just declare all your floating-point variables as being of type `cl_F`. This has the advantage that when you change the precision of some computation (say, from `cl_DF` to `cl_LF`), you don't have to change the code, only the precision of the initial values. Also, many transcendental functions have been declared as returning a `cl_F` when the argument is a `cl_F`, but such declarations are missing for the types `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF`. (Such declarations would be wrong if the floating point contagion rule happened to change in the future.)

3.3 Complex numbers

Complex numbers, as implemented by the class `cl_N`, have a real part and an imaginary part, both real numbers. A complex number whose imaginary part is the exact number `0` is automatically converted to a real number.

Complex numbers can arise from real numbers alone, for example through application of `sqrt` or transcendental functions.

3.4 Conversions

Conversions from any class to any its superclasses ("base classes" in C++ terminology) is done automatically.

Conversions from the C built-in types '`long`' and '`unsigned long`' are provided for the classes `cl_I`, `cl_RA`, `cl_R`, `cl_N` and `cl_number`.

Conversions from the C built-in types '`int`' and '`unsigned int`' are provided for the classes `cl_I`, `cl_RA`, `cl_R`, `cl_N` and `cl_number`. However, these conversions emphasize efficiency. Their range is therefore limited:

- The conversion from '`int`' works only if the argument is < 2^29 and > -2^29.
- The conversion from '`unsigned int`' works only if the argument is < 2^29.

In a declaration like '`cl_I x = 10;`' the C++ compiler is able to do the conversion of `10` from '`int`' to '`cl_I`' at compile time already. On the other hand, code like '`cl_I x = 1000000000;`' is in error. So, if you want to be sure that an '`int`' whose magnitude is not guaranteed to be < 2^29 is correctly converted to a '`cl_I`', first convert it to a '`long`'. Similarly, if a large '`unsigned int`' is to be converted to a '`cl_I`', first convert it to an '`unsigned long`'.

Conversions from the C built-in type '`float`' are provided for the classes `cl_FF`, `cl_F`, `cl_R`, `cl_N` and `cl_number`.

Conversions from the C built-in type '`double`' are provided for the classes `cl_DF`, `cl_F`, `cl_R`, `cl_N` and `cl_number`.

Conversions from '`const char *`' are provided for the classes `cl_I`, `cl_RA`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF`, `cl_F`, `cl_R`, `cl_N`. The easiest way to specify a value which is outside of the range of the C++ built-in types is therefore to specify it as a string, like this:

```
cl_I order_of_rubiks_cube_group = "43252003274489856000";
```

Note that this conversion is done at runtime, not at compile-time.

Conversions from `cl_I` to the C built-in types '`int`', '`unsigned int`', '`long`', '`unsigned long`' are provided through the functions

`int cl_I_to_int (const cl_I& x)`
`unsigned int cl_I_to_uint (const cl_I& x)`
`long cl_I_to_long (const cl_I& x)`
`unsigned long cl_I_to_ulong (const cl_I& x)`

> Returns `x` as element of the C type *ctype*. If `x` is not representable in the range of *ctype*, a runtime error occurs.

Conversions from the classes `cl_I`, `cl_RA`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF`, `cl_F` and `cl_R` to the C built-in types '`float`' and '`double`' are provided through the functions

`float float_approx (const` *type*`& x)`
`double double_approx (const` *type*`& x)`

> Returns an approximation of `x` of C type *ctype*. If `abs(x)` is too close to 0 (underflow), 0 is returned. If `abs(x)` is too large (overflow), an IEEE infinity is returned.

Conversions from any class to any of its subclasses ("derived classes" in C++ terminology) are not provided. Instead, you can assert and check that a value belongs to a certain subclass, and return it as element of that class, using the '`As`' and '`The`' macros. `As(`*type*`)(`*value*`)` checks that *value* belongs to *type* and returns it as such. `The(`*type*`)(`*value*`)` assumes that *value* belongs to *type* and returns it as such. It is your responsibility to ensure that this assumption is valid. Since macros and namespaces don't go together well, there is an equivalent to '`The`': the template '`the`'.

Example:

```
cl_I x = ...;
if (!(x >= 0)) abort();
cl_I ten_x_a = The(cl_I)(expt(10,x)); // If x >= 0, 10^x is an integer.
                // In general, it would be a rational number.
cl_I ten_x_b = the<cl_I>(expt(10,x)); // The same as above.
```

4 Functions on numbers

Each of the number classes declares its mathematical operations in the corresponding include file. For example, if your code operates with objects of type cl_I, it should #include <cln/integer.h>.

4.1 Constructing numbers

Here is how to create number objects "from nothing".

4.1.1 Constructing integers

cl_I objects are most easily constructed from C integers and from strings. See Section 3.4 [Conversions], page 8.

4.1.2 Constructing rational numbers

cl_RA objects can be constructed from strings. The syntax for rational numbers is described in Section 5.1 [Internal and printed representation], page 29. Another standard way to produce a rational number is through application of 'operator /' or 'recip' on integers.

4.1.3 Constructing floating-point numbers

cl_F objects with low precision are most easily constructed from C 'float' and 'double'. See Section 3.4 [Conversions], page 8.

To construct a cl_F with high precision, you can use the conversion from 'const char *', but you have to specify the desired precision within the string. (See Section 5.1 [Internal and printed representation], page 29.) Example:

```
cl_F e = "0.271828182845904523536028747135266249775724709369996e+1_40";
```

will set 'e' to the given value, with a precision of 40 decimal digits.

The programmatic way to construct a cl_F with high precision is through the cl_float conversion function, see Section 4.11.1 [Conversion to floating-point numbers], page 25. For example, to compute e to 40 decimal places, first construct 1.0 to 40 decimal places and then apply the exponential function:

```
float_format_t precision = float_format(40);
cl_F e = exp(cl_float(1,precision));
```

4.1.4 Constructing complex numbers

Non-real cl_N objects are normally constructed through the function

```
cl_N complex (const cl_R& realpart, const cl_R& imagpart)
```

See Section 4.4 [Elementary complex functions], page 12.

4.2 Elementary functions

Each of the classes `cl_N`, `cl_R`, `cl_RA`, `cl_I`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operations:

type `operator + (const` *type*`&, const` *type*`&)`
: Addition.

type `operator - (const` *type*`&, const` *type*`&)`
: Subtraction.

type `operator - (const` *type*`&)`
: Returns the negative of the argument.

type `plus1 (const` *type*`& x)`
: Returns `x + 1`.

type `minus1 (const` *type*`& x)`
: Returns `x - 1`.

type `operator * (const` *type*`&, const` *type*`&)`
: Multiplication.

type `square (const` *type*`& x)`
: Returns `x * x`.

Each of the classes `cl_N`, `cl_R`, `cl_RA`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operations:

type `operator / (const` *type*`&, const` *type*`&)`
: Division.

type `recip (const` *type*`&)`
: Returns the reciprocal of the argument.

The class `cl_I` doesn't define a '`/`' operation because in the C/C++ language this operator, applied to integral types, denotes the '`floor`' or '`truncate`' operation (which one of these, is implementation dependent). (See Section 4.6 [Rounding functions], page 13.) Instead, `cl_I` defines an "exact quotient" function:

`cl_I exquo (const cl_I& x, const cl_I& y)`
: Checks that `y` divides `x`, and returns the quotient `x/y`.

The following exponentiation functions are defined:

`cl_I expt_pos (const cl_I& x, const cl_I& y)`
`cl_RA expt_pos (const cl_RA& x, const cl_I& y)`
: `y` must be > 0. Returns `x^y`.

`cl_RA expt (const cl_RA& x, const cl_I& y)`
`cl_R expt (const cl_R& x, const cl_I& y)`
`cl_N expt (const cl_N& x, const cl_I& y)`
: Returns `x^y`.

Each of the classes `cl_R`, `cl_RA`, `cl_I`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operation:

type `abs (const` *type*`& x)`
: Returns the absolute value of `x`. This is `x` if `x >= 0`, and `-x` if `x <= 0`.

The class `cl_N` implements this as follows:

`cl_R abs (const cl_N x)`
: Returns the absolute value of `x`.

Each of the classes `cl_N`, `cl_R`, `cl_RA`, `cl_I`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operation:

type `signum (const` *type*`& x)`
: Returns the sign of `x`, in the same number format as `x`. This is defined as `x / abs(x)` if `x` is non-zero, and `x` if `x` is zero. If `x` is real, the value is either 0 or 1 or -1.

4.3 Elementary rational functions

Each of the classes `cl_RA`, `cl_I` defines the following operations:

`cl_I numerator (const` *type*`& x)`
: Returns the numerator of `x`.

`cl_I denominator (const` *type*`& x)`
: Returns the denominator of `x`.

The numerator and denominator of a rational number are normalized in such a way that they have no factor in common and the denominator is positive.

4.4 Elementary complex functions

The class `cl_N` defines the following operation:

`cl_N complex (const cl_R& a, const cl_R& b)`
: Returns the complex number `a+bi`, that is, the complex number with real part `a` and imaginary part `b`.

Each of the classes `cl_N`, `cl_R` defines the following operations:

`cl_R realpart (const` *type*`& x)`
: Returns the real part of `x`.

`cl_R imagpart (const` *type*`& x)`
: Returns the imaginary part of `x`.

type `conjugate (const` *type*`& x)`
: Returns the complex conjugate of `x`.

We have the relations

```
x = complex(realpart(x), imagpart(x))
conjugate(x) = complex(realpart(x), -imagpart(x))
```

4.5 Comparisons

Each of the classes `cl_N`, `cl_R`, `cl_RA`, `cl_I`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operations:

bool operator == (const *type*&, const *type*&)
bool operator != (const *type*&, const *type*&)
    Comparison, as in C and C++.

uint32 equal_hashcode (const *type*&)
    Returns a 32-bit hash code that is the same for any two numbers which are the same according to `==`. This hash code depends on the number's value, not its type or precision.

cl_boolean zerop (const *type*& x)
    Compare against zero: `x == 0`

Each of the classes `cl_R`, `cl_RA`, `cl_I`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operations:

cl_signean compare (const *type*& x, const *type*& y)
    Compares `x` and `y`. Returns +1 if `x`>`y`, -1 if `x`<`y`, 0 if `x`=`y`.

bool operator <= (const *type*&, const *type*&)
bool operator < (const *type*&, const *type*&)
bool operator >= (const *type*&, const *type*&)
bool operator > (const *type*&, const *type*&)
    Comparison, as in C and C++.

cl_boolean minusp (const *type*& x)
    Compare against zero: `x < 0`

cl_boolean plusp (const *type*& x)
    Compare against zero: `x > 0`

type max (const *type*& x, const *type*& y)
    Return the maximum of `x` and `y`.

type min (const *type*& x, const *type*& y)
    Return the minimum of `x` and `y`.

When a floating point number and a rational number are compared, the float is first converted to a rational number using the function `rational`. Since a floating point number actually represents an interval of real numbers, the result might be surprising. For example, `(cl_F)(cl_R)"1/3" == (cl_R)"1/3"` returns false because there is no floating point number whose value is exactly `1/3`.

4.6 Rounding functions

When a real number is to be converted to an integer, there is no "best" rounding. The desired rounding function depends on the application. The Common Lisp and ISO Lisp standards offer four rounding functions:

`floor(x)` This is the largest integer <=`x`.

`ceiling(x)`
    This is the smallest integer >=`x`.

`truncate(x)`
Among the integers between 0 and `x` (inclusive) the one nearest to `x`.

`round(x)` The integer nearest to `x`. If `x` is exactly halfway between two integers, choose the even one.

These functions have different advantages:

`floor` and `ceiling` are translation invariant: `floor(x+n) = floor(x) + n` and `ceiling(x+n) = ceiling(x) + n` for every `x` and every integer `n`.

On the other hand, `truncate` and `round` are symmetric: `truncate(-x) = -truncate(x)` and `round(-x) = -round(x)`, and furthermore `round` is unbiased: on the "average", it rounds down exactly as often as it rounds up.

The functions are related like this:

`ceiling(m/n) = floor((m+n-1)/n) = floor((m-1)/n)+1` for rational numbers `m/n` (`m`, `n` integers, `n`>0), and

`truncate(x) = sign(x) * floor(abs(x))`

Each of the classes `cl_R`, `cl_RA`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operations:

`cl_I floor1 (const` *type*`& x)`
Returns `floor(x)`.

`cl_I ceiling1 (const` *type*`& x)`
Returns `ceiling(x)`.

`cl_I truncate1 (const` *type*`& x)`
Returns `truncate(x)`.

`cl_I round1 (const` *type*`& x)`
Returns `round(x)`.

Each of the classes `cl_R`, `cl_RA`, `cl_I`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operations:

`cl_I floor1 (const` *type*`& x, const` *type*`& y)`
Returns `floor(x/y)`.

`cl_I ceiling1 (const` *type*`& x, const` *type*`& y)`
Returns `ceiling(x/y)`.

`cl_I truncate1 (const` *type*`& x, const` *type*`& y)`
Returns `truncate(x/y)`.

`cl_I round1 (const` *type*`& x, const` *type*`& y)`
Returns `round(x/y)`.

These functions are called '`floor1`', ... here instead of '`floor`', ..., because on some systems, system dependent include files define '`floor`' and '`ceiling`' as macros.

In many cases, one needs both the quotient and the remainder of a division. It is more efficient to compute both at the same time than to perform two divisions, one for quotient and the next one for the remainder. The following functions therefore return a structure containing both the quotient and the remainder. The suffix '`2`' indicates the number of "return values". The remainder is defined as follows:

- for the computation of `quotient = floor(x)`, `remainder = x - quotient`,
- for the computation of `quotient = floor(x,y)`, `remainder = x - quotient*y`,

and similarly for the other three operations.

Each of the classes `cl_R`, `cl_RA`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operations:

```
struct type_div_t { cl_I quotient; type remainder; };
type_div_t floor2 (const type& x)
type_div_t ceiling2 (const type& x)
type_div_t truncate2 (const type& x)
type_div_t round2 (const type& x)
```

Each of the classes `cl_R`, `cl_RA`, `cl_I`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operations:

```
struct type_div_t { cl_I quotient; type remainder; };
type_div_t floor2 (const type& x, const type& y)
type_div_t ceiling2 (const type& x, const type& y)
type_div_t truncate2 (const type& x, const type& y)
type_div_t round2 (const type& x, const type& y)
```

Sometimes, one wants the quotient as a floating-point number (of the same format as the argument, if the argument is a float) instead of as an integer. The prefix '`f`' indicates this.

Each of the classes `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operations:

```
type ffloor (const type& x)
type fceiling (const type& x)
type ftruncate (const type& x)
type fround (const type& x)
```

and similarly for class `cl_R`, but with return type `cl_F`.

The class `cl_R` defines the following operations:

```
cl_F ffloor (const type& x, const type& y)
cl_F fceiling (const type& x, const type& y)
cl_F ftruncate (const type& x, const type& y)
cl_F fround (const type& x, const type& y)
```

These functions also exist in versions which return both the quotient and the remainder. The suffix '2' indicates this.

Each of the classes `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operations:

```
struct type_fdiv_t { type quotient; type remainder; };
type_fdiv_t ffloor2 (const type& x)
type_fdiv_t fceiling2 (const type& x)
type_fdiv_t ftruncate2 (const type& x)
type_fdiv_t fround2 (const type& x)
```

and similarly for class `cl_R`, but with quotient type `cl_F`.

The class `cl_R` defines the following operations:

struct *type*_fdiv_t { cl_F quotient; cl_R remainder; };
*type*_fdiv_t ffloor2 (const *type*& x, const *type*& y)
*type*_fdiv_t fceiling2 (const *type*& x, const *type*& y)
*type*_fdiv_t ftruncate2 (const *type*& x, const *type*& y)
*type*_fdiv_t fround2 (const *type*& x, const *type*& y)

Other applications need only the remainder of a division. The remainder of 'floor' and 'ffloor' is called 'mod' (abbreviation of "modulo"). The remainder 'truncate' and 'ftruncate' is called 'rem' (abbreviation of "remainder").

- mod(x,y) = floor2(x,y).remainder = x - floor(x/y)*y
- rem(x,y) = truncate2(x,y).remainder = x - truncate(x/y)*y

If x and y are both >= 0, mod(x,y) = rem(x,y) >= 0. In general, mod(x,y) has the sign of y or is zero, and rem(x,y) has the sign of x or is zero.

The classes cl_R, cl_I define the following operations:

type mod (const *type*& x, const *type*& y)
type rem (const *type*& x, const *type*& y)

4.7 Roots

Each of the classes cl_R, cl_F, cl_SF, cl_FF, cl_DF, cl_LF defines the following operation:

type sqrt (const *type*& x)
: x must be >= 0. This function returns the square root of x, normalized to be >= 0. If x is the square of a rational number, sqrt(x) will be a rational number, else it will return a floating-point approximation.

The classes cl_RA, cl_I define the following operation:

cl_boolean sqrtp (const *type*& x, *type** root)
: This tests whether x is a perfect square. If so, it returns true and the exact square root in *root, else it returns false.

Furthermore, for integers, similarly:

cl_boolean isqrt (const *type*& x, *type** root)
: x should be >= 0. This function sets *root to floor(sqrt(x)) and returns the same value as sqrtp: the boolean value (expt(*root,2) == x).

For nth roots, the classes cl_RA, cl_I define the following operation:

cl_boolean rootp (const *type*& x, const cl_I& n, *type** root)
: x must be >= 0. n must be > 0. This tests whether x is an nth power of a rational number. If so, it returns true and the exact root in *root, else it returns false.

The only square root function which accepts negative numbers is the one for class cl_N:

cl_N sqrt (const cl_N& z)
: Returns the square root of z, as defined by the formula sqrt(z) = exp(log(z)/2). Conversion to a floating-point type or to a complex number are done if necessary. The range of the result is the right half plane realpart(sqrt(z)) >= 0 including the positive imaginary axis and 0, but excluding the negative imaginary axis. The result is an exact number only if z is an exact number.

4.8 Transcendental functions

The transcendental functions return an exact result if the argument is exact and the result is exact as well. Otherwise they must return inexact numbers even if the argument is exact. For example, `cos(0) = 1` returns the rational number `1`.

4.8.1 Exponential and logarithmic functions

`cl_R exp (const cl_R& x)`
`cl_N exp (const cl_N& x)`

Returns the exponential function of `x`. This is `e^x` where `e` is the base of the natural logarithms. The range of the result is the entire complex plane excluding 0.

`cl_R ln (const cl_R& x)`

`x` must be > 0. Returns the (natural) logarithm of x.

`cl_N log (const cl_N& x)`

Returns the (natural) logarithm of x. If `x` is real and positive, this is `ln(x)`. In general, `log(x) = log(abs(x)) + i*phase(x)`. The range of the result is the strip in the complex plane `-pi < imagpart(log(x)) <= pi`.

`cl_R phase (const cl_N& x)`

Returns the angle part of `x` in its polar representation as a complex number. That is, `phase(x) = atan(realpart(x),imagpart(x))`. This is also the imaginary part of `log(x)`. The range of the result is the interval `-pi < phase(x) <= pi`. The result will be an exact number only if `zerop(x)` or if `x` is real and positive.

`cl_R log (const cl_R& a, const cl_R& b)`

`a` and `b` must be > 0. Returns the logarithm of `a` with respect to base `b`. `log(a,b) = ln(a)/ln(b)`. The result can be exact only if `a = 1` or if `a` and `b` are both rational.

`cl_N log (const cl_N& a, const cl_N& b)`

Returns the logarithm of `a` with respect to base `b`. `log(a,b) = log(a)/log(b)`.

`cl_N expt (const cl_N& x, const cl_N& y)`

Exponentiation: Returns `x^y = exp(y*log(x))`.

The constant e = exp(1) = 2.71828. . . is returned by the following functions:

`cl_F exp1 (float_format_t f)`

Returns e as a float of format `f`.

`cl_F exp1 (const cl_F& y)`

Returns e in the float format of `y`.

`cl_F exp1 (void)`

Returns e as a float of format `default_float_format`.

4.8.2 Trigonometric functions

`cl_R sin (const cl_R& x)`

Returns `sin(x)`. The range of the result is the interval `-1 <= sin(x) <= 1`.

`cl_N sin (const cl_N& z)`
: Returns `sin(z)`. The range of the result is the entire complex plane.

`cl_R cos (const cl_R& x)`
: Returns `cos(x)`. The range of the result is the interval `-1 <= cos(x) <= 1`.

`cl_N cos (const cl_N& x)`
: Returns `cos(z)`. The range of the result is the entire complex plane.

`struct cos_sin_t { cl_R cos; cl_R sin; };`
`cos_sin_t cos_sin (const cl_R& x)`
: Returns both `sin(x)` and `cos(x)`. This is more efficient than computing them separately. The relation `cos^2 + sin^2 = 1` will hold only approximately.

`cl_R tan (const cl_R& x)`
`cl_N tan (const cl_N& x)`
: Returns `tan(x) = sin(x)/cos(x)`.

`cl_N cis (const cl_R& x)`
`cl_N cis (const cl_N& x)`
: Returns `exp(i*x)`. The name '`cis`' means "cos + i sin", because `e^(i*x) = cos(x) + i*sin(x)`.

`cl_N asin (const cl_N& z)`
: Returns `arcsin(z)`. This is defined as `arcsin(z) = log(iz+sqrt(1-z^2))/i` and satisfies `arcsin(-z) = -arcsin(z)`. The range of the result is the strip in the complex domain `-pi/2 <= realpart(arcsin(z)) <= pi/2`, excluding the numbers with `realpart = -pi/2` and `imagpart < 0` and the numbers with `realpart = pi/2` and `imagpart > 0`.

`cl_N acos (const cl_N& z)`
: Returns `arccos(z)`. This is defined as `arccos(z) = pi/2 - arcsin(z) = log(z+i*sqrt(1-z^2))/i` and satisfies `arccos(-z) = pi - arccos(z)`. The range of the result is the strip in the complex domain `0 <= realpart(arcsin(z)) <= pi`, excluding the numbers with `realpart = 0` and `imagpart < 0` and the numbers with `realpart = pi` and `imagpart > 0`.

`cl_R atan (const cl_R& x, const cl_R& y)`
: Returns the angle of the polar representation of the complex number `x+iy`. This is `atan(y/x)` if `x>0`. The range of the result is the interval `-pi < atan(x,y) <= pi`. The result will be an exact number only if `x > 0` and `y` is the exact `0`. WARNING: In Common Lisp, this function is called as `(atan y x)`, with reversed order of arguments.

`cl_R atan (const cl_R& x)`
: Returns `arctan(x)`. This is the same as `atan(1,x)`. The range of the result is the interval `-pi/2 < atan(x) < pi/2`. The result will be an exact number only if `x` is the exact `0`.

`cl_N atan (const cl_N& z)`
: Returns `arctan(z)`. This is defined as `arctan(z) = (log(1+iz)-log(1-iz)) / 2i` and satisfies `arctan(-z) = -arctan(z)`. The range of the result is the strip in the complex domain `-pi/2 <= realpart(arctan(z)) <= pi/2`, excluding the numbers with `realpart = -pi/2` and `imagpart >= 0` and the numbers with `realpart = pi/2` and `imagpart <= 0`.

Archimedes' constant pi = 3.14. . . is returned by the following functions:

`cl_F pi (float_format_t f)`
: Returns pi as a float of format `f`.

`cl_F pi (const cl_F& y)`
: Returns pi in the float format of `y`.

`cl_F pi (void)`
: Returns pi as a float of format `default_float_format`.

4.8.3 Hyperbolic functions

`cl_R sinh (const cl_R& x)`
: Returns `sinh(x)`.

`cl_N sinh (const cl_N& z)`
: Returns `sinh(z)`. The range of the result is the entire complex plane.

`cl_R cosh (const cl_R& x)`
: Returns `cosh(x)`. The range of the result is the interval `cosh(x) >= 1`.

`cl_N cosh (const cl_N& z)`
: Returns `cosh(z)`. The range of the result is the entire complex plane.

`struct cosh_sinh_t { cl_R cosh; cl_R sinh; };`
`cosh_sinh_t cosh_sinh (const cl_R& x)`
: Returns both `sinh(x)` and `cosh(x)`. This is more efficient than computing them separately. The relation `cosh^2 - sinh^2 = 1` will hold only approximately.

`cl_R tanh (const cl_R& x)`
`cl_N tanh (const cl_N& x)`
: Returns `tanh(x) = sinh(x)/cosh(x)`.

`cl_N asinh (const cl_N& z)`
: Returns `arsinh(z)`. This is defined as `arsinh(z) = log(z+sqrt(1+z^2))` and satisfies `arsinh(-z) = -arsinh(z)`. The range of the result is the strip in the complex domain `-pi/2 <= imagpart(arsinh(z)) <= pi/2`, excluding the numbers with `imagpart = -pi/2` and `realpart > 0` and the numbers with `imagpart = pi/2` and `realpart < 0`.

`cl_N acosh (const cl_N& z)`
: Returns `arcosh(z)`. This is defined as `arcosh(z) = 2*log(sqrt((z+1)/2)+sqrt((z-1)/2))`. The range of the result is the half-strip in the complex domain `-pi < imagpart(arcosh(z)) <= pi, realpart(arcosh(z)) >= 0`, excluding the numbers with `realpart = 0` and `-pi < imagpart < 0`.

`cl_N atanh (const cl_N& z)`
: Returns `artanh(z)`. This is defined as `artanh(z) = (log(1+z)-log(1-z)) / 2` and satisfies `artanh(-z) = -artanh(z)`. The range of the result is the strip in the complex domain `-pi/2 <= imagpart(artanh(z)) <= pi/2`, excluding the numbers with `imagpart = -pi/2` and `realpart <= 0` and the numbers with `imagpart = pi/2` and `realpart >= 0`.

4.8.4 Euler gamma

Euler's constant C = 0.577... is returned by the following functions:

`cl_F eulerconst (float_format_t f)`
: Returns Euler's constant as a float of format `f`.

`cl_F eulerconst (const cl_F& y)`
: Returns Euler's constant in the float format of `y`.

`cl_F eulerconst (void)`
: Returns Euler's constant as a float of format `default_float_format`.

Catalan's constant G = 0.915... is returned by the following functions:

`cl_F catalanconst (float_format_t f)`
: Returns Catalan's constant as a float of format `f`.

`cl_F catalanconst (const cl_F& y)`
: Returns Catalan's constant in the float format of `y`.

`cl_F catalanconst (void)`
: Returns Catalan's constant as a float of format `default_float_format`.

4.8.5 Riemann zeta

Riemann's zeta function at an integral point `s>1` is returned by the following functions:

`cl_F zeta (int s, float_format_t f)`
: Returns Riemann's zeta function at `s` as a float of format `f`.

`cl_F zeta (int s, const cl_F& y)`
: Returns Riemann's zeta function at `s` in the float format of `y`.

`cl_F zeta (int s)`
: Returns Riemann's zeta function at `s` as a float of format `default_float_format`.

4.9 Functions on integers

4.9.1 Logical functions

Integers, when viewed as in two's complement notation, can be thought as infinite bit strings where the bits' values eventually are constant. For example,

```
17 = ......00010001
-6 = ......11111010
```

The logical operations view integers as such bit strings and operate on each of the bit positions in parallel.

`cl_I lognot (const cl_I& x)`
`cl_I operator ~ (const cl_I& x)`
: Logical not, like `~x` in C. This is the same as `-1-x`.

`cl_I logand (const cl_I& x, const cl_I& y)`
`cl_I operator & (const cl_I& x, const cl_I& y)`
: Logical and, like `x & y` in C.

`cl_I logior (const cl_I& x, const cl_I& y)`
`cl_I operator | (const cl_I& x, const cl_I& y)`
: Logical (inclusive) or, like `x | y` in C.

`cl_I logxor (const cl_I& x, const cl_I& y)`
`cl_I operator ^ (const cl_I& x, const cl_I& y)`
: Exclusive or, like `x ^ y` in C.

`cl_I logeqv (const cl_I& x, const cl_I& y)`
: Bitwise equivalence, like `~(x ^ y)` in C.

`cl_I lognand (const cl_I& x, const cl_I& y)`
: Bitwise not and, like `~(x & y)` in C.

`cl_I lognor (const cl_I& x, const cl_I& y)`
: Bitwise not or, like `~(x | y)` in C.

`cl_I logandc1 (const cl_I& x, const cl_I& y)`
: Logical and, complementing the first argument, like `~x & y` in C.

`cl_I logandc2 (const cl_I& x, const cl_I& y)`
: Logical and, complementing the second argument, like `x & ~y` in C.

`cl_I logorc1 (const cl_I& x, const cl_I& y)`
: Logical or, complementing the first argument, like `~x | y` in C.

`cl_I logorc2 (const cl_I& x, const cl_I& y)`
: Logical or, complementing the second argument, like `x | ~y` in C.

These operations are all available though the function

`cl_I boole (cl_boole op, const cl_I& x, const cl_I& y)`

where `op` must have one of the 16 values (each one stands for a function which combines two bits into one bit): `boole_clr`, `boole_set`, `boole_1`, `boole_2`, `boole_c1`, `boole_c2`, `boole_and`, `boole_ior`, `boole_xor`, `boole_eqv`, `boole_nand`, `boole_nor`, `boole_andc1`, `boole_andc2`, `boole_orc1`, `boole_orc2`.

Other functions that view integers as bit strings:

`cl_boolean logtest (const cl_I& x, const cl_I& y)`
: Returns true if some bit is set in both `x` and `y`, i.e. if `logand(x,y) != 0`.

`cl_boolean logbitp (const cl_I& n, const cl_I& x)`
: Returns true if the `n`th bit (from the right) of `x` is set. Bit 0 is the least significant bit.

`uintL logcount (const cl_I& x)`
: Returns the number of one bits in `x`, if `x` >= 0, or the number of zero bits in `x`, if `x` < 0.

The following functions operate on intervals of bits in integers. The type

```
struct cl_byte { uintL size; uintL position; };
```

represents the bit interval containing the bits `position`...`position+size-1` of an integer. The constructor `cl_byte(size,position)` constructs a `cl_byte`.

`cl_I ldb (const cl_I& n, const cl_byte& b)`
: extracts the bits of `n` described by the bit interval `b` and returns them as a nonnegative integer with `b.size` bits.

`cl_boolean ldb_test (const cl_I& n, const cl_byte& b)`
: Returns true if some bit described by the bit interval `b` is set in `n`.

`cl_I dpb (const cl_I& newbyte, const cl_I& n, const cl_byte& b)`
: Returns `n`, with the bits described by the bit interval `b` replaced by `newbyte`. Only the lowest `b.size` bits of `newbyte` are relevant.

The functions `ldb` and `dpb` implicitly shift. The following functions are their counterparts without shifting:

`cl_I mask_field (const cl_I& n, const cl_byte& b)`
: returns an integer with the bits described by the bit interval `b` copied from the corresponding bits in `n`, the other bits zero.

`cl_I deposit_field (const cl_I& newbyte, const cl_I& n, const cl_byte& b)`
: returns an integer where the bits described by the bit interval `b` come from `newbyte` and the other bits come from `n`.

The following relations hold:

`ldb (n, b) = mask_field(n, b) >> b.position`,

`dpb (newbyte, n, b) = deposit_field (newbyte << b.position, n, b)`,

`deposit_field(newbyte,n,b) = n ^ mask_field(n,b) ^ mask_field(new_byte,b)`.

The following operations on integers as bit strings are efficient shortcuts for common arithmetic operations:

`cl_boolean oddp (const cl_I& x)`
: Returns true if the least significant bit of `x` is 1. Equivalent to `mod(x,2) != 0`.

`cl_boolean evenp (const cl_I& x)`
: Returns true if the least significant bit of `x` is 0. Equivalent to `mod(x,2) == 0`.

`cl_I operator << (const cl_I& x, const cl_I& n)`
: Shifts `x` by `n` bits to the left. `n` should be >=0. Equivalent to `x * expt(2,n)`.

`cl_I operator >> (const cl_I& x, const cl_I& n)`
: Shifts `x` by `n` bits to the right. `n` should be >=0. Bits shifted out to the right are thrown away. Equivalent to `floor(x / expt(2,n))`.

`cl_I ash (const cl_I& x, const cl_I& y)`
: Shifts `x` by `y` bits to the left (if `y`>=0) or by `-y` bits to the right (if `y`<=0). In other words, this returns `floor(x * expt(2,y))`.

`uintL integer_length (const cl_I& x)`
: Returns the number of bits (excluding the sign bit) needed to represent `x` in two's complement notation. This is the smallest n >= 0 such that -2^n <= x < 2^n. If x > 0, this is the unique n > 0 such that 2^(n-1) <= x < 2^n.

`uintL ord2 (const cl_I& x)`
: `x` must be non-zero. This function returns the number of 0 bits at the right of `x` in two's complement notation. This is the largest n >= 0 such that 2^n divides `x`.

`uintL power2p (const cl_I& x)`
: `x` must be > 0. This function checks whether `x` is a power of 2. If `x` = 2^(n-1), it returns n. Else it returns 0. (See also the function `logp`.)

4.9.2 Number theoretic functions

`uint32 gcd (uint32 a, uint32 b)`
`cl_I gcd (const cl_I& a, const cl_I& b)`
: This function returns the greatest common divisor of `a` and `b`, normalized to be >= 0.

`cl_I xgcd (const cl_I& a, const cl_I& b, cl_I* u, cl_I* v)`
: This function ("extended gcd") returns the greatest common divisor `g` of `a` and `b` and at the same time the representation of `g` as an integral linear combination of `a` and `b`: `u` and `v` with `u*a+v*b = g`, `g` >= 0. `u` and `v` will be normalized to be of smallest possible absolute value, in the following sense: If `a` and `b` are non-zero, and `abs(a) != abs(b)`, `u` and `v` will satisfy the inequalities `abs(u) <= abs(b)/(2*g)`, `abs(v) <= abs(a)/(2*g)`.

`cl_I lcm (const cl_I& a, const cl_I& b)`
: This function returns the least common multiple of `a` and `b`, normalized to be >= 0.

`cl_boolean logp (const cl_I& a, const cl_I& b, cl_RA* l)`
`cl_boolean logp (const cl_RA& a, const cl_RA& b, cl_RA* l)`
: `a` must be > 0. `b` must be >0 and != 1. If log(a,b) is rational number, this function returns true and sets *l = log(a,b), else it returns false.

`int jacobi (sint32 a, sint32 b)`
`int jacobi (const cl_I& a, const cl_I& b)`
: Returns the Jacobi symbol $\left(\frac{a}{b}\right)$, `a,b` must be integers, `b>0` and odd. The result is 0 iff gcd(a,b)>1.

`cl_boolean isprobprime (const cl_I& n)`
: Returns true if `n` is a small prime or passes the Miller-Rabin primality test. The probability of a false positive is 1:10^30.

`cl_I nextprobprime (const cl_R& x)`
: Returns the smallest probable prime >=`x`.

4.9.3 Combinatorial functions

`cl_I factorial (uintL n)`
: `n` must be a small integer >= 0. This function returns the factorial `n`! = `1*2*...*n`.

`cl_I doublefactorial (uintL n)`
: `n` must be a small integer >= 0. This function returns the doublefactorial `n`!! = `1*3*...*n` or `n`!! = `2*4*...*n`, respectively.

`cl_I binomial (uintL n, uintL k)`
: `n` and `k` must be small integers >= 0. This function returns the binomial coefficient $\binom{n}{k} = \frac{n!}{n!(n-k)!}$ for 0 <= k <= n, 0 else.

4.10 Functions on floating-point numbers

Recall that a floating-point number consists of a sign `s`, an exponent `e` and a mantissa `m`. The value of the number is `(-1)^s * 2^e * m`.

Each of the classes `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines the following operations.

type `scale_float (const` *type*`& x, sintL delta)`
type `scale_float (const` *type*`& x, const cl_I& delta)`
: Returns `x*2^delta`. This is more efficient than an explicit multiplication because it copies `x` and modifies the exponent.

The following functions provide an abstract interface to the underlying representation of floating-point numbers.

`sintL float_exponent (const` *type*`& x)`
: Returns the exponent `e` of `x`. For `x = 0.0`, this is 0. For `x` non-zero, this is the unique integer with `2^(e-1) <= abs(x) < 2^e`.

`sintL float_radix (const` *type*`& x)`
: Returns the base of the floating-point representation. This is always `2`.

type `float_sign (const` *type*`& x)`
: Returns the sign `s` of `x` as a float. The value is 1 for `x` >= 0, -1 for `x < 0`.

`uintL float_digits (const` *type*`& x)`
: Returns the number of mantissa bits in the floating-point representation of `x`, including the hidden bit. The value only depends on the type of `x`, not on its value.

`uintL float_precision (const` *type*`& x)`
: Returns the number of significant mantissa bits in the floating-point representation of `x`. Since denormalized numbers are not supported, this is the same as `float_digits(x)` if `x` is non-zero, and 0 if `x` = 0.

The complete internal representation of a float is encoded in the type `decoded_float` (or `decoded_sfloat`, `decoded_ffloat`, `decoded_dfloat`, `decoded_lfloat`, respectively), defined by

```
struct decoded_typefloat {
        type mantissa; cl_I exponent; type sign;
};
```

and returned by the function

`decoded_`*type*`float decode_float (const` *type*`& x)`
: For `x` non-zero, this returns `(-1)^s`, `e`, `m` with `x = (-1)^s * 2^e * m` and `0.5 <= m < 1.0`. For `x` = 0, it returns `(-1)^s`=1, `e`=0, `m`=0. `e` is the same as returned by the function `float_exponent`.

A complete decoding in terms of integers is provided as type

```
struct cl_idecoded_float {
        cl_I mantissa; cl_I exponent; cl_I sign;
};
```

by the following function:

`cl_idecoded_float integer_decode_float (const type& x)`
: For `x` non-zero, this returns `(-1)^s`, `e`, `m` with `x = (-1)^s * 2^e * m` and `m` an integer with `float_digits(x)` bits. For `x` = 0, it returns `(-1)^s`=1, `e`=0, `m`=0. WARNING: The exponent `e` is not the same as the one returned by the functions `decode_float` and `float_exponent`.

Some other function, implemented only for class `cl_F`:

`cl_F float_sign (const cl_F& x, const cl_F& y)`
: This returns a floating point number whose precision and absolute value is that of `y` and whose sign is that of `x`. If `x` is zero, it is treated as positive. Same for `y`.

4.11 Conversion functions

4.11.1 Conversion to floating-point numbers

The type `float_format_t` describes a floating-point format.

`float_format_t float_format (uintL n)`
: Returns the smallest float format which guarantees at least `n` decimal digits in the mantissa (after the decimal point).

`float_format_t float_format (const cl_F& x)`
: Returns the floating point format of `x`.

`float_format_t default_float_format`
: Global variable: the default float format used when converting rational numbers to floats.

To convert a real number to a float, each of the types `cl_R`, `cl_F`, `cl_I`, `cl_RA`, `int`, `unsigned int`, `float`, `double` defines the following operations:

`cl_F cl_float (const type&x, float_format_t f)`
: Returns `x` as a float of format `f`.

`cl_F cl_float (const type&x, const cl_F& y)`
: Returns `x` in the float format of `y`.

`cl_F cl_float (const type&x)`
: Returns `x` as a float of format `default_float_format` if it is an exact number, or `x` itself if it is already a float.

Of course, converting a number to a float can lose precision.

Every floating-point format has some characteristic numbers:

`cl_F most_positive_float (float_format_t f)`
: Returns the largest (most positive) floating point number in float format `f`.

`cl_F most_negative_float (float_format_t f)`
: Returns the smallest (most negative) floating point number in float format `f`.

`cl_F least_positive_float (float_format_t f)`
: Returns the least positive floating point number (i.e. > 0 but closest to 0) in float format `f`.

`cl_F least_negative_float (float_format_t f)`
Returns the least negative floating point number (i.e. < 0 but closest to 0) in float format `f`.

`cl_F float_epsilon (float_format_t f)`
Returns the smallest floating point number e > 0 such that `1+e != 1`.

`cl_F float_negative_epsilon (float_format_t f)`
Returns the smallest floating point number e > 0 such that `1-e != 1`.

4.11.2 Conversion to rational numbers

Each of the classes `cl_R`, `cl_RA`, `cl_F` defines the following operation:

`cl_RA rational (const` *type*`& x)`
Returns the value of `x` as an exact number. If `x` is already an exact number, this is `x`. If `x` is a floating-point number, the value is a rational number whose denominator is a power of 2.

In order to convert back, say, `(cl_F)(cl_R)"1/3"` to `1/3`, there is the function

`cl_RA rationalize (const cl_R& x)`
If `x` is a floating-point number, it actually represents an interval of real numbers, and this function returns the rational number with smallest denominator (and smallest numerator, in magnitude) which lies in this interval. If `x` is already an exact number, this function returns `x`.

If `x` is any float, one has

```
cl_float(rational(x),x) = x
cl_float(rationalize(x),x) = x
```

4.12 Random number generators

A random generator is a machine which produces (pseudo-)random numbers. The include file `<cln/random.h>` defines a class `random_state` which contains the state of a random generator. If you make a copy of the random number generator, the original one and the copy will produce the same sequence of random numbers.

The following functions return (pseudo-)random numbers in different formats. Calling one of these modifies the state of the random number generator in a complicated but deterministic way.

The global variable

```
random_state default_random_state
```

contains a default random number generator. It is used when the functions below are called without `random_state` argument.

`uint32 random32 (random_state& randomstate)`
`uint32 random32 ()`
Returns a random unsigned 32-bit number. All bits are equally random.

`cl_I random_I (random_state& randomstate, const cl_I& n)`
`cl_I random_I (const cl_I& n)`

: `n` must be an integer > 0. This function returns a random integer `x` in the range `0 <= x < n`.

`cl_F random_F (random_state& randomstate, const cl_F& n)`
`cl_F random_F (const cl_F& n)`

: `n` must be a float > 0. This function returns a random floating-point number of the same format as `n` in the range `0 <= x < n`.

`cl_R random_R (random_state& randomstate, const cl_R& n)`
`cl_R random_R (const cl_R& n)`

: Behaves like `random_I` if `n` is an integer and like `random_F` if `n` is a float.

4.13 Obfuscating operators

The modifying C/C++ operators `+=`, `-=`, `*=`, `/=`, `&=`, `|=`, `^=`, `<<=`, `>>=` are not available by default because their use tends to make programs unreadable. It is trivial to get away without them. However, if you feel that you absolutely need these operators to get happy, then add

```
#define WANT_OBFUSCATING_OPERATORS
```

to the beginning of your source files, before the inclusion of any CLN include files. This flag will enable the following operators:

For the classes `cl_N`, `cl_R`, `cl_RA`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF`:

type`& operator += (`*type*`&, const `*type*`&)`
type`& operator -= (`*type*`&, const `*type*`&)`
type`& operator *= (`*type*`&, const `*type*`&)`
type`& operator /= (`*type*`&, const `*type*`&)`

For the class `cl_I`:

type`& operator += (`*type*`&, const `*type*`&)`
type`& operator -= (`*type*`&, const `*type*`&)`
type`& operator *= (`*type*`&, const `*type*`&)`
type`& operator &= (`*type*`&, const `*type*`&)`
type`& operator |= (`*type*`&, const `*type*`&)`
type`& operator ^= (`*type*`&, const `*type*`&)`
type`& operator <<= (`*type*`&, const `*type*`&)`
type`& operator >>= (`*type*`&, const `*type*`&)`

For the classes `cl_N`, `cl_R`, `cl_RA`, `cl_I`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF`:

type`& operator ++ (`*type*`& x)`

: The prefix operator `++x`.

`void operator ++ (`*type*`& x, int)`

: The postfix operator `x++`.

type`& operator -- (`*type*`& x)`

: The prefix operator `--x`.

`void operator -- (`*type*`& x, int)`

: The postfix operator `x--`.

Note that by using these obfuscating operators, you wouldn't gain efficiency: In CLN '`x += y;`' is exactly the same as '`x = x+y;`', not more efficient.

5 Input/Output

5.1 Internal and printed representation

All computations deal with the internal representations of the numbers.

Every number has an external representation as a sequence of ASCII characters. Several external representations may denote the same number, for example, "20.0" and "20.000".

Converting an internal to an external representation is called "printing", converting an external to an internal representation is called "reading". In CLN, it is always true that conversion of an internal to an external representation and then back to an internal representation will yield the same internal representation. Symbolically: `read(print(x)) == x`. This is called "print-read consistency".

Different types of numbers have different external representations (case is insignificant):

Integers
: External representation: *sign*{*digit*}+. The reader also accepts the Common Lisp syntaxes *sign*{*digit*}+. with a trailing dot for decimal integers and the `#nR`, `#b`, `#o`, `#x` prefixes.

Rational numbers
: External representation: *sign*{*digit*}+/{*digit*}+. The `#nR`, `#b`, `#o`, `#x` prefixes are allowed here as well.

Floating-point numbers
: External representation: *sign*{*digit*}**exponent* or *sign*{*digit*}*.{*digit*}**exponent* or *sign*{*digit*}*.{*digit*}+. A precision specifier of the form _*prec* may be appended. There must be at least one digit in the non-exponent part. The exponent has the syntax *expmarker expsign* {*digit*}+. The exponent marker is

  '`s`' for short-floats,

  '`f`' for single-floats,

  '`d`' for double-floats,

  '`L`' for long-floats,

  or '`e`', which denotes a default float format. The precision specifying suffix has the syntax _*prec* where *prec* denotes the number of valid mantissa digits (in decimal, excluding leading zeroes), cf. also function '`float_format`'.

Complex numbers
: External representation:

  In algebraic notation: *realpart*+*imagpart*`i`. Of course, if *imagpart* is negative, its printed representation begins with a '`-`', and the '`+`' between *realpart* and *imagpart* may be omitted. Note that this notation cannot be used when the *imagpart* is rational and the rational number's base is >18, because the '`i`' is then read as a digit.

  In Common Lisp notation: `#C(`*realpart imagpart*`)`.

5.2 Input functions

Including `<cln/io.h>` defines a number of simple input functions that read from `std::istream&`:

`int freadchar (std::istream& stream)`
: Reads a character from `stream`. Returns `cl_EOF` (not a '`char`'!) if the end of stream was encountered or an error occurred.

`int funreadchar (std::istream& stream, int c)`
: Puts back `c` onto `stream`. `c` must be the result of the last `freadchar` operation on `stream`.

Each of the classes `cl_N`, `cl_R`, `cl_RA`, `cl_I`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines, in `<cln/`*type*`_io.h>`, the following input function:

`std::istream& operator>> (std::istream& stream, `*type*`& result)`
: Reads a number from `stream` and stores it in the `result`.

The most flexible input functions, defined in `<cln/`*type*`_io.h>`, are the following:

`cl_N read_complex (std::istream& stream, const cl_read_flags& flags)`
`cl_R read_real (std::istream& stream, const cl_read_flags& flags)`
`cl_F read_float (std::istream& stream, const cl_read_flags& flags)`
`cl_RA read_rational (std::istream& stream, const cl_read_flags& flags)`
`cl_I read_integer (std::istream& stream, const cl_read_flags& flags)`
: Reads a number from `stream`. The `flags` are parameters which affect the input syntax. Whitespace before the number is silently skipped.

`cl_N read_complex (const cl_read_flags& flags, const char * string, const char * string_limit, const char * * end_of_parse)`
`cl_R read_real (const cl_read_flags& flags, const char * string, const char * string_limit, const char * * end_of_parse)`
`cl_F read_float (const cl_read_flags& flags, const char * string, const char * string_limit, const char * * end_of_parse)`
`cl_RA read_rational (const cl_read_flags& flags, const char * string, const char * string_limit, const char * * end_of_parse)`
`cl_I read_integer (const cl_read_flags& flags, const char * string, const char * string_limit, const char * * end_of_parse)`
: Reads a number from a string in memory. The `flags` are parameters which affect the input syntax. The string starts at `string` and ends at `string_limit` (exclusive limit). `string_limit` may also be `NULL`, denoting the entire string, i.e. equivalent to `string_limit = string + strlen(string)`. If `end_of_parse` is `NULL`, the string in memory must contain exactly one number and nothing more, else a fatal error will be signalled. If `end_of_parse` is not `NULL`, `*end_of_parse` will be assigned a pointer past the last parsed character (i.e. `string_limit` if nothing came after the number). Whitespace is not allowed.

The structure `cl_read_flags` contains the following fields:

`cl_read_syntax_t syntax`
: The possible results of the read operation. Possible values are `syntax_number`, `syntax_real`, `syntax_rational`, `syntax_integer`, `syntax_float`, `syntax_sfloat`, `syntax_ffloat`, `syntax_dfloat`, `syntax_lfloat`.

`cl_read_lsyntax_t lsyntax`
: Specifies the language-dependent syntax variant for the read operation. Possible values are

  `lsyntax_standard`
  : accept standard algebraic notation only, no complex numbers,

  `lsyntax_algebraic`
  : accept the algebraic notation *x*`+`*y*`i` for complex numbers,

  `lsyntax_commonlisp`
  : accept the `#b`, `#o`, `#x` syntaxes for binary, octal, hexadecimal numbers, `#`*base*`R` for rational numbers in a given base, `#c(`*realpart imagpart*`)` for complex numbers,

  `lsyntax_all`
  : accept all of these extensions.

`unsigned int rational_base`
: The base in which rational numbers are read.

`float_format_t float_flags.default_float_format`
: The float format used when reading floats with exponent marker ‘`e`’.

`float_format_t float_flags.default_lfloat_format`
: The float format used when reading floats with exponent marker ‘`l`’.

`cl_boolean float_flags.mantissa_dependent_float_format`
: When this flag is true, floats specified with more digits than corresponding to the exponent marker they contain, but without *_nnn* suffix, will get a precision corresponding to their number of significant digits.

5.3 Output functions

Including `<cln/io.h>` defines a number of simple output functions that write to `std::ostream&`:

`void fprintchar (std::ostream& stream, char c)`
: Prints the character `x` literally on the `stream`.

`void fprint (std::ostream& stream, const char * string)`
: Prints the `string` literally on the `stream`.

`void fprintdecimal (std::ostream& stream, int x)`
`void fprintdecimal (std::ostream& stream, const cl_I& x)`
: Prints the integer `x` in decimal on the `stream`.

`void fprintbinary (std::ostream& stream, const cl_I& x)`
: Prints the integer `x` in binary (base 2, without prefix) on the `stream`.

`void fprintoctal (std::ostream& stream, const cl_I& x)`
: Prints the integer `x` in octal (base 8, without prefix) on the `stream`.

`void fprinthexadecimal (std::ostream& stream, const cl_I& x)`
: Prints the integer `x` in hexadecimal (base 16, without prefix) on the `stream`.

Each of the classes `cl_N`, `cl_R`, `cl_RA`, `cl_I`, `cl_F`, `cl_SF`, `cl_FF`, `cl_DF`, `cl_LF` defines, in `<cln/type_io.h>`, the following output functions:

`void fprint (std::ostream& stream, const type& x)`
`std::ostream& operator<< (std::ostream& stream, const type& x)`

Prints the number `x` on the `stream`. The output may depend on the global printer settings in the variable `default_print_flags`. The `ostream` flags and settings (flags, width and locale) are ignored.

The most flexible output function, defined in `<cln/type_io.h>`, are the following:

```
void print_complex  (std::ostream& stream, const cl_print_flags& flags,
                     const cl_N& z);
void print_real     (std::ostream& stream, const cl_print_flags& flags,
                     const cl_R& z);
void print_float    (std::ostream& stream, const cl_print_flags& flags,
                     const cl_F& z);
void print_rational (std::ostream& stream, const cl_print_flags& flags,
                     const cl_RA& z);
void print_integer  (std::ostream& stream, const cl_print_flags& flags,
                     const cl_I& z);
```

Prints the number `x` on the `stream`. The `flags` are parameters which affect the output.

The structure type `cl_print_flags` contains the following fields:

`unsigned int rational_base`

The base in which rational numbers are printed. Default is `10`.

`cl_boolean rational_readably`

If this flag is true, rational numbers are printed with radix specifiers in Common Lisp syntax (`#nR` or `#b` or `#o` or `#x` prefixes, trailing dot). Default is false.

`cl_boolean float_readably`

If this flag is true, type specific exponent markers have precedence over 'E'. Default is false.

`float_format_t default_float_format`

Floating point numbers of this format will be printed using the 'E' exponent marker. Default is `float_format_ffloat`.

`cl_boolean complex_readably`

If this flag is true, complex numbers will be printed using the Common Lisp syntax `#C(realpart imagpart)`. Default is false.

`cl_string univpoly_varname`

Univariate polynomials with no explicit indeterminate name will be printed using this variable name. Default is `"x"`.

The global variable `default_print_flags` contains the default values, used by the function `fprint`.

6 Rings

CLN has a class of abstract rings.

```
Ring
cl_ring
<cln/ring.h>
```

Rings can be compared for equality:

`bool operator== (const cl_ring&, const cl_ring&)`
`bool operator!= (const cl_ring&, const cl_ring&)`
These compare two rings for equality.

Given a ring `R`, the following members can be used.

`void R->fprint (std::ostream& stream, const cl_ring_element& x)`
`cl_boolean R->equal (const cl_ring_element& x, const cl_ring_element& y)`
`cl_ring_element R->zero ()`
`cl_boolean R->zerop (const cl_ring_element& x)`
`cl_ring_element R->plus (const cl_ring_element& x, const cl_ring_element& y)`
`cl_ring_element R->minus (const cl_ring_element& x, const cl_ring_element& y)`
`cl_ring_element R->uminus (const cl_ring_element& x)`
`cl_ring_element R->one ()`
`cl_ring_element R->canonhom (const cl_I& x)`
`cl_ring_element R->mul (const cl_ring_element& x, const cl_ring_element& y)`
`cl_ring_element R->square (const cl_ring_element& x)`
`cl_ring_element R->expt_pos (const cl_ring_element& x, const cl_I& y)`

The following rings are built-in.

`cl_null_ring cl_0_ring`
The null ring, containing only zero.

`cl_complex_ring cl_C_ring`
The ring of complex numbers. This corresponds to the type `cl_N`.

`cl_real_ring cl_R_ring`
The ring of real numbers. This corresponds to the type `cl_R`.

`cl_rational_ring cl_RA_ring`
The ring of rational numbers. This corresponds to the type `cl_RA`.

`cl_integer_ring cl_I_ring`
The ring of integers. This corresponds to the type `cl_I`.

Type tests can be performed for any of `cl_C_ring`, `cl_R_ring`, `cl_RA_ring`, `cl_I_ring`:

`cl_boolean instanceof (const cl_number& x, const cl_number_ring& R)`
Tests whether the given number is an element of the number ring R.

7 Modular integers

7.1 Modular integer rings

CLN implements modular integers, i.e. integers modulo a fixed integer N. The modulus is explicitly part of every modular integer. CLN doesn't allow you to (accidentally) mix elements of different modular rings, e.g. `(3 mod 4) + (2 mod 5)` will result in a runtime error. (Ideally one would imagine a generic data type `cl_MI(N)`, but C++ doesn't have generic types. So one has to live with runtime checks.)

The class of modular integer rings is

```
             Ring
           cl_ring
        <cln/ring.h>
              |
              |
     Modular integer ring
        cl_modint_ring
     <cln/modinteger.h>
```

and the class of all modular integers (elements of modular integer rings) is

```
       Modular integer
            cl_MI
     <cln/modinteger.h>
```

Modular integer rings are constructed using the function

`cl_modint_ring find_modint_ring (const cl_I& N)`

This function returns the modular ring 'Z/NZ'. It takes care of finding out about special cases of `N`, like powers of two and odd numbers for which Montgomery multiplication will be a win, and precomputes any necessary auxiliary data for computing modulo `N`. There is a cache table of rings, indexed by `N` (or, more precisely, by `abs(N)`). This ensures that the precomputation costs are reduced to a minimum.

Modular integer rings can be compared for equality:

`bool operator== (const cl_modint_ring&, const cl_modint_ring&)`
`bool operator!= (const cl_modint_ring&, const cl_modint_ring&)`

These compare two modular integer rings for equality. Two different calls to `find_modint_ring` with the same argument necessarily return the same ring because it is memoized in the cache table.

7.2 Functions on modular integers

Given a modular integer ring `R`, the following members can be used.

`cl_I R->modulus`

This is the ring's modulus, normalized to be nonnegative: `abs(N)`.

`cl_MI R->zero()`

This returns `0 mod N`.

`cl_MI R->one()`
: This returns `1 mod N`.

`cl_MI R->canonhom (const cl_I& x)`
: This returns `x mod N`.

`cl_I R->retract (const cl_MI& x)`
: This is a partial inverse function to `R->canonhom`. It returns the standard representative (`>=0`, `<N`) of `x`.

`cl_MI R->random(random_state& randomstate)`
`cl_MI R->random()`
: This returns a random integer modulo `N`.

The following operations are defined on modular integers.

`cl_modint_ring x.ring ()`
: Returns the ring to which the modular integer `x` belongs.

`cl_MI operator+ (const cl_MI&, const cl_MI&)`
: Returns the sum of two modular integers. One of the arguments may also be a plain integer.

`cl_MI operator- (const cl_MI&, const cl_MI&)`
: Returns the difference of two modular integers. One of the arguments may also be a plain integer.

`cl_MI operator- (const cl_MI&)`
: Returns the negative of a modular integer.

`cl_MI operator* (const cl_MI&, const cl_MI&)`
: Returns the product of two modular integers. One of the arguments may also be a plain integer.

`cl_MI square (const cl_MI&)`
: Returns the square of a modular integer.

`cl_MI recip (const cl_MI& x)`
: Returns the reciprocal `x^-1` of a modular integer `x`. `x` must be coprime to the modulus, otherwise an error message is issued.

`cl_MI div (const cl_MI& x, const cl_MI& y)`
: Returns the quotient `x*y^-1` of two modular integers `x`, `y`. `y` must be coprime to the modulus, otherwise an error message is issued.

`cl_MI expt_pos (const cl_MI& x, const cl_I& y)`
: `y` must be > 0. Returns `x^y`.

`cl_MI expt (const cl_MI& x, const cl_I& y)`
: Returns `x^y`. If `y` is negative, `x` must be coprime to the modulus, else an error message is issued.

`cl_MI operator<< (const cl_MI& x, const cl_I& y)`
: Returns `x*2^y`.

`cl_MI operator>> (const cl_MI& x, const cl_I& y)`
: Returns `x*2^-y`. When `y` is positive, the modulus must be odd, or an error message is issued.

`bool operator== (const cl_MI&, const cl_MI&)`
`bool operator!= (const cl_MI&, const cl_MI&)`
: Compares two modular integers, belonging to the same modular integer ring, for equality.

`cl_boolean zerop (const cl_MI& x)`
: Returns true if `x` is `0 mod N`.

The following output functions are defined (see also the chapter on input/output).

`void fprint (std::ostream& stream, const cl_MI& x)`
`std::ostream& operator<< (std::ostream& stream, const cl_MI& x)`
: Prints the modular integer `x` on the `stream`. The output may depend on the global printer settings in the variable `default_print_flags`.

8 Symbolic data types

CLN implements two symbolic (non-numeric) data types: strings and symbols.

8.1 Strings

The class

```
String
cl_string
<cln/string.h>
```

implements immutable strings.

Strings are constructed through the following constructors:

`cl_string (const char * s)`
: Returns an immutable copy of the (zero-terminated) C string `s`.

`cl_string (const char * ptr, unsigned long len)`
: Returns an immutable copy of the `len` characters at `ptr[0]`, ..., `ptr[len-1]`. NUL characters are allowed.

The following functions are available on strings:

`operator =`
: Assignment from `cl_string` and `const char *`.

`s.length()`
`strlen(s)`
: Returns the length of the string `s`.

`s[i]`
: Returns the `i`th character of the string `s`. `i` must be in the range `0 <= i < s.length()`.

`bool equal (const cl_string& s1, const cl_string& s2)`
: Compares two strings for equality. One of the arguments may also be a plain `const char *`.

8.2 Symbols

Symbols are uniquified strings: all symbols with the same name are shared. This means that comparison of two symbols is fast (effectively just a pointer comparison), whereas comparison of two strings must in the worst case walk both strings until their end. Symbols are used, for example, as tags for properties, as names of variables in polynomial rings, etc.

Symbols are constructed through the following constructor:

`cl_symbol (const cl_string& s)`
: Looks up or creates a new symbol with a given name.

The following operations are available on symbols:

`cl_string (const cl_symbol& sym)`
: Conversion to `cl_string`: Returns the string which names the symbol `sym`.

`bool equal (const cl_symbol& sym1, const cl_symbol& sym2)`
: Compares two symbols for equality. This is very fast.

9 Univariate polynomials

9.1 Univariate polynomial rings

CLN implements univariate polynomials (polynomials in one variable) over an arbitrary ring. The indeterminate variable may be either unnamed (and will be printed according to `default_print_flags.univpoly_varname`, which defaults to ‘`x`’) or carry a given name. The base ring and the indeterminate are explicitly part of every polynomial. CLN doesn’t allow you to (accidentally) mix elements of different polynomial rings, e.g. `(a^2+1) * (b^3-1)` will result in a runtime error. (Ideally this should return a multivariate polynomial, but they are not yet implemented in CLN.)

The classes of univariate polynomial rings are

```
                          Ring
                        cl_ring
                      <cln/ring.h>
                           |
                           |
                Univariate polynomial ring
                     cl_univpoly_ring
                     <cln/univpoly.h>
                           |
          +----------------+-------------------+
          |                |                   |
Complex polynomial ring    |    Modular integer polynomial ring
cl_univpoly_complex_ring   |        cl_univpoly_modint_ring
<cln/univpoly_complex.h>   |        <cln/univpoly_modint.h>
                           |
          +----------------+
          |                |
  Real polynomial ring     |
  cl_univpoly_real_ring    |
  <cln/univpoly_real.h>    |
                           |
          +----------------+
          |                |
Rational polynomial ring   |
cl_univpoly_rational_ring  |
<cln/univpoly_rational.h>  |
                           |
          +----------------+
          |
Integer polynomial ring
cl_univpoly_integer_ring
<cln/univpoly_integer.h>
```

and the corresponding classes of univariate polynomials are

```
                  Univariate polynomial
                         cl_UP
                     <cln/univpoly.h>
```

```
                                 |
                +----------------+-------------------+
                |                |                   |
        Complex polynomial       |      Modular integer polynomial
             cl_UP_N             |                cl_UP_MI
      <cln/univpoly_complex.h>   |        <cln/univpoly_modint.h>
                                 |
                +----------------+
                |                |
          Real polynomial        |
             cl_UP_R             |
       <cln/univpoly_real.h>     |
                                 |
                +----------------+
                |                |
        Rational polynomial      |
             cl_UP_RA            |
      <cln/univpoly_rational.h>  |
                                 |
                +----------------+
                |
        Integer polynomial
             cl_UP_I
      <cln/univpoly_integer.h>
```

Univariate polynomial rings are constructed using the functions

`cl_univpoly_ring find_univpoly_ring (const cl_ring& R)`
`cl_univpoly_ring find_univpoly_ring (const cl_ring& R, const cl_symbol& varname)`

This function returns the polynomial ring '`R[X]`', unnamed or named. `R` may be an arbitrary ring. This function takes care of finding out about special cases of `R`, such as the rings of complex numbers, real numbers, rational numbers, integers, or modular integer rings. There is a cache table of rings, indexed by `R` and `varname`. This ensures that two calls of this function with the same arguments will return the same polynomial ring.

`cl_univpoly_complex_ring find_univpoly_ring (const cl_complex_ring& R)`
`cl_univpoly_complex_ring find_univpoly_ring (const cl_complex_ring& R, const cl_symbol& varname)`
`cl_univpoly_real_ring find_univpoly_ring (const cl_real_ring& R)`
`cl_univpoly_real_ring find_univpoly_ring (const cl_real_ring& R, const cl_symbol& varname)`
`cl_univpoly_rational_ring find_univpoly_ring (const cl_rational_ring& R)`
`cl_univpoly_rational_ring find_univpoly_ring (const cl_rational_ring& R, const cl_symbol& varname)`
`cl_univpoly_integer_ring find_univpoly_ring (const cl_integer_ring& R)`
`cl_univpoly_integer_ring find_univpoly_ring (const cl_integer_ring& R, const cl_symbol& varname)`
`cl_univpoly_modint_ring find_univpoly_ring (const cl_modint_ring& R)`
`cl_univpoly_modint_ring find_univpoly_ring (const cl_modint_ring& R, const cl_symbol& varname)`

These functions are equivalent to the general `find_univpoly_ring`, only the return type is more specific, according to the base ring's type.

9.2 Functions on univariate polynomials

Given a univariate polynomial ring `R`, the following members can be used.

`cl_ring R->basering()`

This returns the base ring, as passed to '`find_univpoly_ring`'.

`cl_UP R->zero()`

This returns `0 in R`, a polynomial of degree -1.

`cl_UP R->one()`

This returns `1 in R`, a polynomial of degree == 0.

`cl_UP R->canonhom (const cl_I& x)`

This returns `x in R`, a polynomial of degree <= 0.

`cl_UP R->monomial (const cl_ring_element& x, uintL e)`

This returns a sparse polynomial: `x * X^e`, where `X` is the indeterminate.

`cl_UP R->create (sintL degree)`

Creates a new polynomial with a given degree. The zero polynomial has degree `-1`. After creating the polynomial, you should put in the coefficients, using the `set_coeff` member function, and then call the `finalize` member function.

The following are the only destructive operations on univariate polynomials.

`void set_coeff (cl_UP& x, uintL index, const cl_ring_element& y)`

This changes the coefficient of `X^index` in `x` to be `y`. After changing a polynomial and before applying any "normal" operation on it, you should call its `finalize` member function.

`void finalize (cl_UP& x)`
: This function marks the endpoint of destructive modifications of a polynomial. It normalizes the internal representation so that subsequent computations have less overhead. Doing normal computations on unnormalized polynomials may produce wrong results or crash the program.

The following operations are defined on univariate polynomials.

`cl_univpoly_ring x.ring ()`
: Returns the ring to which the univariate polynomial `x` belongs.

`cl_UP operator+ (const cl_UP&, const cl_UP&)`
: Returns the sum of two univariate polynomials.

`cl_UP operator- (const cl_UP&, const cl_UP&)`
: Returns the difference of two univariate polynomials.

`cl_UP operator- (const cl_UP&)`
: Returns the negative of a univariate polynomial.

`cl_UP operator* (const cl_UP&, const cl_UP&)`
: Returns the product of two univariate polynomials. One of the arguments may also be a plain integer or an element of the base ring.

`cl_UP square (const cl_UP&)`
: Returns the square of a univariate polynomial.

`cl_UP expt_pos (const cl_UP& x, const cl_I& y)`
: `y` must be > 0. Returns `x^y`.

`bool operator== (const cl_UP&, const cl_UP&)`
`bool operator!= (const cl_UP&, const cl_UP&)`
: Compares two univariate polynomials, belonging to the same univariate polynomial ring, for equality.

`cl_boolean zerop (const cl_UP& x)`
: Returns true if `x` is `0 in R`.

`sintL degree (const cl_UP& x)`
: Returns the degree of the polynomial. The zero polynomial has degree `-1`.

`sintL ldegree (const cl_UP& x)`
: Returns the low degree of the polynomial. This is the degree of the first non-vanishing polynomial coefficient. The zero polynomial has ldegree `-1`.

`cl_ring_element coeff (const cl_UP& x, uintL index)`
: Returns the coefficient of `X^index` in the polynomial `x`.

`cl_ring_element x (const cl_ring_element& y)`
: Evaluation: If `x` is a polynomial and `y` belongs to the base ring, then '`x(y)`' returns the value of the substitution of `y` into `x`.

`cl_UP deriv (const cl_UP& x)`
: Returns the derivative of the polynomial `x` with respect to the indeterminate `X`.

The following output functions are defined (see also the chapter on input/output).

`void fprint (std::ostream& stream, const cl_UP& x)`
`std::ostream& operator<< (std::ostream& stream, const cl_UP& x)`
: Prints the univariate polynomial `x` on the `stream`. The output may depend on the global printer settings in the variable `default_print_flags`.

9.3 Special polynomials

The following functions return special polynomials.

`cl_UP_I tschebychev (sintL n)`
: Returns the n-th Chebyshev polynomial (n >= 0).

`cl_UP_I hermite (sintL n)`
: Returns the n-th Hermite polynomial (n >= 0).

`cl_UP_RA legendre (sintL n)`
: Returns the n-th Legendre polynomial (n >= 0).

`cl_UP_I laguerre (sintL n)`
: Returns the n-th Laguerre polynomial (n >= 0).

Information how to derive the differential equation satisfied by each of these polynomials from their definition can be found in the `doc/polynomial/` directory.

10 Internals

10.1 Why C++ ?

Using C++ as an implementation language provides

- Efficiency: It compiles to machine code.
- Portability: It runs on all platforms supporting a C++ compiler. Because of the availability of GNU C++, this includes all currently used 32-bit and 64-bit platforms, independently of the quality of the vendor's C++ compiler.
- Type safety: The C++ compilers knows about the number types and complains if, for example, you try to assign a float to an integer variable. However, a drawback is that C++ doesn't know about generic types, hence a restriction like that `operator+ (const cl_MI&, const cl_MI&)` requires that both arguments belong to the same modular ring cannot be expressed as a compile-time information.
- Algebraic syntax: The elementary operations `+`, `-`, `*`, `=`, `==`, ... can be used in infix notation, which is more convenient than Lisp notation '`(+ x y)`' or C notation '`add(x,y,&z)`'.

With these language features, there is no need for two separate languages, one for the implementation of the library and one in which the library's users can program. This means that a prototype implementation of an algorithm can be integrated into the library immediately after it has been tested and debugged. No need to rewrite it in a low-level language after having prototyped in a high-level language.

10.2 Memory efficiency

In order to save memory allocations, CLN implements:

- Object sharing: An operation like `x+0` returns `x` without copying it.
- Garbage collection: A reference counting mechanism makes sure that any number object's storage is freed immediately when the last reference to the object is gone.
- Small integers are represented as immediate values instead of pointers to heap allocated storage. This means that integers `> -2^29`, `< 2^29` don't consume heap memory, unless they were explicitly allocated on the heap.

10.3 Speed efficiency

Speed efficiency is obtained by the combination of the following tricks and algorithms:

- Small integers, being represented as immediate values, don't require memory access, just a couple of instructions for each elementary operation.
- The kernel of CLN has been written in assembly language for some CPUs (`i386`, `m68k`, `sparc`, `mips`, `arm`).
- On all CPUs, CLN may be configured to use the superefficient low-level routines from GNU GMP version 3.
- For large numbers, CLN uses, instead of the standard `O(N^2)` algorithm, the Karatsuba multiplication, which is an $O(N^{1.6})$ algorithm.

- For very large numbers (more than 12000 decimal digits), CLN uses Schönhage-Strassen multiplication, which is an asymptotically optimal multiplication algorithm.
- These fast multiplication algorithms also give improvements in the speed of division and radix conversion.

10.4 Garbage collection

All the number classes are reference count classes: They only contain a pointer to an object in the heap. Upon construction, assignment and destruction of number objects, only the objects' reference count are manipulated.

Memory occupied by number objects are automatically reclaimed as soon as their reference count drops to zero.

For number rings, another strategy is implemented: There is a cache of, for example, the modular integer rings. A modular integer ring is destroyed only if its reference count dropped to zero and the cache is about to be resized. The effect of this strategy is that recently used rings remain cached, whereas undue memory consumption through cached rings is avoided.

11 Using the library

For the following discussion, we will assume that you have installed the CLN source in `$CLN_DIR` and built it in `$CLN_TARGETDIR`. For example, for me it's `CLN_DIR="$HOME/cln"` and `CLN_TARGETDIR="$HOME/cln/linuxelf"`. You might define these as environment variables, or directly substitute the appropriate values.

11.1 Compiler options

Until you have installed CLN in a public place, the following options are needed:

When you compile CLN application code, add the flags

```
-I$CLN_DIR/include -I$CLN_TARGETDIR/include
```

to the C++ compiler's command line (`make` variable CFLAGS or CXXFLAGS). When you link CLN application code to form an executable, add the flags

```
$CLN_TARGETDIR/src/libcln.a
```

to the C/C++ compiler's command line (`make` variable LIBS).

If you did a `make install`, the include files are installed in a public directory (normally `/usr/local/include`), hence you don't need special flags for compiling. The library has been installed to a public directory as well (normally `/usr/local/lib`), hence when linking a CLN application it is sufficient to give the flag `-lcln`.

Since CLN version 1.1, there are two tools to make the creation of software packages that use CLN easier:

- `cln-config` is a shell script that you can use to determine the compiler and linker command line options required to compile and link a program with CLN. Start it with `--help` to learn about its options or consult the manpage that comes with it.
- `AC_PATH_CLN` is for packages configured using GNU automake. The synopsis is:

  `AC_PATH_CLN([`*MIN-VERSION*`, [`*ACTION-IF-FOUND*` [, `*ACTION-IF-NOT-FOUND*`]]])`

  This macro determines the location of CLN using `cln-config`, which is either found in the user's path, or from the environment variable `CLN_CONFIG`. It tests the installed libraries to make sure that their version is not earlier than *MIN-VERSION* (a default version will be used if not specified). If the required version was found, sets the `CLN_CPPFLAGS` and the `CLN_LIBS` variables. This macro is in the file '`cln.m4`' which is installed in '`$datadir/aclocal`'. Note that if automake was installed with a different '`--prefix`' than CLN, you will either have to manually move '`cln.m4`' to automake's '`$datadir/aclocal`', or give aclocal the '`-I`' option when running it. Here is a possible example to be included in your package's '`configure.ac`':

  ```
  AC_PATH_CLN(1.1.0, [
    LIBS="$LIBS $CLN_LIBS"
    CPPFLAGS="$CPPFLAGS $CLN_CPPFLAGS"
  ], AC_MSG_ERROR([No suitable installed version of CLN could be found.]))
  ```

11.2 Compatibility to old CLN versions

As of CLN version 1.1 all non-macro identifiers were hidden in namespace `cln` in order to avoid potential name clashes with other C++ libraries. If you have an old application, you will have to manually port it to the new scheme. The following principles will help during the transition:

- All headers are now in a separate subdirectory. Instead of including `cl_`*something*`.h`, include `cln/`*something*`.h` now.
- All public identifiers (typenames and functions) have lost their `cl_` prefix. Exceptions are all the typenames of number types, (cl_N, cl_I, cl_MI, . . .), rings, symbolic types (cl_string, cl_symbol) and polynomials (cl_UP_*type*). (This is because their names would not be mnemonic enough once the namespace `cln` is imported. Even in a namespace we favor `cl_N` over `N`.)
- All public *functions* that had by a `cl_` in their name still carry that `cl_` if it is intrinsic part of a typename (as in `cl_I_to_int ()`).

When developing other libraries, please keep in mind not to import the namespace `cln` in one of your public header files by saying `using namespace cln;`. This would propagate to other applications and can cause name clashes there.

11.3 Include files

Here is a summary of the include files and their contents.

`<cln/object.h>`
: General definitions, reference counting, garbage collection.

`<cln/number.h>`
: The class cl_number.

`<cln/complex.h>`
: Functions for class cl_N, the complex numbers.

`<cln/real.h>`
: Functions for class cl_R, the real numbers.

`<cln/float.h>`
: Functions for class cl_F, the floats.

`<cln/sfloat.h>`
: Functions for class cl_SF, the short-floats.

`<cln/ffloat.h>`
: Functions for class cl_FF, the single-floats.

`<cln/dfloat.h>`
: Functions for class cl_DF, the double-floats.

`<cln/lfloat.h>`
: Functions for class cl_LF, the long-floats.

`<cln/rational.h>`
: Functions for class cl_RA, the rational numbers.

`<cln/integer.h>`
: Functions for class cl_I, the integers.

`<cln/io.h>`
: Input/Output.

`<cln/complex_io.h>`
: Input/Output for class cl_N, the complex numbers.

`<cln/real_io.h>`
: Input/Output for class cl_R, the real numbers.

`<cln/float_io.h>`
: Input/Output for class cl_F, the floats.

`<cln/sfloat_io.h>`
: Input/Output for class cl_SF, the short-floats.

`<cln/ffloat_io.h>`
: Input/Output for class cl_FF, the single-floats.

`<cln/dfloat_io.h>`
: Input/Output for class cl_DF, the double-floats.

`<cln/lfloat_io.h>`
: Input/Output for class cl_LF, the long-floats.

`<cln/rational_io.h>`
: Input/Output for class cl_RA, the rational numbers.

`<cln/integer_io.h>`
: Input/Output for class cl_I, the integers.

`<cln/input.h>`
: Flags for customizing input operations.

`<cln/output.h>`
: Flags for customizing output operations.

`<cln/malloc.h>`
: `malloc_hook`, `free_hook`.

`<cln/abort.h>`
: `cl_abort`.

`<cln/condition.h>`
: Conditions/exceptions.

`<cln/string.h>`
: Strings.

`<cln/symbol.h>`
: Symbols.

`<cln/proplist.h>`
: Property lists.

`<cln/ring.h>`
: General rings.

`<cln/null_ring.h>`
: The null ring.

`<cln/complex_ring.h>`
: The ring of complex numbers.

`<cln/real_ring.h>`
: The ring of real numbers.

`<cln/rational_ring.h>`
: The ring of rational numbers.

`<cln/integer_ring.h>`
: The ring of integers.

`<cln/numtheory.h>`
: Number threory functions.

`<cln/modinteger.h>`
: Modular integers.

`<cln/V.h>`
: Vectors.

`<cln/GV.h>`
: General vectors.

`<cln/GV_number.h>`
: General vectors over cl_number.

`<cln/GV_complex.h>`
: General vectors over cl_N.

`<cln/GV_real.h>`
: General vectors over cl_R.

`<cln/GV_rational.h>`
: General vectors over cl_RA.

`<cln/GV_integer.h>`
: General vectors over cl_I.

`<cln/GV_modinteger.h>`
: General vectors of modular integers.

`<cln/SV.h>`
: Simple vectors.

`<cln/SV_number.h>`
: Simple vectors over cl_number.

`<cln/SV_complex.h>`
: Simple vectors over cl_N.

`<cln/SV_real.h>`
: Simple vectors over cl_R.

<cln/SV_rational.h>
: Simple vectors over cl_RA.

<cln/SV_integer.h>
: Simple vectors over cl_I.

<cln/SV_ringelt.h>
: Simple vectors of general ring elements.

<cln/univpoly.h>
: Univariate polynomials.

<cln/univpoly_integer.h>
: Univariate polynomials over the integers.

<cln/univpoly_rational.h>
: Univariate polynomials over the rational numbers.

<cln/univpoly_real.h>
: Univariate polynomials over the real numbers.

<cln/univpoly_complex.h>
: Univariate polynomials over the complex numbers.

<cln/univpoly_modint.h>
: Univariate polynomials over modular integer rings.

<cln/timing.h>
: Timing facilities.

<cln/cln.h>
: Includes all of the above.

11.4 An Example

A function which computes the nth Fibonacci number can be written as follows.

```
#include <cln/integer.h>
#include <cln/real.h>
using namespace cln;

// Returns F_n, computed as the nearest integer to
// ((1+sqrt(5))/2)^n/sqrt(5). Assume n>=0.
const cl_I fibonacci (int n)
{
        // Need a precision of ((1+sqrt(5))/2)^-n.
        float_format_t prec = float_format((int)(0.208987641*n+5));
        cl_R sqrt5 = sqrt(cl_float(5,prec));
        cl_R phi = (1+sqrt5)/2;
        return round1( expt(phi,n)/sqrt5 );
}
```

Let's explain what is going on in detail.

The include file <cln/integer.h> is necessary because the type cl_I is used in the function, and the include file <cln/real.h> is needed for the type cl_R and the floating point number functions.

The order of the include files does not matter. In order not to write out `cln::`*foo* in this simple example we can safely import the whole namespace `cln`.

Then comes the function declaration. The argument is an `int`, the result an integer. The return type is defined as '`const cl_I`', not simply '`cl_I`', because that allows the compiler to detect typos like '`fibonacci(n) = 100`'. It would be possible to declare the return type as `const cl_R` (real number) or even `const cl_N` (complex number). We use the most specialized possible return type because functions which call '`fibonacci`' will be able to profit from the compiler's type analysis: Adding two integers is slightly more efficient than adding the same objects declared as complex numbers, because it needs less type dispatch. Also, when linking to CLN as a non-shared library, this minimizes the size of the resulting executable program.

The result will be computed as expt(phi,n)/sqrt(5), rounded to the nearest integer. In order to get a correct result, the absolute error should be less than 1/2, i.e. the relative error should be less than sqrt(5)/(2*expt(phi,n)). To this end, the first line computes a floating point precision for sqrt(5) and phi.

Then sqrt(5) is computed by first converting the integer 5 to a floating point number and than taking the square root. The converse, first taking the square root of 5, and then converting to the desired precision, would not work in CLN: The square root would be computed to a default precision (normally single-float precision), and the following conversion could not help about the lacking accuracy. This is because CLN is not a symbolic computer algebra system and does not represent sqrt(5) in a non-numeric way.

The type `cl_R` for sqrt5 and, in the following line, phi is the only possible choice. You cannot write `cl_F` because the C++ compiler can only infer that `cl_float(5,prec)` is a real number. You cannot write `cl_N` because a '`round1`' does not exist for general complex numbers.

When the function returns, all the local variables in the function are automatically reclaimed (garbage collected). Only the result survives and gets passed to the caller.

The file `fibonacci.cc` in the subdirectory `examples` contains this implementation together with an even faster algorithm.

11.5 Debugging support

When debugging a CLN application with GNU `gdb`, two facilities are available from the library:

- The library does type checks, range checks, consistency checks at many places. When one of these fails, the function `cl_abort()` is called. Its default implementation is to perform an `exit(1)`, so you won't have a core dump. But for debugging, it is best to set a breakpoint at this function:

  ```
  (gdb) break cl_abort
  ```

  When this breakpoint is hit, look at the stack's backtrace:

  ```
  (gdb) where
  ```

- The debugger's normal `print` command doesn't know about CLN's types and therefore prints mostly useless hexadecimal addresses. CLN offers a function `cl_print`, callable from the debugger, for printing number objects. In order to get this function, you have to define the macro '`CL_DEBUG`' and then include all the header files for which you want `cl_print` debugging support. For example:

  ```
  #define CL_DEBUG
  ```

```
#include <cln/string.h>
```

Now, if you have in your program a variable `cl_string s`, and inspect it under `gdb`, the output may look like this:

```
(gdb) print s
$7 = {<cl_gcpointer> = { = {pointer = 0x8055b60, heappointer = 0x8055b60,
  word = 134568800}}, }
(gdb) call cl_print(s)
(cl_string) ""
$8 = 134568800
```

Note that the output of `cl_print` goes to the program's error output, not to gdb's standard output.

Note, however, that the above facility does not work with all CLN types, only with number objects and similar. Therefore CLN offers a member function `debug_print()` on all CLN types. The same macro '`CL_DEBUG`' is needed for this member function to be implemented. Under `gdb`, you call it like this:

```
(gdb) print s
$7 = {<cl_gcpointer> = { = {pointer = 0x8055b60, heappointer = 0x8055b60,
  word = 134568800}}, }
(gdb) call s.debug_print()
(cl_string) ""
(gdb) define cprint
>call ($1).debug_print()
>end
(gdb) cprint s
(cl_string) ""
```

Unfortunately, this feature does not seem to work under all circumstances.

12 Customizing

12.1 Error handling

When a fatal error occurs, an error message is output to the standard error output stream, and the function `cl_abort` is called. The default version of this function (provided in the library) terminates the application. To catch such a fatal error, you need to define the function `cl_abort` yourself, with the prototype

```
#include <cln/abort.h>
void cl_abort (void);
```

This function must not return control to its caller.

12.2 Floating-point underflow

Floating point underflow denotes the situation when a floating-point number is to be created which is so close to `0` that its exponent is too low to be represented internally. By default, this causes a fatal error. If you set the global variable

```
cl_boolean cl_inhibit_floating_point_underflow
```

to `cl_true`, the error will be inhibited, and a floating-point zero will be generated instead. The default value of `cl_inhibit_floating_point_underflow` is `cl_false`.

12.3 Customizing I/O

The output of the function `fprint` may be customized by changing the value of the global variable `default_print_flags`.

12.4 Customizing the memory allocator

Every memory allocation of CLN is done through the function pointer `malloc_hook`. Freeing of this memory is done through the function pointer `free_hook`. The default versions of these functions, provided in the library, call `malloc` and `free` and check the `malloc` result against `NULL`. If you want to provide another memory allocator, you need to define the variables `malloc_hook` and `free_hook` yourself, like this:

```
#include <cln/malloc.h>
namespace cln {
        void* (*malloc_hook) (size_t size) = ...;
        void (*free_hook) (void* ptr)      = ...;
}
```

The `cl_malloc_hook` function must not return a `NULL` pointer.

It is not possible to change the memory allocator at runtime, because it is already called at program startup by the constructors of some global variables.

Index

A

abs () ... 12
abstract class ... 6
AC_PATH_CLN ... 45
acos () ... 18
acosh () ... 19
advocacy ... 43
Archimedes' constant ... 19
As()() ... 9
ash () ... 22
asin ... 18
asin () ... 18
asinh () ... 19
atan ... 18
atan () ... 18
atanh () ... 19

B

basering () ... 40
binomial () ... 23
boole () ... 21
boole_1 ... 21
boole_2 ... 21
boole_and ... 21
boole_andc1 ... 21
boole_andc2 ... 21
boole_c1 ... 21
boole_c2 ... 21
boole_clr ... 21
boole_eqv ... 21
boole_nand ... 21
boole_nor ... 21
boole_orc1 ... 21
boole_orc2 ... 21
boole_set ... 21
boole_xor ... 21

C

canonhom () ... 33, 35, 40
cast ... 9
Catalan's constant ... 20
catalanconst () ... 20
ceiling1 () ... 14
ceiling2 () ... 15
Chebyshev polynomial ... 42
cis () ... 18
cl_abort () ... 52
cl_byte ... 21
CL_DEBUG ... 50
cl_DF ... 7
cl_DF_fdiv_t ... 15
cl_F ... 6, 7
cl_F_fdiv_t ... 15
cl_FF ... 7
cl_FF_fdiv_t ... 15
cl_float () ... 25
cl_I_to_int () ... 9
cl_I_to_long () ... 9
cl_I_to_uint () ... 9
cl_I_to_ulong () ... 9
cl_idecoded_float ... 24
cl_LF ... 7
cl_LF_fdiv_t ... 15
cl_modint_ring ... 34
cl_N ... 6
cl_number ... 6
cl_R ... 6
cl_R_fdiv_t ... 15
cl_RA ... 6
cl_SF ... 7
cl_SF_fdiv_t ... 15
cl_string ... 37
cl_symbol ... 37
cln-config ... 45
coeff () ... 41
compare () ... 13
comparison ... 13
compatibility ... 46
compiler options ... 45
complex () ... 12
complex number ... 6, 8
conjugate () ... 12
conversion ... 8, 25
cos () ... 18
cos_sin () ... 18
cos_sin_t ... 18
cosh () ... 19
cosh_sinh () ... 19
cosh_sinh_t ... 19
create () ... 40
customizing ... 52

D

debug_print () ... 51
debugging ... 50
decode_float () ... 24
decoded_dfloat ... 24
decoded_ffloat ... 24
decoded_float ... 24
decoded_lfloat ... 24
decoded_sfloat ... 24

default_float_format 25
default_print_flags 52
default_random_state 26
degree () 41
denominator () 12
deposit_field () 22
deriv () 41
div () 35
double_approx () 9
doublefactorial () 23
dpb () 22

E

equal () 33, 37
equal_hashcode () 13
Euler's constant 20
eulerconst () 20
evenp () 22
exact number 6
exp () 17
exp1 () 17
expt () 11, 17, 35
expt_pos () 11, 33, 35, 41
exquo () 11

F

factorial () 23
fceiling () 15
fceiling2 () 15
ffloor () 15
ffloor2 () 15
Fibonacci number 49
finalize () 41
find_modint_ring () 34
find_univpoly_ring () 40
float_approx () 9
float_digits () 24
float_epsilon () 26
float_exponent () 24
float_format () 25
float_format_t 25
float_negative_epsilon () 26
float_precision () 24
float_radix () 24
float_sign () 24, 25
floating-point number 7
floor1 () 14
floor2 () 15
fprint () 33, 36, 42
free_hook () 52
fround () 15
fround2 () 15
ftruncate () 15
ftruncate2 () 15

G

garbage collection 43, 44
gcd () 23
GMP 1, 5

H

header files 46
hermite () 42
Hermite polynomial 42

I

imagpart () 12
immediate numbers 7, 43
include files 46
Input/Output 29
installation 5
instanceof () 33
integer 6
integer_decode_float () 25
integer_length () 22
isprobprime() 23
isqrt () 16

J

jacobi() 23

L

laguerre () 42
Laguerre polynomial 42
lcm () 23
ldb () 22
ldb_test () 22
least_negative_float () 26
least_positive_float () 25
Legende polynomial 42
legendre () 42
length () 37
ln () 17
log () 17
logand () 21
logandc1 () 21
logandc2 () 21

logbitp () 21
logcount () 21
logeqv () 21
logior () 21
lognand () 21
lognor () 21
lognot () 20
logorc1 () 21
logorc2 () 21
logp () 23
logtest () 21
logxor () 21

M

make 3
malloc_hook () 52
mask_field () 22
max () 13
min () 13
minus () 33
minus1 () 11
minusp () 13
mod () 16
modifying operators 27
modular integer 34
modulus 34
monomial () 40
Montgomery multiplication 34
most_negative_float () 25
most_positive_float () 25
mul () 33

N

namespace 2, 46
nextprobprime() 23
numerator () 12

O

oddp () 22
one () 33, 35, 40
operator != () 13, 34, 36, 41
operator & () 21
operator &= () 27
operator () () 41
operator * () 11, 35, 41
operator *= () 27
operator - () 11, 35, 41
operator -- () 27
operator -= () 27
operator / () 11
operator /= () 27
operator == () 13, 34, 36, 41
operator [] () 37
operator | () 21
operator |= () 27
operator ~ () 20
operator + () 11, 35, 41
operator += () 27
operator ++ () 27
operator > () 13
operator >= () 13
operator >> () 22, 36
operator >>= () 27
operator ^ () 21
operator ^= () 27
operator < () 13
operator <= () 13
operator << () 22, 35, 36, 42
operator <<= () 27
ord2 () 22

P

phase () 17
pi 19
pi () 19
plus () 33
plus1 () 11
plusp () 13
polynomial 38
portability 43
power2p () 23
prime 23
printing 29

R

random () 35
random_F () 27
random_I () 27
random_R () 27
random_state 26
random32 () 26
rational () 26
rational number 6
rationalize () 26
reading 29
real number 6
realpart () 12
recip () 11, 35
reference counting 43

`rem ()` 16
representation 29
`retract ()` 35
Riemann's zeta 20
ring 34
`ring ()` 35, 41
`rootp ()` 16
`round1 ()` 14
`round2 ()` 15
rounding 13
rounding error 7
Rubik's cube 8

S

`scale_float ()` 24
Schönhage-Strassen multiplication 1, 44
`sed` 3
`set_coeff ()` 40
`signum ()` 12
`sin ()` 17
`sinh ()` 19
`sqrt ()` 16
`sqrtp ()` 16
`square ()` 11, 33, 35, 41
string 37
`strlen ()` 37
symbol 37
symbolic type 37

T

`tan ()` 18
`tanh ()` 19
`The()()` 9
transcendental functions 17
`truncate1 ()` 14
`truncate2 ()` 15
`tschebychev ()` 42

U

`uminus ()` 33
underflow 52
univariate polynomial 38

W

`WANT_OBFUSCATING_OPERATORS` 27

X

`xgcd ()` 23

Z

`zero ()` 33, 34, 40
`zerop ()` 13, 33, 36, 41
`zeta ()` 20