Doc No: N2020=06-0090
Date: 2006-6-22
Reply to: M.J. Kronenburg
M.Kronenburg@inter.nl.net

Proposal for an Infinite Precision Integer for Library Technical Report 2, Revision 1

Contents

Chapter 1

Introduction

1.1 Motivation and Scope [integer.scope]

The infinite precision integer is an integer that is not limited in range by the computer word length. The computer word length of 32-bit processors can represent integers up to about 9 decimals, and the computer word length of 64-bit processors can represent integers up to about 19 decimals. The infinite precision integer can represent integers with any number of decimals, as long as its binary representation fits into memory. With the current memory sizes the range of the infinite precision integer is practically unlimited.
Infinite precision means that the precision is set by the program to any value needed to represent the integer, and that it is not set by the user as in the case of arbitrary precision.
The main user group today is the user group in the field of cryptography. Cryptography is needed for all kinds of electronic security systems such as public key encryption and digital signatures [4,8].
Another user group is the group of mathematicians using number theory and computer algebra, a field that is much broader than cryptography.
An existing implementation that is already part of the Gnu C++ compiler library is the Gnu Multiple Precision Arithmetic Library, see `http://www.swox.com/gmp`.

1.2 Impact on the Standard [integer.impact]

There is no impact of change on the standard, as the infinite precision integer class is fully self-contained with its own memory management. For input/output, the standard istream and ostream classes and string class are used, and internally many other standard library elements are used.

1.3 General Requirements [integer.requirements]

The infinite precision integer functionality is provided by the C++ class `integer`.
The general main requirements of the class `integer` are:

1. An `integer` object can represent any integer value, positive, negative or zero, which is not limited by the range of the computer word length.

2. The functionality of the `integer` should at least have the functionality that is available for the base type `int`. This functionality consists of the addition, subtraction, multiplication, division and remainder, exponentiation, absolute value, negation, square root, the boolean operators, the shift operators, the bitwise operators and the stream inserter and extractor. The class `integer` strictly mimics the base type `int`.

3. The modulo, modular exponentiation, modular inverse, the greatest common divisor and least common multiple, and the extended greatest common divisor are provided. These functions are used in cryptography, number theory and computer algebra. With the basic functionality described above, implementation of these functions is relatively easy [2,3,4,8].

4. The user must have bit access to the binary representation. This requirement implies that the sign and the non-negative binary representation of the absolute value are stored separately. This way the bit access function (`get_sub`, see 2.6.7) is uniquely defined.

5. For interfacing with the arithmetic base types, for base type to `integer` conversion, constructors are provided. For `integer` to base type conversion, non-member conversion functions are provided.

6. As an unsigned integer is an integer, and a modular integer is an integer, the derived classes `unsigned_integer` (1.6.1) and the `modular_integer` with a static modulus (1.6.2) are provided, and users can derive others. In these derived classes, certain member functions and operators are overridden (possibly in terms of the base class ones), that is the class `integer` is a run-time polymorphic class [5]. This design ensures that any expression with mixed compile-time and run-time types derived from `integer` is possible (1.6).

7. For defining user-provided memory management functions, an abstract class `integer_allocator` is provided, which can be activated and deactivated at runtime. A class derived from class `integer` can then be used with a static allocator derived from `integer_allocator` (1.5) which is activated and deactivated for each derived constructor, destructor and member function and operator, providing run-time polymorphism [5] between different allocators. In this document the `allocated_integer` (1.6.3) is defined and provided.

8. The performance of the arithmetic operations of the class `integer` should be optimized and comparable with GMP, see `http://www.swox.com/gmp`.

1.4 Design Decisions **[integer.design]**

The infinite precision integer is represented as a C++ class with the name `integer`. An object of class `integer` holds a reference to a contiguous memory block holding the binary representation of the integer, the length of that memory block, and a sign. The internal representation is binary

so that the arithmetic operations can be performed efficiently. When the value of the object is zero, the memory reference, the length and the sign are zero.
In the internal representation, the sign and the non-negative binary representation of the absolute value are stored separately, so that the bit access function `get_sub()` (see 2.6.7) is uniquely defined.
The interface of the class `integer` provides member and non-member functions and operators so that all the basic arithmetic integer operations are available. The lengths of the binary representations are automatically adjusted to hold the result of the arithmetic operations as if the arguments of the arithmetic operations were padded with zero-bits to infinity.
Because classes may be derived from class `integer`, the member functions and operators that can be overridden, including the destructor, are virtual [class.virtual]. This makes the class `integer` a run-time polymorphic class [5]. Non-virtual member or non-member functions or operators must not be redefined in derived classes. The class `integer` is not made abstract because it is clearly a concrete class [5,6,7].
When using `integer` arithmetic expressions with arithmetic base types, like `x += 3`, implicit conversion of the base type through the appropriate `integer` converting constructor (2.3) takes place [class.conv.ctor]. An implementation may add overloads of the member functions and operators for the arithmetic base types [over.match.best] if necessary.
Conversion operators from `integer` to the arithmetic base types are not provided, as combination of conversion operators and converting constructors leads to ambiguities in expressions [5]. Instead, for conversion from `integer` to the arithmetic base types, conversion non-member functions are provided (2.15).
A base type `int` can be implicitly converted to a `bool` [conv.bool], yielding true if not zero and false if zero. For the `integer` implicit conversion to `bool` is provided by the *`unspecified-bool-type`*`()` conversion operator (2.4.2). This way `integer` objects can be used in boolean contexts, such as `if( x )`.
In the boolean comparison `x >= 0` an `integer` with value zero is constructed, and the `integer` boolean operator is called. Such boolean comparisons against zero can be optimized with the `integer` member function `get_sign()` (2.6.3), for example in this case with the boolean comparison `x.get_sign() >= 0`.
The user may provide infinite precision integers in decimal, hexadecimal or octal notation, in input streams, which are converted to binary by the `integer` instream operator. When an `integer` object is streamed to output, binary to decimal, hexadecimal or octal conversion takes place by the `integer` outstream operator. Whether conversion takes place to or from decimal, hexadecimal or octal, is determined by the `basefield` flag of the stream, which can be set by the user with the `dec`, `hex` and `oct` stream manipulators (2.16).
The basic stringstream and filestream classes derive from the basic stream classes [lib.string.streams][lib.file.streams], and therefore streams from and to stringstreams and filestreams are also available.
For conversion to or from `std::string`, constructors and conversion functions are provided

(2.3.14, 2.15.11). For conversion from a C-style string, constructors are provided (2.3.12). For example a constant `integer` may be declared as follows:

```
const integer x( "-12345678901234567890" );
```

For conversion to and from wide strings, wide stringstreams can be used, as mentioned above.

1.5 The allocator class `integer_allocator` [integer.allocator]

An abstract class `integer_allocator` is provided, which provides an interface for a user-defined class derived from class `integer_allocator` with its own memory management functions. The existing standard library allocator [lib.default.allocator] is not used, because it does not provide a member function `reallocate()` [lib.allocator.members], and because it must be possible to switch the allocator at runtime. The allocated elements of the `integer_allocator` are always bytes, so a template type parameter is not needed for this class.

An integer template class with an `Allocator` type parameter cannot be used, because templates provide only compile-time polymorphism [5], so that expressions with different template types are not possible. Instead, a class derived from class `integer` should be used with a static allocator, providing run-time polymorphism [5].

An allocator class derived from `integer_allocator` can be activated and deactivated at runtime, because when modifying an object, intermediate results must also be allocated, reallocated and deallocated with the same allocator, until the final result is assigned.

In the `integer` derived constructors, member functions and operators, and destructor, this static allocator, which is derived from class `integer_allocator`, is activated, then integer member functions and operators are used, and then it is deactivated. Here it must be used that an `integer` with value zero does not have memory allocated.

The `integer` member function `swap()` should only swap the pointers to the allocated memory if the allocators of those pointers are identical, otherwise the allocated data should be swapped. Therefore the `get_allocator()` member function is added to the `integer` class. Classes derived from class `integer` with a static allocator must therefore also override and implement `get_allocator()` and `swap()`.

As classes derived from class `integer_allocator` are thus only used as static class data members, and not as non-static class data members, in the derived constructor, memory management initialization can be done, such as creating a heap handle, and in its derived destructor, memory management cleanup can be done. This requires that static class data members, such as the static allocator, are constructed before a class object can be constructed, and are deleted after each class object is deleted. This seems to be a general requirement for classes to function properly.

1.6 Classes derived from class `integer` [integer.derive]

For defining an `integer` with some specific properties, classes may be derived from base class `integer`, overriding some virtual member functions or operators (possibly in terms of the base class ones), but leaving the other member functions and operators and the non-member functions and operators unchanged. This way expressions with variables of mixed compile-time and run-time types are possible. Examples are an `unsigned_integer`, or a `modular_integer` with a static modulus, or an integer with a specific allocator (see below). The `integer` member functions and operators that can be overridden are virtual, and in their derived implementation, the base class versions can be used with the scope resolution operator [class.derived], in this case with `integer::`. The `integer` member functions and operators that are non-virtual must not be redefined in a derived class.
The `integer` virtual member functions and operators that are overridden in a derived class possibly have as argument a reference to class `integer`. This means that their implementations don't have access to the arguments derived data members, unless a `dynamic_cast` is used [5,7]. Furthermore, the operators returning an `integer` by value cannot return derived data members. Therefore classes derived from class `integer` cannot have derived non-static data members. In other words: classes derived from class `integer` can only override some virtual member functions and operators of class `integer`, possibly using only static data members. This way also pointer arithmetic in arrays remains valid for the derived class [7].
A class derived from class `integer` must thus fulfill the following requirements:

1 It cannot have derived non-static data members

2 It must have constructors, calling the corresponding `integer` constructor (2.3), and converting the object to the derived class.

3 It must override the virtual copy constructor `clone()` (2.5.2) returning a pointer of the derived class and calling the derived copy constuctor

4 It must override `operator=( const integer & )` (2.5.3), converting a (possibly temporary) object to the derived class.

5 It must override virtual member functions and/or operators when they have another meaning for the derived class, possibly having as argument a reference to class `integer`, and preferably calling the `integer::` functions and/or operators in their implementations

6 It must not redefine non-virtual member functions

7 It must not redefine non-member functions or operators

8 When it uses a static allocator (see below), the virtual member functions `get_allocator()` and `swap()` and all other virtual member functions and operators must be overridden and implemented.

The conversion to the derived class in the derived constructors and derived functions and operators, in the case of the unsigned integer means check if the object is non-negative, if not throw an exception, and in the case of the modular integer means check if the object is between zero and the static modulus, if not reduce the object modulo the static modulus. In this context this operation is called normalization, and is provided by the virtual method `normalize()` (2.5.5).
In expressions with operators that return `integer` by value, a derived object is copy constructed to a base class `integer` object, and the corresponding `integer` function or operator is called. The `integer` result is returned by value, and as mentioned above the derived assignment operator converts the result back to the derived class. This way any expression with mixed compile-time and run-time types derived from `integer` is possible, and two expressions which are mathematically equivalent always yield the same result.
When it is required that all objects in an expression are derived objects only, no operators returning `integer` by value must be used.
The class `integer` and its derived classes can also be used as data member of other classes, or as data member of templates.

1.6.1 The class `unsigned_integer` [integer.unsigned]

A class derived from class integer that is specified in this document is the class `unsigned_integer`.
An `unsigned_integer` is an infinite precision `integer` that can only be positive or zero.
The class `unsigned_integer` is identical to the class `integer`, except that all derived constructors and all the overridden virtual member functions and operators first call the corresponding `integer` constructor or member function or operator. When the result is negative, a run-time error in the form of an exception is thrown, which is provided by the derived member function `normalize()` (2.21.5).
The following program:

```
unsigned int x = -2;
cout << x << endl;
```

on most systems give:

```
4294967294
```

which most users will not understand. The following program:

```
unsigned_integer x = -2;
cout << x << endl;
```

will throw an exception in the first line.
Subtraction of two `unsigned_integers` that give a negative result throws an exception. As negation is subtraction from zero, negation of non-zero `unsigned_integers` throws an exception.
The simple rule is that when a variable of run-time type `unsigned_integer` actually becomes negative, an exception is thrown.
Temporary results in expressions however may be negative. Only when the result of an expression is assigned to a variable of run-time type `unsigned_integer`, the is-negative check is made. So

the following expressions with `unsigned_integer` variables `x`, `a` and `b`:

```
x = -a + b;
```
and
```
x = b - a;
```
will both not throw an exception when `a==3` and `b==4` (result is `1`), but will both throw an exception when `a==4` and `b==3` (result is `-1`). This is because (as mentioned above) the `integer` unary `operator-` and binary `operator+` and `operator-` copy construct an argument to `integer`, call the corresponding `integer` member function or operator, and return `integer` by value. In the assignment, the derived `operator=` is called, which converts the temporary `integer` back to the derived run-time type `unsigned_integer`, where the is-negative check is made. This way mathematically equivalent expressions do or do not throw an exception with equivalent variable values. But when the expression is splitted up into two steps:

```
x = -a;
x += b;
```
or
```
x = b;
x -= a;
```
then in the cases mentioned above, in the first program always an exception is thrown, as variable `x` always becomes negative in the first step, but in the second program, only in the second case an exception is thrown, as variable `x` only in the second case becomes negative in the second step.

The equivalent expressions:

```
x.negate();
```
and
```
x = -x;
```
both throw an exception when `x` is non-zero, but in the second case a temporary `integer` with the negative value is made, and during assignment an exception is thrown.

As mentioned, the simple rule is that when a variable of run-time type `unsigned_integer` actually becomes negative, an exception is thrown.

The base type `unsigned int` is actually a modular integer with modulus 2^n. Therefore users that require an unsigned integer that mimics the base type `unsigned int` and its negation [expr.unary.op], should use the class `modular_integer` (see below).

1.6.2 The class `modular_integer` [integer.modular]

A class derived from class integer that is specified in this document is the class `modular_integer`. A `modular_integer` is an `integer` that can only have values between zero and a static modulus. The class `modular_integer` is identical to the class `integer`, except that all derived constructors and all the overridden virtual member functions and operators first call the corresponding `integer` constructor or member function or operator. When the result is not between zero and

the static modulus, the result is reduced modulo the static modulus using the `mod` function (2.7.6), which is provided by the derived member function `normalize()` (2.23.5).
The static modulus of the class `modular_integer` can be set with the static function `set_modulus()`, and must be set before objects of this class are constructed. When the static modulus is not set, its value is zero, and the class `modular_integer` behaves as the class `integer`. The sign of x `mod` y is the sign of y or zero. Therefore variables of run-time type `modular_integer` with a positive static modulus, are always positive or zero, and those with a negative static modulus, are always negative or zero.
Mathematically equivalent expressions always yield identical results, although the temporary `integer` objects may have different values. The expression:

```
x = a * b * c;
```

is equivalent to the following:

```
x = a;
x *= b;
x *= c;
```

The resulting value of `x` is identical in both cases, but in the first case, only the end result is reduced modulo the static modulus, while in the second case, the temporary result is also reduced modulo the static modulus. Which of the two methods is faster, depends on the implementation and on the value of the static modulus.
The base type `unsigned int` is in fact an integer with a modulus 2^n, where n is the number of bits, so that for positive x, $-x$ becomes $2^n - x$ [expr.unary.op]. Users that require an unsigned integer that mimics the base type `unsigned int` and its negation [expr.unary.op], should use the class `modular_integer` and set its static modulus to the value 2^n.
When modular integers with different moduli are required, then the user may use a template class with a `modular_integer` data member and the modulus supplied in a template non-type parameter.

1.6.3 The class `allocated_integer` [integer.allocated]

A class derived from class integer that is specified in this document is the class `allocated_integer`.
The class `allocated_integer` is identical to the class `integer`, except that all derived constructors, destructor and all the overridden virtual member functions and operators first activate an allocator, then call the corresponding `integer` constructor or member function or operator, and then deactivate the allocator. This means that all objects of class `allocated_integer` only have memory allocated, reallocated and freed with this allocator. The allocator is set with the static function `set_allocator()`, which must be called before objects of this class are constructed. This function takes a pointer parameter, which must be supplied with a pointer to an object derived from `integer_allocator` created by new, like in:

```
allocated_integer::set_allocator( new my_allocator() );
```

This object is deleted by the `allocated_integer` class as if it were a static variable.
When allocated integers with different allocators are required, then the user may use a template class with an `allocated_integer` data member and the allocator supplied in a template type parameter.
For completeness, also the derived classes `allocated_unsigned_integer` and `allocated_modular_integer` are provided.

1.7 Required Complexities [integer.complexities]

For all functions the required time complexities are provided. For specifying the time complexities, the following symbols are used for the run time:

Table 1.1: Specific time complexities and their meaning

complexity	**meaning**
N	number of decimals or bits
$\mathrm{M}(N)$	multiplication of two infinite precision integers
$\mathrm{D}(N)$	division and/or remainder of two infinite precision integers
$\mathrm{G}(N)$	greatest common divisor of two infinite precision integers

On most systems $\mathrm{M}(N)<\mathrm{D}(N)<\mathrm{G}(N)$. These basic time complexities depend on the choice of algorithm [1,2,3,4,8], but for large N in general $\mathrm{M}(N)$ and $\mathrm{D}(N)$ should be subquadratic, while $\mathrm{G}(N)$ may be quadratic.
The time complexity of division with Newton iteration is $\mathrm{D}(N) = \mathrm{O}(\mathrm{M}(N))$ [2,9]. The time complexity of square root with Newton iteration is also $\mathrm{O}(\mathrm{M}(N))$ [9]. The time complexity of radix conversion from decimal to binary is $\mathrm{O}(\mathrm{M}(N)\log(N))$ and from binary to decimal $\mathrm{O}(\mathrm{D}(N)\log(N))$ [2].
The time complexities of the member functions `is_zero()`, `get_sign()`, `negate()` and `abs()` are only $\mathrm{O}(1)$ when the sign and the absolute value of the `integer` are stored separately.
The time complexities and algorithms of the other functions can be found in literature [1,2,3,4,8].

1.8 Performance Optimization [integer.performance]

For optimization of the performance of applications using the `integer` class, the `integer` operations must be optimized in performance. For performance optimization, the following considerations may be helpful for the different groups of operations [1,2,3,8]:

1. greatest common divisor
   For the greatest common divisor the binary euclidean algorithm is used, which is order N^2.

The least common multiple is a simple function of the greatest common divisor. For the extended greatest common divisor, the non-binary extended euclidean algorithm is used, which is also order N^2.

2. exponentiation
For exponentiation and modular exponentiation, the binary algorithm for exponentiation is used. The performance of exponentiation also depends on the performance of multiplication.

3. multiplication
The multiplication is usually split up in three ranges: small sizes, medium sizes and large sizes. For small sizes, the basecase multiplication algorithm is used, which is order N^2. For medium sizes, the Karatsuba multiplication algorithm is used, which is order $N^{1.585}$. For large sizes, the Number Theoretic Transform, the Fast Fourier Transform or the n-way Toom-Cook algorithm is used. Each of these algorithms have certain advantages and disadvantages. These three algorithms make up the multiplication performance cost $\mathrm{M}(N)$.
Using assembler can speed up multiplications by about a factor 3. Therefore, in implementations, it should not be difficult to replace basic operations with assembler versions.

4. division and remainder
For small sizes, basecase division is used, which is order N^2. For large sizes, Newton iteration is used, which is order $\mathrm{M}(N)$.

5. square root
For the square root, Newton iteration is used, which is order $\mathrm{M}(N)$.

6. general optimizations
For all operations, including addition and subtraction, a check is made if one of the operands is zero, or if the operands are identical, in which cases the operation may be simplified. This implements optimizations like translating `a+=a` into `a<<=1`.

7. operations on unit size
Because of the checks for the optimizations above, and the checks on for example carries, the operations for arguments of size 1 (that is the size of base type `int`) are much slower (about a factor 10 to 100) than the corresponding base type `int` operations. Because the class `integer` is typically used for values much larger than size 1, other factors (see above) determine the performance of applications of class `integer`. Therefore this should be accepted. For problems that certainly only involve integers always smaller than size 1, the base type `int` should be used, that is: not all base type `int` variables should be replaced with `integer` variables. The same holds for base types `long` and `long long`, if available.

8. virtual functions and operators
The cost of making functions and operators virtual, that is implicitly using virtual table lookups [7], is in all cases negligible.

Chapter 2

Proposed Text for Library Technical Report 2

2.1 General **[integer.general]**

1. This clause describes components that C++ programs may use to perform arithmetic calculations on integers of any precision, limited only by memory availability.

2. The infinite precision integer has a header file that is included with:
`#include <integer>`
which defines the class `integer` and its constructors, destructor, member and non-member functions and operators, its allocator class and its derived classes, which all reside in namespace `std::tr2`.

3. An implementation may add overloads of the member functions and operators for the arithmetic base types [over.match.best] if necessary.

4. The member and non-member functions and operators of these classes throw an appropriate exception, which is derived from `std::exception`, if they cannot achieve their specified effects, postconditions, or returns clauses.

5. The required complexities are provided where N is the number of decimals or bits, and where $\mathrm{M}(N)$ is the multiplication complexity, $\mathrm{D}(N)$ is the division complexity and $\mathrm{G}(N)$ is the greatest common divisor complexity (1.7).

2.2 Synopsis **[integer.synopsis]**

```
namespace std::tr2 {

class integer_allocator;

class integer {
public:
// types:
  typedef std::size_t size_type;
  typedef unsigned int radix_type;
// 2.3 constructors and destructor:
  integer();
  integer( int );
  integer( unsigned int );
  integer( long );
  integer( unsigned long );
  integer( long long );
  integer( unsigned long long );
  explicit integer( float );
  explicit integer( double );
  explicit integer( long double );
  explicit integer( const char * );
  explicit integer( const char *, radix_type );
  explicit integer( const std::string & );
  explicit integer( const std::string &, radix_type );
  integer( const integer & );
  virtual ~integer();
// 2.4  conversion member operators:
  operator unspecified-bool-type() const;
// 2.5 copying and assignment member functions:
  virtual integer * clone() const;
  virtual integer & operator=( const integer & );
  virtual integer_allocator * get_allocator() const;
  virtual void normalize();
  virtual void swap( integer & );
// 2.6 arithmetic member functions:
  bool is_zero() const;
  int get_sign() const;
  bool is_odd() const;
  size_type highest_bit() const;
  size_type lowest_bit() const;
  const integer get_sub( size_type, size_type ) const;
  virtual integer & negate();
  virtual integer & abs();
// 2.8 arithmetic member operators:
  virtual integer & operator++();
  virtual integer & operator--();
  virtual integer & operator+=( const integer & );
  virtual integer & operator-=( const integer & );
  virtual integer & operator*=( const integer & );
  virtual integer & operator/=( const integer & );
  virtual integer & operator%=( const integer & );
// 2.11 shift member operators:
  virtual integer & operator<<=( size_type );
  virtual integer & operator>>=( size_type );
// 2.13 bitwise member operators:
  virtual integer & operator|=( const integer & );
  virtual integer & operator&=( const integer & );
  virtual integer & operator^=( const integer & );
};

// 2.7 arithmetic non-member functions:
const integer sqrt( const integer & );
void sqrtrem( const integer &, integer &, integer & );
void divrem( const integer &, const integer &, integer &, integer & );
const integer pow( const integer &, const integer & );
const integer mod( const integer &, const integer & );
const integer powmod( const integer &, const integer &, const integer & );
const integer invmod( const integer &, const integer & );
const integer gcd( const integer &, const integer & );
const integer lcm( const integer &, const integer & );
const integer extgcd( const integer &, const integer &, integer &, integer & );
void swap( integer &, integer & );
// 2.9 arithmetic non-member operators:
const integer operator++( integer &, int );
const integer operator--( integer &, int );
const integer operator+( const integer & );
const integer operator-( const integer & );
const integer operator+( const integer &, const integer & );
const integer operator-( const integer &, const integer & );
const integer operator*( const integer &, const integer & );
const integer operator/( const integer &, const integer & );
const integer operator%( const integer &, const integer & );
// 2.10 boolean non-member operators:
bool operator==( const integer &, const integer & );
bool operator!=( const integer &, const integer & );
bool operator<( const integer &, const integer & );
bool operator<=( const integer &, const integer & );
bool operator>( const integer &, const integer & );
bool operator>=( const integer &, const integer & );
// 2.12 shift non-member operators:
const integer operator<<( const integer &, size_type );
const integer operator>>( const integer &, size_type );
// 2.14 bitwise non-member operators:
const integer operator|( const integer &, const integer & );
const integer operator&( const integer &, const integer & );
const integer operator^( const integer &, const integer & );
// 2.15 conversion non-member functions:
int to_int( const integer & );
unsigned int to_unsigned_int( const integer & );
long to_long( const integer & );
unsigned long to_unsigned_long( const integer & );
long long to_long_long( const integer & );
unsigned long long to_unsigned_long_long( const integer & );
float to_float( const integer & );
double to_double( const integer & );
long double to_long_double( const integer & );
const std::string to_string( const integer & );
const std::string to_string( const integer &, radix_type );
// 2.16 stream non-member operators:
template<class charT, class traits>
std::basic_istream<charT, traits> &
operator>>( std::basic_istream<charT, traits> & , integer & );
template<class charT, class traits>
std::basic_ostream<charT, traits> &
operator<<( std::basic_ostream<charT, traits> & , const integer & );
// 2.17 random non-member functions:
const integer random( const integer &, const integer & );

class integer_allocator {
protected:
// 2.18 allocator constructors and destructor
   integer_allocator();
   virtual ~integer_allocator();
public:
// 2.19 allocator member functions
   void activate();
   void deactivate();
   virtual void * allocate( size_type ) = 0;
   virtual void * reallocate( void *, size_type ) = 0;
   virtual void deallocate( void * ) = 0;
};

class unsigned_integer : public integer {
public:
// 2.20 unsigned integer constructors and destructor
  unsigned_integer();
  unsigned_integer( int );
  unsigned_integer( unsigned int );
  unsigned_integer( long );
  unsigned_integer( unsigned long );
  unsigned_integer( long long );
  unsigned_integer( unsigned long long );
  explicit unsigned_integer( float );
  explicit unsigned_integer( double );
  explicit unsigned_integer( long double );
  explicit unsigned_integer( const char * );
  explicit unsigned_integer( const char *, radix_type );
  explicit unsigned_integer( const std::string & );
  explicit unsigned_integer( const std::string &, radix_type );
  unsigned_integer( const integer & );
  ~unsigned_integer();
// 2.21 unsigned integer member functions and operators
  unsigned_integer * clone() const;
  unsigned_integer & operator=( const integer & );
  integer_allocator * get_allocator() const;
  void normalize();
  void swap( integer & );
  unsigned_integer & negate();
  unsigned_integer & abs();
  unsigned_integer & operator++();
  unsigned_integer & operator--();
  unsigned_integer & operator+=( const integer & );
  unsigned_integer & operator-=( const integer & );
  unsigned_integer & operator*=( const integer & );
  unsigned_integer & operator/=( const integer & );
  unsigned_integer & operator%=( const integer & );
  unsigned_integer & operator<<=( size_type );
  unsigned_integer & operator>>=( size_type );
  unsigned_integer & operator|=( const integer & );
  unsigned_integer & operator&=( const integer & );
  unsigned_integer & operator^=( const integer & );
};

class modular_integer : public integer {
public:
// 2.22 modular integer constructors and destructor
  modular_integer();
  modular_integer( int );
  modular_integer( unsigned int );
  modular_integer( long );
  modular_integer( unsigned long );
  modular_integer( long long );
  modular_integer( unsigned long long );
  explicit modular_integer( float );
  explicit modular_integer( double );
  explicit modular_integer( long double );
  explicit modular_integer( const char * );
  explicit modular_integer( const char *, radix_type );
  explicit modular_integer( const std::string & );
  explicit modular_integer( const std::string &, radix_type );
  modular_integer( const integer & );
  ~modular_integer();
// 2.23 modular integer member functions and operators
  modular_integer * clone() const;
  modular_integer & operator=( const integer & );
  integer_allocator * get_allocator() const;
  void normalize();
  void swap( integer & );
  modular_integer & negate();
  modular_integer & abs();
  modular_integer & operator++();
  modular_integer & operator--();
  modular_integer & operator+=( const integer & );
  modular_integer & operator-=( const integer & );
  modular_integer & operator*=( const integer & );
  modular_integer & operator/=( const integer & );
  modular_integer & operator%=( const integer & );
  modular_integer & operator<<=( size_type );
  modular_integer & operator>>=( size_type );
  modular_integer & operator|=( const integer & );
  modular_integer & operator&=( const integer & );
  modular_integer & operator^=( const integer & );
// 2.24 modular integer static functions
  static void set_modulus( const integer & );
  static const integer & modulus();
};

class allocated_integer : public integer {
public:
// 2.25 allocated integer constructors and destructor
  allocated_integer();
  allocated_integer( int );
  allocated_integer( unsigned int );
  allocated_integer( long );
  allocated_integer( unsigned long );
  allocated_integer( long long );
  allocated_integer( unsigned long long );
  explicit allocated_integer( float );
  explicit allocated_integer( double );
  explicit allocated_integer( long double );
  explicit allocated_integer( const char * );
  explicit allocated_integer( const char *, radix_type );
  explicit allocated_integer( const std::string & );
  explicit allocated_integer( const std::string &, radix_type );
  allocated_integer( const integer & );
  ~allocated_integer();
// 2.26 allocated integer member functions and operators
  allocated_integer * clone() const;
  allocated_integer & operator=( const integer & );
  integer_allocator * get_allocator() const;
  void normalize();
  void swap( integer & );
  allocated_integer & negate();
  allocated_integer & abs();
  allocated_integer & operator++();
  allocated_integer & operator--();
  allocated_integer & operator+=( const integer & );
  allocated_integer & operator-=( const integer & );
  allocated_integer & operator*=( const integer & );
  allocated_integer & operator/=( const integer & );
  allocated_integer & operator%=( const integer & );
  allocated_integer & operator<<=( size_type );
  allocated_integer & operator>>=( size_type );
  allocated_integer & operator|=( const integer & );
  allocated_integer & operator&=( const integer & );
  allocated_integer & operator^=( const integer & );
// 2.27 allocated integer static functions
  static void set_allocator( integer_allocator * );
};

class allocated_unsigned_integer : public unsigned_integer {
public:
// 2.28 allocated unsigned integer constructors and destructor
  allocated_unsigned_integer();
  allocated_unsigned_integer( int );
  allocated_unsigned_integer( unsigned int );
  allocated_unsigned_integer( long );
  allocated_unsigned_integer( unsigned long );
  allocated_unsigned_integer( long long );
  allocated_unsigned_integer( unsigned long long );
  explicit allocated_unsigned_integer( float );
  explicit allocated_unsigned_integer( double );
  explicit allocated_unsigned_integer( long double );
  explicit allocated_unsigned_integer( const char * );
  explicit allocated_unsigned_integer( const char *, radix_type );
  explicit allocated_unsigned_integer( const std::string & );
  explicit allocated_unsigned_integer( const std::string &, radix_type );
  allocated_unsigned_integer( const integer & );
  ~allocated_unsigned_integer();
// 2.29 allocated unsigned integer member functions and operators
  allocated_unsigned_integer * clone() const;
  allocated_unsigned_integer & operator=( const integer & );
  integer_allocator * get_allocator() const;
  void normalize();
  void swap( integer & );
  allocated_unsigned_integer & negate();
  allocated_unsigned_integer & abs();
  allocated_unsigned_integer & operator++();
  allocated_unsigned_integer & operator--();
  allocated_unsigned_integer & operator+=( const integer & );
  allocated_unsigned_integer & operator-=( const integer & );
  allocated_unsigned_integer & operator*=( const integer & );
  allocated_unsigned_integer & operator/=( const integer & );
  allocated_unsigned_integer & operator%=( const integer & );
  allocated_unsigned_integer & operator<<=( size_type );
  allocated_unsigned_integer & operator>>=( size_type );
  allocated_unsigned_integer & operator|=( const integer & );
  allocated_unsigned_integer & operator&=( const integer & );
  allocated_unsigned_integer & operator^=( const integer & );
// 2.30 allocated integer static functions
  static void set_allocator( integer_allocator * );
};

class allocated_modular_integer : public modular_integer {
public:
// 2.31 allocated modular integer constructors and destructor
  allocated_modular_integer();
  allocated_modular_integer( int );
  allocated_modular_integer( unsigned int );
  allocated_modular_integer( long );
  allocated_modular_integer( unsigned long );
  allocated_modular_integer( long long );
  allocated_modular_integer( unsigned long long );
  explicit allocated_modular_integer( float );
  explicit allocated_modular_integer( double );
  explicit allocated_modular_integer( long double );
  explicit allocated_modular_integer( const char * );
  explicit allocated_modular_integer( const char *, radix_type );
  explicit allocated_modular_integer( const std::string & );
  explicit allocated_modular_integer( const std::string &, radix_type );
  allocated_modular_integer( const integer & );
  ~allocated_modular_integer();
// 2.32 allocated modular integer member functions and operators
  allocated_modular_integer * clone() const;
  allocated_modular_integer & operator=( const integer & );
  integer_allocator * get_allocator() const;
  void normalize();
  void swap( integer & );
  allocated_modular_integer & negate();
  allocated_modular_integer & abs();
  allocated_modular_integer & operator++();
  allocated_modular_integer & operator--();
  allocated_modular_integer & operator+=( const integer & );
  allocated_modular_integer & operator-=( const integer & );
  allocated_modular_integer & operator*=( const integer & );
  allocated_modular_integer & operator/=( const integer & );
  allocated_modular_integer & operator%=( const integer & );
  allocated_modular_integer & operator<<=( size_type );
  allocated_modular_integer & operator>>=( size_type );
  allocated_modular_integer & operator|=( const integer & );
  allocated_modular_integer & operator&=( const integer & );
  allocated_modular_integer & operator^=( const integer & );
// 2.33 allocated modular integer static functions
  static void set_allocator( integer_allocator * );
};
```

2.3 Constructors and Destructor [integer.ctors]

2.3.1 Rationale

The constructors can be used to convert the arithmetic base types and strings to `integer`. They can be called explicitly, for example in a declaration. The non-explicit constructors can also be called through implicit conversion [class.conv.ctor]. An implementation may add overloads of all functions and operators for all arithmetic base types, but this is not required, as long as the constructors below are available for implicit conversion. The destructor is virtual so that derivation from class `integer` is possible.

2.3.2

`integer();`

1 *Effects:* Constructs an object of class `integer`.

2 *Postconditions:* `integer() == 0`

3 *Complexity:* O(1)

2.3.3

`integer( int` *`arg`* `);`

4 *Effects:* Constructs an object of class `integer`.

5 *Postconditions:* `to_int( integer(` *`arg`* `) ) ==` *`arg`*.

6 *Complexity:* O(N)

2.3.4

`integer( unsigned int` *`arg`* `);`

7 *Effects:* Constructs an object of class `integer`.

8 *Postconditions:* `to_unsigned_int( integer(` *`arg`* `) ) ==` *`arg`*.

9 *Complexity:* O(N)

2.3.5

`integer( long` *`arg`* `);`

10 *Effects:* Constructs an object of class `integer`.

11 *Postconditions:* `to_long( integer(` *`arg`* `) ) ==` *`arg`*.

12 *Complexity:* O(N)

2.3.6

`integer( unsigned long` *`arg`* `);`

13 *Effects:* Constructs an object of class `integer`.

14 *Postconditions:* `to_unsigned_long( integer(` *`arg`* `) ) ==` *`arg`*.

15 *Complexity:* O(N)

2.3.7

`integer( long long` *`arg`* `);`

16 *Effects:* Constructs an object of class `integer`.

17 *Postconditions:* `to_long_long( integer(` *`arg`* `) ) ==` *`arg`*.

18 *Complexity:* O(N)

2.3.8

`integer( unsigned long long arg );`

19 *Effects:* Constructs an object of class `integer`.

20 *Postconditions:* `to_unsigned_long_long( integer( arg ) ) == arg`.

21 *Complexity:* O(N)

2.3.9

`explicit integer( float arg );`

22 *Effects:* Constructs an object of class `integer`.

23 *Postconditions:* `to_float( integer( arg ) ) == trunc( arg )`.

24 *Throws:* `conversion_error` when the value of *arg* is +infinity, -infinity or not a number.

25 *Remarks:* The `trunc` function truncates toward zero [lib.c.math].

26 *Complexity:* O(N)

2.3.10

`explicit integer( double arg );`

27 *Effects:* Constructs an object of class `integer`.

28 *Postconditions:* `to_double( integer( arg ) ) == trunc( arg )`.

29 *Throws:* `conversion_error` when the value of *arg* is +infinity, -infinity or not a number.

30 *Remarks:* The `trunc` function truncates toward zero [lib.c.math].

31 *Complexity:* O(N)

2.3.11

`explicit integer( long double arg );`

32 *Effects:* Constructs an object of class `integer`.

33 *Postconditions:* `to_long_double( integer( arg ) ) == trunc( arg )`.

34 *Throws:* `conversion_error` when the value of *`arg`* is +infinity, -infinity or not a number.

35 *Remarks:* The `trunc` function truncates toward zero [lib.c.math].

36 *Complexity:* O(N)

2.3.12

```
explicit integer( const char * arg );
```

37 *Effects:* Constructs an object of class `integer` with the value of the C-style string *`arg`*. When the numbers start with "0x" or "0X", hexadecimal notation is assumed; otherwise when the numbers start with "0", octal notation is assumed; otherwise decimal notation is assumed.

38 *Postconditions:* `to_string( integer( arg ) ) == std::string( arg )` (when decimal).

39 *Throws:* `invalid_input_error` when the string does not consist of only numbers (decimal, hexadecimal or octal), possibly headed by a `+` or `-` sign.

40 *Complexity:* O(M(N)log(N))

2.3.13

```
explicit integer( const char * arg, radix_type radix );
```

41 *Effects:* Constructs an object of class `integer` with the value of string *`arg`*.
Notation with base *`radix`* is assumed, where `2 <= radix <= 36`. For `radix > 10`, the letters may be uppercase or lowercase.

42 *Postconditions:*
`to_string( integer( arg, radix ), radix ) == std::string( arg )`.

43 *Throws:* `invalid_input_error` when not `2 <= radix <= 36`, or when the string does not consist of only numbers (or letters) of base *`radix`*, possibly headed by a `+` or `-` sign.

44 *Complexity:* O(M(N)log(N))

2.3.14

```
explicit integer( const std::string & arg );
```

45 *Effects:* Constructs an object of class `integer` with the value of string *`arg`*.
When the numbers start with "0x" or "0X", hexadecimal notation is assumed; otherwise when the numbers start with "0", octal notation is assumed; otherwise decimal notation is assumed.

46 *Postconditions:* `to_string( integer(` *`arg`* `) ) ==` *`arg`* (when decimal).

47 *Throws:* `invalid_input_error` when the string does not consist of only numbers (decimal, hexadecimal or octal), possibly headed by a `+` or `-` sign.

48 *Complexity:* O(M(N)log(N))

2.3.15

`explicit integer( const std::string &` *`arg`*`, radix_type` *`radix`* `);`

49 *Effects:* Constructs an object of class `integer` with the value of string *`arg`*.
Notation with base *`radix`* is assumed, where `2 <=` *`radix`* `<= 36`. For *`radix`* `> 10`, the letters may be uppercase or lowercase.

50 *Postconditions:* `to_string( integer(` *`arg`*`,` *`radix`* `),` *`radix`* `) ==` *`arg`*.

51 *Throws:* `invalid_input_error` when not `2 <=` *`radix`* `<= 36`, or when the string does not consist of only numbers (or letters) of base *`radix`*, possibly headed by a `+` or `-` sign.

52 *Complexity:* O(M(N)log(N))

2.3.16

`integer( const integer &` *`arg`* `);`

53 *Effects:* Copy constructs an object of class `integer`.

54 *Postconditions:* `integer(` *`arg`* `) ==` *`arg`*.

55 *Complexity:* O(N)

2.3.17

`virtual ~integer();`

56 *Effects:* Destructs an object of class `integer`.

57 *Complexity:* O(1)

2.4 Conversion Member Operators [integer.conv.mem.ops]

2.4.1 Rationale

This conversion member operator allows `integer` objects to be used in boolean contexts, such as: `if( x )`, which is equivalent to: `if( !x.is_zero() )`. A conversion member operator to `bool` cannot be used, because this would generate ambiguities in expressions.

2.4.2

`operator` *`unspecified-bool-type`*`() const;`

1 *Returns:* An unspecified value that, when used in boolean contexts, is equivalent to `!is_zero()`.

2 *Complexity:* O(1)

3 *Notes:* This conversion operator allows `integer` objects to be used in boolean contexts. One possible choice for the return type is a pointer to member function.

2.5 Copying and Assignment Member Functions [integer.copy]

2.5.1 Rationale

To make user-defined functions or operators virtual, the virtual copy constructor `clone()` can be used, and that objects corresponding virtual member function or operator can be called [5,7]. The member function `get_allocator()` is used by `swap()` to see if the pointers or the values must be swapped.

2.5.2

```
virtual integer * clone() const;
```

1 *Returns:* `new integer( *this );`

2 *Postconditions:* `*clone() == *this`.

3 *Complexity:* O(N)

4 *Notes:* This is a virtual copy constructor [5,7]. Each derived class must override this member function, calling the derived copy constructor.

2.5.3

`virtual integer & operator=( const integer &` *`rhs`* `);`

5 *Effects:* Assigns the value of *`rhs`* to `*this`.

6 *Postconditions:* `*this ==` *`rhs`*.

7 *Returns:* `*this`.

8 *Complexity:* O(N)

9 *Notes:* Derived classes must override this member operator and use it to convert (temporary) objects of type `integer` back to the derived class.

2.5.4

`virtual integer_allocator * get_allocator() const;`

10 *Returns:* a pointer to the default static allocator.

11 *Complexity:* O(1)

12 *Notes:* Derived classes with a static allocator must override this member function.

2.5.5

`virtual void normalize();`

13 *Effects:* None.

14 *Complexity:* O(1)

15 *Notes:* Derived classes with some form of normalization must override this member function.

2.5.6

`virtual void swap( integer &` *`arg`* `);`

16 *Effects:*
```
if( get_allocator() != arg.get_allocator() )
{ integer temp( arg );
  arg.operator=( *this );
  _swap( temp );
} else
{ _swap( arg );
  arg.normalize();
};
```

17 *Remarks:* `_swap()` swaps the pointers, the signs and the lengths.

18 *Complexity:* O(1) (O(N) when allocators differ, O(D(N)) when *arg* is modular)

19 *Notes:* Derived classes with a static allocator must override this member function.

2.6 Arithmetic Member Functions [integer.arith.mem.funs]

2.6.1 Rationale

The member function `get_sign()` may be used for efficient comparisons against zero, like in `x.get_sign() >= 0`. The virtual member functions can be overridden in a derived class; the non-virtual member functions only extract information from `integer` base class data, and must not be redefined.

2.6.2

`bool is_zero() const;`

1 *Returns:* `*this == 0`

2 *Complexity:* O(1)

2.6.3

`int get_sign() const;`

3 *Returns:* `*this == 0 ? 0 : ( *this > 0 ? 1 : -1 )`

4 *Complexity:* O(1)

2.6.4

`bool is_odd() const;`

5 *Returns:* `( *this & 1 ) == 1`

6 *Complexity:* O(1)

7 *Notes:* This is equivalent to `*this != 0 && lowest_bit() == 0`.

2.6.5

`size_type highest_bit() const;`

8 *Returns:* The bit number of the highest set bit.

9 *Throws:* `invalid_argument_error` when `*this == 0`.

10 *Remarks:* The bit numbering starts with zero. The result is independent of sign.

11 *Complexity:* O(N)

2.6.6

`size_type lowest_bit() const;`

12 *Returns:* The bit number of the lowest set bit.

13 *Throws:* `invalid_argument_error` when `*this == 0`.

14 *Remarks:* The bit numbering starts with zero. The result is independent of sign.

15 *Complexity:* O(N)

2.6.7

`const integer get_sub( size_type startbit, size_type nbits ) const;`

16 *Returns:* The sub integer with the `nbits` bits starting with bit number `startbit`.

17 *Remarks:* The bit numbering starts with zero. The result, when non-zero, preserves the sign, and the absolute value of the result is independent of sign.

18 *Complexity:* O(1) - O(N) depending on `nbits`

19 *Notes:* The return value is as if the integers are padded with zeros to infinity. This means that when `startbit > highest_bit()` or `nbits == 0` the result is 0.

2.6.8

`virtual integer & negate();`

20 *Effects:* `*this = -*this`

21 *Returns:* `*this`.

22 *Complexity:* O(1)

2.6.9

`virtual integer & abs();`

23 *Effects:* `if( *this < 0 ) negate()`

24 *Returns:* `*this`.

25 *Complexity:* O(1)

2.7 Arithmetic Non-Member Functions [integer.arith.nonmem.funs]

2.7.1 Rationale

The non-member function `swap()` calls the corresponding virtual member function. The other arithmetic non-member functions treat their arguments as `integer`. When the functions return `const integer`, the `const` makes expressions like `sqrt( x ) = 10` illegal [7].

2.7.2

`const integer sqrt( const integer & arg );`

1 *Returns:* $\lfloor\sqrt{arg}\rfloor$.

2 *Throws:* `invalid_argument_error` when *arg* `< 0`.

3 *Complexity:* O(M(N))

2.7.3

`void sqrtrem( const integer & arg, integer & res, integer & rem );`

4 *Effects:* $res = \lfloor\sqrt{arg}\rfloor$, $rem = arg - \lfloor\sqrt{arg}\rfloor^2$

5 *Throws:* `invalid_argument_error` when `arg < 0`.

6 *Complexity:* $\mathrm{O}(\mathrm{M}(N))$

7 *Notes:* this function is faster than calling `sqrt` and computing the remainder separately.

2.7.4

`void divrem( const integer & lhs, const integer & rhs, integer & quot, integer & rem );`

8 *Effects:* `quot = lhs / rhs, rem = lhs % rhs`.

9 *Throws:* `division_by_zero_error` when `rhs == 0`.

10 *Complexity:* $\mathrm{D}(N) = \mathrm{O}(\mathrm{M}(N))$

11 *Notes:* this function is about twice as fast as calling the `integer` operators `/` and `%` separately.

2.7.5

`const integer pow( const integer & x, const integer & n );`

12 *Returns:* x^n.

13 *Throws:* `invalid_argument_error` when `n < 0`.

14 *Complexity:* $\mathrm{O}(\mathrm{M}(\lfloor n/2\rfloor N_x))$

2.7.6

`const integer mod( const integer & x, const integer & y );`

15 *Returns:* `y == 0 ? x :` $x \bmod y$ `=` $x - y * \lfloor x/y\rfloor$.

16 *Complexity:* $\mathrm{D}(N) = \mathrm{O}(\mathrm{M}(N))$

17 *Notes:* This is not always equal to the remainder $x\%y$, see 2.8.8.

2.7.7

`const integer powmod( const integer & x, const integer & n, const integer & y );`

18 *Returns:* `y == 0 ?` x^n `:` $x^n \bmod y$.

19 *Throws:* `invalid_argument_error` when `n < 0`.

20 *Complexity:* $O(\log(n)M(N_m))$

21 *Notes:* this function is usually much faster than calling `pow` and `mod` separately.

2.7.8

`const integer invmod( const integer & x, const integer & y );`

22 *Returns:* $x^{-1} \bmod y$ when this modular inverse exists, `0` when it does not exist.

23 *Throws:* `invalid_argument_error` when `y <= 0`, and `division_by_zero_error` when `x == 0`.

24 *Complexity:* $O(G(N))$

2.7.9

`const integer gcd( const integer & x, const integer & y );`

25 *Returns:* the greatest common divisor of `x` and `y`.

26 *Complexity:* $G(N) = O(N^2)$

2.7.10

`const integer lcm( const integer & x, const integer & y );`

27 *Returns:* the least common multiple of `x` and `y`.

28 *Complexity:* $O(G(N))$

2.7.11

`const integer extgcd( const integer &` *x*`, const integer &` *y*`, integer &` *a*`, integer &` *b* `);`

29 *Effects:* computes the greatest common divisor of *x* and *y*, and computes *a* and *b* such that $xa + yb = gcd(x, y)$.

30 *Returns:* the greatest common divisor of *x* and *y*.

31 *Complexity:* O(G(N))

2.7.12

`void swap( integer &` *x*`, integer &` *y* `);`

32 *Effects:* *x*`.swap(` *y* `);`

33 *Complexity:* O(1) (O(N) when allocators differ, O(D(N)) when *x* or *y* is modular)

2.8 Arithmetic Member Operators [integer.arith.mem.ops]

2.8.1 Rationale

The member operators that are virtual can be overridden in a derived class.

2.8.2

`virtual integer & operator++();`

1 *Effects:* Increments `*this` by one and stores the result in `*this`.

2 *Returns:* `*this`.

3 *Remarks:* unary prefix operator.

4 *Complexity:* O(1) amortized

2.8.3

`virtual integer & operator--();`

5 *Effects:* Decrements `*this` by one and stores the result in `*this`.

6 *Returns:* `*this`.

7 *Remarks:* unary prefix operator.

8 *Complexity:* O(1) amortized

2.8.4

`virtual integer & operator+=( const integer & rhs );`

9 *Effects:* Adds *`rhs`* to `*this` and stores the result in `*this`.

10 *Returns:* `*this`.

11 *Complexity:* O(N)

2.8.5

`virtual integer & operator-=( const integer & rhs );`

12 *Effects:* Subtracts *`rhs`* from `*this` and stores the result in `*this`.

13 *Returns:* `*this`.

14 *Complexity:* O(N)

2.8.6

`virtual integer & operator*=( const integer & rhs );`

15 *Effects:* Multiplies `*this` with *`rhs`* and stores the result in `*this`.

16 *Returns:* `*this`.

17 *Complexity:* $\mathrm{M}(N) = \mathrm{O}(< N^2)$

2.8.7

`virtual integer & operator/=( const integer & rhs );`

18 *Effects:* Divides `*this` by *rhs* and stores the result in `*this`, the result being truncated toward zero.

19 *Returns:* `*this`.

20 *Throws:* `division_by_zero_error` when *rhs* is zero.

21 *Complexity:* $\mathrm{D}(N) = \mathrm{O}(\mathrm{M}(N))$

2.8.8

`virtual integer & operator%=( const integer & rhs );`

22 *Effects:* Divides `*this` by *rhs* and stores the remainder in `*this`, the remainder being $x\%y = x - y * \mathrm{trunc}(x/y)$, where the trunc function truncates toward zero.

23 *Returns:* `*this`.

24 *Throws:* `division_by_zero_error` when *rhs* is zero.

25 *Complexity:* $\mathrm{D}(N) = \mathrm{O}(\mathrm{M}(N))$

2.9 Arithmetic Non-Member Operators [integer.arith.nonmem.ops]

2.9.1 Rationale

These non-member operators must not be redefined in a derived class. These operators return `const integer`; the `const` makes expressions like `x + y = 10` illegal [7].

2.9.2

`const integer operator++( integer & arg , int );`

1 *Returns:*
```
integer temp( arg );
++arg;
return temp;
```

2 *Remarks:* Unary postfix operator

3 *Complexity:* $\mathrm{O}(N)$

2.9.3

`const integer operator--( integer & arg , int );`

4 *Returns:*
```
integer temp( arg );
--arg;
return temp;
```

5 *Remarks:* Unary postfix operator

6 *Complexity:* O(N)

2.9.4

`const integer operator+( const integer & arg  );`

7 *Returns:* `arg`

8 *Remarks:* Unary operator

9 *Complexity:* O(N)

2.9.5

`const integer operator-( const integer & arg  );`

10 *Returns:* `integer( arg ).negate()`

11 *Remarks:* Unary operator

12 *Complexity:* O(N)

13 *Notes:* The `integer` member negation operator is called.

2.9.6

`const integer operator+( const integer & lhs, const integer & rhs );`

14 *Returns:* `integer( lhs ) += rhs`.

15 *Complexity:* O(N)

16 *Notes:* The `integer` member operator is called.

2.9.7

`const integer operator-( const integer & lhs, const integer & rhs );`

17 *Returns:* `integer( lhs ) -= rhs`.

18 *Complexity:* $\mathrm{O}(N)$

19 *Notes:* The `integer` member operator is called.

2.9.8

`const integer operator*( const integer & lhs, const integer & rhs );`

20 *Returns:* `integer( lhs ) *= rhs`.

21 *Complexity:* $\mathrm{M}(N) = \mathrm{O}(< N^2)$

22 *Notes:* The `integer` member operator is called.

2.9.9

`const integer operator/( const integer & lhs, const integer & rhs );`

23 *Returns:* `integer( lhs ) /= rhs`.

24 *Throws:* `division_by_zero_error` when *rhs* is zero.

25 *Complexity:* $\mathrm{D}(N) = \mathrm{O}(\mathrm{M}(N))$

26 *Notes:* The `integer` member operator is called.

2.9.10

`const integer operator%( const integer & lhs, const integer & rhs );`

27 *Returns:* `integer( lhs ) %= rhs`.

28 *Throws:* `division_by_zero_error` when *rhs* is zero.

29 *Complexity:* $\mathrm{D}(N) = \mathrm{O}(\mathrm{M}(N))$

30 *Notes:* The `integer` member operator is called.

2.10 Boolean Non-Member Operators [integer.bool.nonmem.ops]

2.10.1 Rationale

The boolean non-member operators are used for boolean comparison between two `integer`s. They must not be redefined in a derived class.

2.10.2

`bool operator==( const integer & lhs, const integer & rhs );`

1 *Returns:* `true` if *lhs* equals *rhs*, otherwise `false`.

2 *Complexity:* O(N)

2.10.3

`bool operator!=( const integer & lhs, const integer & rhs );`

3 *Returns:* `!( lhs == rhs )`.

4 *Complexity:* O(N)

2.10.4

`bool operator<( const integer & lhs, const integer & rhs );`

5 *Returns:* `true` if *lhs* is less than *rhs*, otherwise `false`.

6 *Complexity:* O(N)

2.10.5

`bool operator<=( const integer & lhs, const integer & rhs );`

7 *Returns:* `lhs < rhs || lhs == rhs`.

8 *Complexity:* O(N)

2.10.6

`bool operator>( const integer & lhs, const integer & rhs );`

9 *Returns:* `!( lhs <= rhs )`.

10 *Complexity:* O(N)

2.10.7

`bool operator>=( const integer & lhs, const integer & rhs );`

11 *Returns:* `!( lhs < rhs )`.

12 *Complexity:* $O(N)$

2.11 Shift Member Operators [integer.shift.mem.ops]

2.11.1 Rationale

`x <<= n` is equivalent to `x *=` 2^n, and `x >>= n` is equivalent to `x /=` 2^n. These member operators are virtual and can be overridden in a derived class.

2.11.2

`virtual integer & operator<<=( size_type rhs );`

1 *Effects:* Shifts `*this` to the left by *`rhs`* bits and stores the result in `*this`, leaving the sign of `*this` unchanged.

2 *Returns:* `*this`.

3 *Complexity:* $O(N)$

2.11.3

`virtual integer & operator>>=( size_type rhs );`

4 *Effects:* Shifts `*this` to the right by *`rhs`* bits and stores the result in `*this`, leaving the sign of `*this` unchanged.

5 *Returns:* `*this`.

6 *Remarks:* As the sign is stored separately, no distinction between arithmetic and logical shift exists.

7 *Complexity:* $O(N)$

2.12 Shift Non-Member Operators [integer.shift.nonmem.ops]

2.12.1 Rationale

`x << n` is equivalent to `x *` 2^n, and `x >> n` is equivalent to `x /` 2^n. These non-member operators must not be redefined in a derived class. These operators return `const integer`; the `const` makes expressions like `x << y = 10` illegal [7].

2.12.2

`const integer operator<<( const integer &` *`lhs`*`, size_type` *`rhs`* `);`

1 *Returns:* `integer(` *`lhs`* `) <<=` *`rhs`*.

2 *Complexity:* O(N)

3 *Notes:* The `integer` member operator is called.

2.12.3

`const integer operator>>( const integer &` *`lhs`*`, size_type` *`rhs`* `);`

4 *Returns:* `integer(` *`lhs`* `) >>=` *`rhs`*.

5 *Remarks:* As the sign is stored separately, no distinction between arithmetic and logical shift exists.

6 *Complexity:* O(N)

7 *Notes:* The `integer` member operator is called.

2.13 Bitwise Member Operators [integer.bit.mem.ops]

2.13.1 Rationale

The bitwise operators perform a logical function on two `integer`s, bit by bit, assuming that both `integer`s are padded with zero-bits to infinity. The sign of the result is computed as if the sign was also a bit, which is one when negative and zero otherwise. These member operators are virtual and can be overridden in a derived class.

2.13.2

`virtual integer & operator|=( const integer & rhs );`

1 *Effects:* Bitwise ORs `*this` with *`rhs`* and stores the result in `*this`.

2 *Returns:* `*this`.

3 *Complexity:* O(N)

2.13.3

`virtual integer & operator&=( const integer & rhs );`

4 *Effects:* Bitwise ANDs `*this` with *`rhs`* and stores the result in `*this`.

5 *Returns:* `*this`.

6 *Complexity:* O(N)

2.13.4

`virtual integer & operator^=( const integer & rhs );`

7 *Effects:* Bitwise XORs `*this` with *`rhs`* and stores the result in `*this`.

8 *Returns:* `*this`.

9 *Complexity:* O(N)

2.14 Bitwise Non-Member Operators [integer.bit.nonmem.ops]

2.14.1 Rationale

The bitwise operators perform a logical function on two `integer`s, bit by bit, assuming that both `integer`s are padded with zero-bits to infinity. The sign of the result is computed as if the sign was also a bit, which is one when negative and zero otherwise. These non-member operators must not be redefined in a derived class. These operators return `const integer`; the `const` makes expressions like `x | y = 10` illegal [7].

2.14.2

`const integer operator|( const integer & lhs, const integer & rhs );`

1 *Returns:* `integer( lhs ) |= rhs`.

2 *Complexity:* O(N)

3 *Notes:* The `integer` member operator is called.

2.14.3

`const integer operator&( const integer & lhs, const integer & rhs );`

4 *Returns:* `integer( lhs ) &= rhs`.

5 *Complexity:* O(N)

6 *Notes:* The `integer` member operator is called.

2.14.4

`const integer operator^( const integer & lhs, const integer & rhs );`

7 *Returns:* `integer( lhs ) ^= rhs`.

8 *Complexity:* O(N)

9 *Notes:* The `integer` member operator is called.

2.15 Conversion Non-Member Functions [integer.conv.nonmem.funs]

2.15.1 Rationale

Conversion operators from `integer` to the arithmetic base types and strings are not provided, as they may lead to ambiguities in expressions [5]. Instead, for conversion from `integer` to the arithmetic base types and strings, conversion non-member functions are provided.

2.15.2

`int to_int( const integer & arg );`

1 *Returns:* the value of `arg` converted to `int`.

2 *Postconditions:* `integer( to_int( arg ) ) == arg`.

3 *Throws:* `invalid_argument_error` when the value does not fit into an `int`.

4 *Complexity:* O(1)

2.15.3

`unsigned int to_unsigned_int( const integer &` *`arg`* `);`

5 *Returns:* the value of *`arg`* converted to `unsigned int`.

6 *Postconditions:* `integer( to_unsigned_int(` *`arg`* `) ) ==` *`arg`*.

7 *Throws:* `invalid_argument_error` when the value is negative or does not fit into an `unsigned int`.

8 *Complexity:* O(1)

2.15.4

`long to_long( const integer &` *`arg`* `);`

9 *Returns:* the value of *`arg`* converted to `long`.

10 *Postconditions:* `integer( to_long(` *`arg`* `) ) ==` *`arg`*.

11 *Throws:* `invalid_argument_error` when the value does not fit into an `long`.

12 *Complexity:* O(1)

2.15.5

`unsigned long to_unsigned_long( const integer &` *`arg`* `);`

13 *Returns:* the value of *`arg`* converted to `unsigned long`.

14 *Postconditions:* `integer( to_unsigned_long(` *`arg`* `) ) ==` *`arg`*.

15 *Throws:* `invalid_argument_error` when the value is negative or does not fit into an `unsigned long`.

16 *Complexity:* O(1)

2.15.6

`long long to_long_long( const integer & arg );`

17 *Returns:* the value of *`arg`* converted to `long long`.

18 *Postconditions:* `integer( to_long_long( arg ) ) == arg`.

19 *Throws:* `invalid_argument_error` when the value does not fit into an `long long`.

20 *Complexity:* O(1)

2.15.7

`unsigned long long to_unsigned_long_long( const integer & arg );`

21 *Returns:* the value of *`arg`* converted to `unsigned long long`.

22 *Postconditions:* `integer( to_unsigned_long_long( arg ) ) == arg`.

23 *Throws:* `invalid_argument_error` when the value is negative or does not fit into an `unsigned long long`.

24 *Complexity:* O(1)

2.15.8

`float to_float( const integer & arg );`

25 *Returns:* the value of *`arg`* converted to `float`.

26 *Throws:* `invalid_argument_error` when the absolute value is greater than the range of a `float`.

27 *Remarks:* the value of *`arg`* may be truncated toward zero to fit into a `float`.

28 *Complexity:* O(1)

2.15.9

`double to_double( const integer & arg );`

29 *Returns:* the value of *`arg`* converted to `double`.

30 *Throws:* `invalid_argument_error` when the absolute value is greater than range of a `double`.

31 *Remarks:* the value of `arg` may be truncated toward zero to fit into a `double`.

32 *Complexity:* O(1)

2.15.10

`long double to_long_double( const integer & arg );`

33 *Returns:* the value of `arg` converted to `long double`.

34 *Throws:* `invalid_argument_error` when the absolute value is greater than the range of a `long double`.

35 *Remarks:* the value of `arg` may be truncated toward zero to fit into a `long double`.

36 *Complexity:* O(1)

2.15.11

`const std::string to_string( const integer & arg );`

37 *Returns:* the value of `arg` converted to `std::string`, where decimal notation is assumed.

38 *Postconditions:* `integer( to_string( arg ) ) == arg`.

39 *Complexity:* O(D(N)log(N))

2.15.12

`const std::string to_string( const integer & arg, radix_type radix );`

40 *Returns:* the value of `arg` converted to `std::string`. Notation with base `radix` is assumed, where `2 <= radix <= 36`. For `radix > 10`, the letters are lowercase.

41 *Postconditions:* `integer( to_string( arg, radix ), radix ) == arg`.

42 *Throws:* `invalid_input_error` when not `2 <= radix <= 36`.

43 *Complexity:* O(D(N)log(N))

2.16 Stream Non-Member Operators [integer.stream.nonmem.ops]

2.16.1 Rationale

The stream non-member operators provide a way to read an `integer` from instreams [lib.istream] and to write an `integer` to outstreams [lib.ostream]. Characters from the streams character set are converted to and from simple `char`s using the streams `narrow()` and `widen()` member functions [lib.basic.ios.members]. The numeric base or radix of the numbers read and written are controlled by the `basefield` flags of the streams [lib.fmtflags.state], which can be changed by the user with the `dec`, `hex` and `oct` stream manipulators [lib.basefield.manip]; by default, the `basefield` flag is set to decimal.

2.16.2

```
template<class charT, class traits>
std::basic_istream<charT, traits> &
operator>>( std::basic_istream<charT, traits> & lhs, integer & rhs );
```

1 *Effects:* An integer value is read from *`lhs`*, converted from decimal, hexadecimal or octal (depending on the `basefield` flag of *`lhs`*) to binary, and stored into *`rhs`*. For each base, the integer read is represented by its absolute value, possibly headed with a `+` or `-` sign, and it is negative only when headed with a `-` sign. Leading whitespaces (determined with `isspace`) are skipped. When hexadecimal, the absolute value can be preceded with `0x` or `0X`, and the hexadecimal letters can be uppercase or lowercase. When octal, the absolute value can be preceded with `0`.

2 *Returns:* *`lhs`*.

3 *Throws:* invalid_input when the input read does not consist of only numbers (decimal, hexadecimal or octal, see above) possibly headed with a `+` or `-` sign.

4 *Remarks:* The `basefield` flag can be changed by the user with the `dec`, `hex` and `oct` stream manipulators.

5 *Complexity:* O(M(N)log(N))

2.16.3

```
template<class charT, class traits>
std::basic_ostream<charT, traits> &
operator<<( std::basic_ostream<charT, traits> & lhs, const integer & rhs );
```

6 *Effects:* The integer value of *`rhs`* is converted from binary to decimal, hexadecimal or octal (depending on the `basefield` flag of *`lhs`*), and written to *`lhs`*. For each base, the integer written is represented by its absolute value, headed with a `-` sign only when it is negative, and headed with a `+` sign only when it is positive or zero and the `showpos` flag of *`lhs`* is set. The hexadecimal letters are uppercase or lowercase depending on the `uppercase` flag of *`lhs`*. When the `showbase` flag of *`lhs`* is set, the absolute value is preceded with `0x` or `0X` (depending on the `uppercase` flag) when hexadecimal, and with `0` when octal. When the field width is not zero, the field width, `adjustfield` flag and fill character are used to fill the field if necessary.

7 *Returns:* *`lhs`*.

8 *Remarks:* The `basefield` flag can be changed by the user with the `dec`, `hex` and `oct` stream manipulators, and the other flags with the `showpos`, `noshowpos`, `uppercase`, `nouppercase`, `showbase`, `noshowbase`, `setw(int)`, `setfill(char)`, `left`, `right` and `internal` stream manipulators.

9 *Complexity:* O(D(N)log(N))

2.17 Random Non-Member Functions [integer.rand.nonmem.funs]

2.17.1 Rationale

This function generates a uniformly distributed random `integer` in a certain interval using a specific pseudo-random number engine [tr.rand.eng]. It may be used to test other `integer` functions and operators. For generating non-uniformly distributed random `integer`s, or for using other pseudo-random number engines, the random distribution class templates [tr.rand.dist] may be used.

2.17.2

`const integer random( const integer & min, const integer & max );`

1 *Returns:* An `integer` random value between *`min`* and *`max`* inclusive. The random number is uniformly distributed and generated using a static default constructed predefined pseudo-random number engine of type `std::tr1::minstd_rand0` [tr.rand.predef].

2 *Throws:* `invalid_argument_error` when *`min`* > *`max`*.

3 *Remarks:* The random `integer` generated is platform independent.

4 *Complexity:* O(N)

2.18 Allocator Constructors and Destructor [integer.allocator.ctors]

2.18.1 Rationale

The abstract class `integer_allocator` provides the interface to provide a user-defined derived allocator class for class `integer`. Classes derived from this class can only be used as static allocators for other classes derived from class `integer` (1.5).

2.18.2

`integer_allocator();`

1 *Effects:* When called from a derived class, constructs an object of class `integer_allocator`.

2 *Remarks:* The derived constructor can do memory management initialization, such as creating a heap handle.

3 *Complexity:* O(1)

2.18.3

`virtual ~integer_allocator();`

4 *Effects:* When called from a derived class, destructs an object of class `integer_allocator`.

5 *Remarks:* The derived destructor can do memory management cleanup.

6 *Complexity:* O(1)

2.19 Allocator Member Functions [integer.allocator.mem.funs]

2.19.1 Rationale

The abstract class `integer_allocator` provides the interface to provide a user-defined derived allocator class for class `integer`. Classes derived from this class can only be used as static allocators for other classes derived from class `integer` (1.5).

2.19.2

`void activate();`

1 *Effects:* Makes the static allocator active for all `integer` memory allocations.

2 *Complexity:* O(1)

3 *Notes:* This function must not be redefined in a derived class.

2.19.3

`void deactivate();`

4 *Effects:* Makes the static allocator deactive for all `integer` memory allocations.

5 *Complexity:* O(1)

6 *Notes:* This function must not be redefined in a derived class.

2.19.4

`virtual void * allocate( size_type nbytes ) = 0;`

7 *Returns:* A pointer to an allocated memory block of size *`nbytes`* bytes.

8 *Remarks:* The allocated memory block does not need to be initalized.

9 *Complexity:* O(1)

2.19.5

`virtual void * reallocate( void * pdata, size_type nbytes ) = 0;`

10 *Returns:* A pointer to a reallocated memory block, first pointed to by *`pdata`*, of size *`nbytes`* bytes.

11 *Remarks:* The reallocated memory block does not need to be initalized.

12 *Complexity:* O(1)

2.19.6

`virtual void deallocate( void * pdata ) = 0;`

13 *Effects:* The memory block pointed to by *`pdata`* is deallocated.

14 *Complexity:* O(1)

2.20 Unsigned Integer Constructors and Destructor

2.20.1 Rationale [integer.unsigned.ctors]

The class `unsigned_integer` is derived from class `integer`. The `unsigned_integer` constructors are equivalent to the `integer` constructors, except that when the result is negative, an exception is thrown, by calling the derived `normalize()`.

2.20.2

`unsigned_integer();`

1 *Effects:* Constructs an object of class `unsigned_integer`, calling `integer()`.

2 *Complexity:* O(1)

2.20.3

`unsigned_integer( int` *`arg`* `);`

3 *Effects:* Constructs an object of class `unsigned_integer`, calling `integer(` *`arg`* `)`, `unsigned_integer::normalize();`

4 *Complexity:* O(N)

2.20.4

`unsigned_integer( unsigned int` *`arg`* `);`

5 *Effects:* Constructs an object of class `unsigned_integer`, calling `integer(` *`arg`* `)`.

6 *Complexity:* O(N)

2.20.5

`unsigned_integer( long` *`arg`* `);`

7 *Effects:* Constructs an object of class `unsigned_integer`, calling `integer(` *`arg`* `)`, `unsigned_integer::normalize();`

8 *Complexity:* O(N)

2.20.6

`unsigned_integer( unsigned long` *`arg`* `);`

9 *Effects:* Constructs an object of class `unsigned_integer`, calling `integer(` *`arg`* `)`.

10 *Complexity:* O(N)

2.20.7

`unsigned_integer( long long arg );`

11 *Effects:* Constructs an object of class `unsigned_integer`,
calling `integer( arg )`,
`unsigned_integer::normalize();`

12 *Complexity:* O(N)

2.20.8

`unsigned_integer( unsigned long long arg );`

13 *Effects:* Constructs an object of class `unsigned_integer`,
calling `integer( arg )`.

14 *Complexity:* O(N)

2.20.9

`explicit unsigned_integer( float arg );`

15 *Effects:* Constructs an object of class `unsigned_integer`,
calling `integer( arg )`,
`unsigned_integer::normalize();`

16 *Complexity:* O(N)

2.20.10

`explicit unsigned_integer( double arg );`

17 *Effects:* Constructs an object of class `unsigned_integer`,
calling `integer( arg )`,
`unsigned_integer::normalize();`

18 *Complexity:* O(N)

2.20.11

`explicit unsigned_integer( long double arg );`

19 *Effects:* Constructs an object of class `unsigned_integer`,
calling `integer( arg )`,
`unsigned_integer::normalize();`

20 *Complexity:* O(N)

2.20.12

`explicit unsigned_integer( const char * arg );`

21 *Effects:* Constructs an object of class `unsigned_integer`,
calling `integer( arg )`,
`unsigned_integer::normalize();`

22 *Complexity:* O(M(N)log(N))

2.20.13

`explicit unsigned_integer( const char * arg, radix_type radix );`

23 *Effects:* Constructs an object of class `unsigned_integer`,
calling `integer( arg, radix )`,
`unsigned_integer::normalize();`

24 *Complexity:* O(M(N)log(N))

2.20.14

`explicit unsigned_integer( const std::string & arg );`

25 *Effects:* Constructs an object of class `unsigned_integer`,
calling `integer( arg )`,
`unsigned_integer::normalize();`

26 *Complexity:* O(M(N)log(N))

2.20.15

`explicit unsigned_integer( const std::string & arg, radix_type radix );`

27 *Effects:* Constructs an object of class `unsigned_integer`,
calling `integer( arg, radix )`,
`unsigned_integer::normalize();`

28 *Complexity:* O(M(N)log(N))

2.20.16

`unsigned_integer( const integer & arg );`

29 *Effects:* Copy constructs an object of class `unsigned_integer`,
calling `integer( arg )`,
`unsigned_integer::normalize();`

30 *Complexity:* O(N)

2.20.17

`~unsigned_integer();`

31 *Effects:* Destructs an object of class `unsigned_integer`.

32 *Complexity:* O(1)

2.21 Unsigned Integer Member Functions and Operators

2.21.1 Rationale [integer.unsigned.funsops]

The class `unsigned_integer` is derived from class `integer`. The `unsigned_integer` member functions and operators are equivalent to the `integer` member functions and operators, except that when the result is negative, an exception is thrown, by calling the derived `normalize()`.

2.21.2

`unsigned_integer * clone() const;`

1 *Returns:* `new unsigned_integer( *this );`

2 *Postconditions:* `*clone() == *this`.

3 *Complexity:* O(N)

2.21.3

`unsigned_integer & operator=( const integer & rhs );`

4 *Effects:*
```
integer::operator=( rhs );
unsigned_integer::normalize();
```

5 *Returns:* `*this`.

6 *Complexity:* O(N)

2.21.4

`integer_allocator * get_allocator() const;`

7 *Returns:* `integer::get_allocator()`

8 *Complexity:* O(1)

2.21.5

`void normalize();`

20 *Effects:* None.

21 *Throws:* `unsigned_is_negative` when `*this < 0`.

22 *Complexity:* O(1)

2.21.6

`void swap( integer & arg );`

9 *Effects:*
```
if( get_allocator() != arg.get_allocator() )
{ unsigned_integer temp( arg );
  arg.operator=( *this );
  _swap( temp );
} else
{ _swap( arg );
  unsigned_integer::normalize();
  arg.normalize();
};
```

10 *Remarks:* `_swap()` swaps the pointers, the signs and the lengths.

11 *Complexity:* O(1) (O(N) when allocators differ, O(D(N)) when *`arg`* is modular)

2.21.7

`unsigned_integer & negate();`

12 *Effects:*
```
integer::negate();
unsigned_integer::normalize();
```

13 *Returns:* `*this`.

14 *Complexity:* O(1)

2.21.8

`unsigned_integer & abs();`

15 *Effects:* None.

16 *Returns:* `*this`.

17 *Complexity:* O(1)

2.21.9

`unsigned_integer & operator++();`

18 *Effects:*
```
integer::operator++();
```

19 *Returns:* `*this`.

20 *Remarks:* unary prefix operator.

21 *Complexity:* O(1) amortized

2.21.10

`unsigned_integer & operator--();`

22 *Effects:*
```
integer::operator--();
unsigned_integer::normalize();
```

23 *Returns:* `*this`.

24 *Remarks:* unary prefix operator.

25 *Complexity:* O(1) amortized

2.21.11

`unsigned_integer & operator+=( const integer & rhs );`

26 *Effects:*
```
integer::operator+=( rhs );
unsigned_integer::normalize();
```

27 *Returns:* `*this`.

28 *Complexity:* O(N)

2.21.12

`unsigned_integer & operator-=( const integer & rhs );`

29 *Effects:*
```
integer::operator-=( rhs );
unsigned_integer::normalize();
```

30 *Returns:* `*this`.

31 *Complexity:* O(N)

2.21.13

`unsigned_integer & operator*=( const integer & rhs );`

32 *Effects:*
```
integer::operator*=( rhs );
unsigned_integer::normalize();
```

33 *Returns:* `*this`.

34 *Complexity:* $\mathrm{M}(N) = \mathrm{O}(< N^2)$

2.21.14

```
unsigned_integer & operator/=( const integer & rhs );
```

35 *Effects:*
```
integer::operator/=( rhs );
unsigned_integer::normalize();
```

36 *Returns:* `*this`.

37 *Complexity:* $\mathrm{D}(N) = \mathrm{O}(\mathrm{M}(N))$

2.21.15

```
unsigned_integer & operator%=( const integer & rhs );
```

38 *Effects:*
```
integer::operator%=( rhs );
unsigned_integer::normalize();
```

39 *Returns:* `*this`.

40 *Complexity:* $\mathrm{D}(N) = \mathrm{O}(\mathrm{M}(N))$

2.21.16

```
unsigned_integer & operator<<=( size_type rhs );
```

41 *Effects:*
```
integer::operator<<=( rhs );
```

42 *Returns:* `*this`.

43 *Complexity:* $\mathrm{O}(N)$

2.21.17

`unsigned_integer & operator>>=( size_type rhs );`

44 *Effects:*
```
integer::operator>>=( rhs );
```

45 *Returns:* `*this`.

46 *Complexity:* O(N)

2.21.18

`unsigned_integer & operator|=( const integer & rhs );`

47 *Effects:*
```
integer::operator|=( rhs );
unsigned_integer::normalize();
```

48 *Returns:* `*this`.

49 *Complexity:* O(N)

2.21.19

`unsigned_integer & operator&=( const integer & rhs );`

50 *Effects:*
```
integer::operator&=( rhs );
unsigned_integer::normalize();
```

51 *Returns:* `*this`.

52 *Complexity:* O(N)

2.21.20

`unsigned_integer & operator^=( const integer & rhs );`

53 *Effects:*
```
integer::operator^=( rhs );
unsigned_integer::normalize();
```

54 *Returns:* `*this`.

55 *Complexity:* O(N)

2.22 Modular Integer Constructors and Destructor

2.22.1 Rationale [integer.modular.ctors]

The class `modular_integer` is derived from class `integer`. The `modular_integer` constructors are equivalent to the `integer` constructors, except that when the result is not between zero and the static modulus, the result is reduced modulo the static modulus using the `mod` function (2.7.6), by calling the derived `normalize()`. The static modulus is set with the static function `set_modulus()`, and is given here the name *_stat_mod*.

2.22.2

`modular_integer();`

1 *Effects:* Constructs an object of class `modular_integer`,
calling `integer()`.

2 *Complexity:* O(1)

2.22.3

`modular_integer( int` *arg* `);`

3 *Effects:* Constructs an object of class `modular_integer`,
calling `integer(` *arg* `),`
`modular_integer::normalize();`

4 *Postconditions:* `modular_integer(` *arg* `) == mod(` *arg*`,` *_stat_mod* `).`

5 *Complexity:* O(D(N))

2.22.4

`modular_integer( unsigned int` *arg* `);`

6 *Effects:* Constructs an object of class `modular_integer`,
calling `integer(` *arg* `),`
`modular_integer::normalize();`

7 *Postconditions:* `modular_integer(` *arg* `) == mod(` *arg*`,` *_stat_mod* `).`

8 *Complexity:* O(D(N))

2.22.5

`modular_integer( long *arg* );`

9 *Effects:* Constructs an object of class `modular_integer`,
calling `integer( arg )`,
`modular_integer::normalize();`

10 *Postconditions:* `modular_integer( arg ) == mod( arg, _stat_mod ).`

11 *Complexity:* O(D(N))

2.22.6

`modular_integer( unsigned long *arg* );`

12 *Effects:* Constructs an object of class `modular_integer`,
calling `integer( arg )`,
`modular_integer::normalize();`

13 *Postconditions:* `modular_integer( arg ) == mod( arg, _stat_mod ).`

14 *Complexity:* O(D(N))

2.22.7

`modular_integer( long long *arg* );`

15 *Effects:* Constructs an object of class `modular_integer`,
calling `integer( arg )`,
`modular_integer::normalize();`

16 *Postconditions:* `modular_integer( arg ) == mod( arg, _stat_mod ).`

17 *Complexity:* O(D(N))

2.22.8

`modular_integer( unsigned long long *arg* );`

18 *Effects:* Constructs an object of class `modular_integer`,
calling `integer( arg )`,
`modular_integer::normalize();`

19 *Postconditions:* `modular_integer( arg ) == mod( arg, _stat_mod ).`

20 *Complexity:* O(D(N))

2.22.9

`explicit modular_integer( float` *`arg`* `);`

21 *Effects:* Constructs an object of class `modular_integer`,
calling `integer(` *`arg`* `)`,
`modular_integer::normalize();`

22 *Postconditions:* `modular_integer(` *`arg`* `) == mod( integer(` *`arg`* `),` *`_stat_mod`* `)`.

23 *Complexity:* O(D(N))

2.22.10

`explicit modular_integer( double` *`arg`* `);`

24 *Effects:* Constructs an object of class `modular_integer`,
calling `integer(` *`arg`* `)`,
`modular_integer::normalize();`

25 *Postconditions:* `modular_integer(` *`arg`* `) == mod( integer(` *`arg`* `),` *`_stat_mod`* `)`.

26 *Complexity:* O(D(N))

2.22.11

`explicit modular_integer( long double` *`arg`* `);`

27 *Effects:* Constructs an object of class `modular_integer`,
calling `integer(` *`arg`* `)`,
`modular_integer::normalize();`

28 *Postconditions:* `modular_integer(` *`arg`* `) == mod( integer(` *`arg`* `),` *`_stat_mod`* `)`.

29 *Complexity:* O(D(N))

2.22.12

`explicit modular_integer( const char *` *`arg`* `);`

30 *Effects:* Constructs an object of class `modular_integer`,
calling `integer(` *`arg`* `)`,
`modular_integer::normalize();`

31 *Postconditions:* `modular_integer(` *`arg`* `) == mod( integer(` *`arg`* `),` *`_stat_mod`* `)`.

32 *Complexity:* O(D(N)+M(N)log(N))

2.22.13

`explicit modular_integer( const char * arg, radix_type radix );`

33 *Effects:* Constructs an object of class `modular_integer`,
calling `integer( arg, radix )`,
`modular_integer::normalize();`

34 *Postconditions:*
`modular_integer( arg, radix ) == mod( integer( arg, radix ), _stat_mod )`.

35 *Complexity:* O(D(N)+M(N)log(N))

2.22.14

`explicit modular_integer( const std::string & arg );`

36 *Effects:* Constructs an object of class `modular_integer`,
calling `integer( arg )`,
`modular_integer::normalize();`

37 *Postconditions:* `modular_integer( arg ) == mod( integer( arg ), _stat_mod )`.

38 *Complexity:* O(D(N)+M(N)log(N))

2.22.15

`explicit modular_integer( const std::string & arg, radix_type radix );`

39 *Effects:* Constructs an object of class `modular_integer`,
calling `integer( arg, radix )`,
`modular_integer::normalize();`

40 *Postconditions:*
`modular_integer( arg, radix ) == mod( integer( arg, radix ), _stat_mod )`.

41 *Complexity:* O(D(N)+M(N)log(N))

2.22.16

`modular_integer( const integer & arg );`

42 *Effects:* Copy constructs an object of class `modular_integer`,
calling `integer( arg )`,
`modular_integer::normalize();`

43 *Postconditions:* `modular_integer( arg ) == mod( arg, _stat_mod )`.

44 *Complexity:* O(D(N))

2.22.17

`~modular_integer();`

45 *Effects:* Destructs an object of class `modular_integer`.

46 *Complexity:* O(1)

2.23 Modular Integer Member Functions and Operators

2.23.1 Rationale **[integer.modular.funsops]**

The class `modular_integer` is derived from class `integer`. The `modular_integer` member functions and operators are equivalent to the `integer` member functions and operators, except that when the result is not between zero and the static modulus, the result is reduced modulo the static modulus using the `mod` function (2.7.6), by calling the derived `normalize()`. The static modulus is set with the static function `set_modulus()`, and is given here the name *`_stat_mod`*.

2.23.2

`modular_integer * clone() const;`

1 *Returns:* `new modular_integer( *this );`

2 *Postconditions:* `*clone() == *this`.

3 *Complexity:* O(N)

2.23.3

`modular_integer & operator=( const integer & rhs );`

4 *Effects:*
```
integer::operator=( rhs );
modular_integer::normalize();
```

5 *Postconditions:* `*this == mod( rhs, _stat_mod )`.

6 *Returns:* `*this`.

7 *Complexity:* O(D(N))

2.23.4

`integer_allocator * get_allocator() const;`

8 *Returns:* `integer::get_allocator()`

9 *Complexity:* O(1)

2.23.5

`void normalize();`

23 *Effects:*

```
*this = mod( *this, _stat_mod );
```

24 *Complexity:* O(D(N))

2.23.6

`void swap( integer & arg );`

10 *Effects:*

```
if( get_allocator() != arg.get_allocator() )
{ modular_integer temp( arg );
  arg.operator=( *this );
  _swap( temp );
} else
{ _swap( arg );
  modular_integer::normalize();
  arg.normalize();
};
```

11 *Remarks:* `_swap()` swaps the pointers, the signs and the lengths.

12 *Complexity:* O(D(N))

2.23.7

`modular_integer & negate();`

13 *Effects:*

```
integer::negate();
modular_integer::normalize();
```

14 *Returns:* `*this`.

15 *Complexity:* O(N)

2.23.8

`modular_integer & abs();`

16 *Effects:*
```
integer::abs();
modular_integer::normalize();
```

17 *Returns:* `*this`.

18 *Complexity:* O(N)

2.23.9

`modular_integer & operator++();`

19 *Effects:*
```
integer::operator++();
modular_integer::normalize();
```

20 *Returns:* `*this`.

21 *Remarks:* unary prefix operator.

22 *Complexity:* O(N)

2.23.10

`modular_integer & operator--();`

23 *Effects:*
```
integer::operator--();
modular_integer::normalize();
```

24 *Returns:* `*this`.

25 *Remarks:* unary prefix operator.

26 *Complexity:* O(N)

2.23.11

`modular_integer & operator+=( const integer & rhs );`

27 *Effects:*
```
integer::operator+=( rhs );
modular_integer::normalize();
```

28 *Returns:* `*this`.

29 *Complexity:* O(D(N))

2.23.12

`modular_integer & operator-=( const integer & rhs );`

30 *Effects:*
```
integer::operator-=( rhs );
modular_integer::normalize();
```

31 *Returns:* `*this`.

32 *Complexity:* O(D(N))

2.23.13

`modular_integer & operator*=( const integer & rhs );`

33 *Effects:*
```
integer::operator*=( rhs );
modular_integer::normalize();
```

34 *Returns:* `*this`.

35 *Complexity:* O(D(N))

2.23.14

`modular_integer & operator/=( const integer & rhs );`

36 *Effects:*
```
integer::operator/=( rhs );
modular_integer::normalize();
```

37 *Returns:* `*this`.

38 *Complexity:* O(D(N))

39 *Notes:* This is not proper modular division, for which `invmod` (2.7.8) must be used [2,3].

2.23.15

`modular_integer & operator%=( const integer & rhs );`

40 *Effects:*
```
integer::operator%=( rhs );
modular_integer::normalize();
```

41 *Returns:* `*this`.

42 *Complexity:* O(D(N))

43 *Notes:* This is not proper modular division, for which `invmod` (2.7.8) must be used [2,3].

2.23.16

`modular_integer & operator<<=( size_type rhs );`

44 *Effects:*
```
integer::operator<<=( rhs );
modular_integer::normalize();
```

45 *Returns:* `*this`.

46 *Complexity:* O(D(N))

2.23.17

`modular_integer & operator>>=( size_type rhs );`

47 *Effects:*
```
integer::operator>>=( rhs );
```

48 *Returns:* `*this`.

49 *Complexity:* O(N)

50 *Notes:* As the absolute value can only become smaller and the sign cannot change, modular reduction is never needed.

2.23.18

`modular_integer & operator|=( const integer & rhs );`

51 *Effects:*
```
integer::operator|=( rhs );
modular_integer::normalize();
```

52 *Returns:* `*this`.

53 *Complexity:* O(D(N))

2.23.19

`modular_integer & operator&=( const integer & rhs );`

54 *Effects:*
```
integer::operator&=( rhs );
modular_integer::normalize();
```

55 *Returns:* `*this`.

56 *Complexity:* O(D(N))

2.23.20

`modular_integer & operator^=( const integer & rhs );`

57 *Effects:*
```
integer::operator^=( rhs );
modular_integer::normalize();
```

58 *Returns:* `*this`.

59 *Complexity:* O(D(N))

2.24 Modular Integer Static Functions

2.24.1 Rationale [integer.modular.static]

The class `modular_integer` is derived from class `integer`. The `modular_integer` constructors, member functions and operators are equivalent to the `integer` constructors, member functions and operators, except that when the result is not between zero and the static modulus, the result is reduced modulo the static modulus using the `mod` function (2.7.6). This static modulus, which is mentioned as *_stat_mod* above, is set with the static function `set_modulus()`. The static modulus can be accessed with the static function `modulus()`.

2.24.2

`static void set_modulus( const integer & arg );`

1 *Effects:* `_stat_mod = arg`.

2 *Remarks:* This function must be called before objects of this class are constructed.

3 *Complexity:* O(N)

4 *Notes:* When the static modulus is not set, its value is zero, and the class `modular_integer` behaves as class `integer`. When the static modulus is not zero, as the sign of `x mod y` is the sign of `y` or zero, the sign of the modular integers is always the sign of `arg` or zero.

2.24.3

`static const integer & modulus();`

5 *Returns:* `_stat_mod`.

6 *Complexity:* O(N)

7 *Notes:* The return value is `const`, because changing the modulus for existing objects is undefined.

2.25 Allocated Integer Constructors and Destructor

2.25.1 Rationale [integer.allocated.ctors]

The class `allocated_integer` is derived from class `integer`. The `allocated_integer` constructors and destructor are equivalent to the `integer` constructors and destructor, except that the static allocator is activated, the `integer` constructor and assigment is called, and the static allocator is deactivated. A pointer to the static allocator is set with the static function `set_allocator()`, and is given here the name `_stat_alloc`. An integer with value zero does not have memory allocated (1.4).

2.25.2

`allocated_integer();`

1 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`.

2 *Complexity:* O(1)

3 *Notes:* Here it is used that an integer with value zero does not have memory allocated (1.4).

2.25.3

`allocated_integer( int` *`arg`* `);`

4 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,

```
_stat_alloc->activate();
integer::operator=( integer( arg ) );
_stat_alloc->deactivate();
```

5 *Complexity:* $\mathrm{O}(N)$

2.25.4

`allocated_integer( unsigned int` *`arg`* `);`

6 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,

```
_stat_alloc->activate();
integer::operator=( integer( arg ) );
_stat_alloc->deactivate();
```

7 *Complexity:* $\mathrm{O}(N)$

2.25.5

`allocated_integer( long` *`arg`* `);`

8 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,

```
_stat_alloc->activate();
integer::operator=( integer( arg ) );
_stat_alloc->deactivate();
```

9 *Complexity:* $\mathrm{O}(N)$

2.25.6

`allocated_integer( unsigned long` *`arg`* `);`

10 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,
`_stat_alloc``->activate();`
`integer::operator=( integer(` *`arg`* `) );`
`_stat_alloc``->deactivate();`

11 *Complexity:* O(N)

2.25.7

`allocated_integer( long long` *`arg`* `);`

12 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,
`_stat_alloc``->activate();`
`integer::operator=( integer(` *`arg`* `) );`
`_stat_alloc``->deactivate();`

13 *Complexity:* O(N)

2.25.8

`allocated_integer( unsigned long long` *`arg`* `);`

14 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,
`_stat_alloc``->activate();`
`integer::operator=( integer(` *`arg`* `) );`
`_stat_alloc``->deactivate();`

15 *Complexity:* O(N)

2.25.9

`explicit allocated_integer( float` *`arg`* `);`

16 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,
`_stat_alloc``->activate();`

`integer::operator=( integer( arg ) );`
`_stat_alloc->deactivate();`

17 *Complexity:* O(N)

2.25.10

`explicit allocated_integer( double arg );`

18 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,
`_stat_alloc->activate();`
`integer::operator=( integer( arg ) );`
`_stat_alloc->deactivate();`

19 *Complexity:* O(N)

2.25.11

`explicit allocated_integer( long double arg );`

20 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,
`_stat_alloc->activate();`
`integer::operator=( integer( arg ) );`
`_stat_alloc->deactivate();`

21 *Complexity:* O(N)

2.25.12

`explicit allocated_integer( const char * arg );`

22 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,
`_stat_alloc->activate();`
`integer::operator=( integer( arg ) );`
`_stat_alloc->deactivate();`

23 *Complexity:* O(M(N)log(N))

2.25.13

`explicit allocated_integer( const char * arg, radix_type radix );`

24 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,
`_stat_alloc->activate();`
`integer::operator=( integer( arg, radix ) );`
`_stat_alloc->deactivate();`

25 *Complexity:* O(M(N)log(N))

2.25.14

`explicit allocated_integer( const std::string & arg );`

26 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,
`_stat_alloc->activate();`
`integer::operator=( integer( arg ) );`
`_stat_alloc->deactivate();`

27 *Complexity:* O(M(N)log(N))

2.25.15

`explicit allocated_integer( const std::string & arg, radix_type radix );`

28 *Effects:* Constructs an object of class `allocated_integer`,
calling `integer()`,
`_stat_alloc->activate();`
`integer::operator=( integer( arg, radix ) );`
`_stat_alloc->deactivate();`

29 *Complexity:* O(M(N)log(N))

2.25.16

`allocated_integer( const integer & arg );`

30 *Effects:* Copy constructs an object of class `allocated_integer`,
calling `integer()`,
`_stat_alloc->activate();`

```
integer::operator=( integer( arg ) );
_stat_alloc->deactivate();
```

31 *Complexity:* $O(N)$

2.25.17

```
~allocated_integer();
```

32 *Effects:* Destructs an object of class `allocated_integer`:

```
_stat_alloc->activate();
integer::operator=( integer() );
_stat_alloc->deactivate();
```

33 *Complexity:* $O(1)$

34 *Notes:* Here it is used that an integer with value zero does not have memory allocated (1.4).

2.26 Allocated Integer Member Functions and Operators

2.26.1 Rationale [integer.allocated.funsops]

The class `allocated_integer` is derived from class `integer`. The `allocated_integer` member functions and operators are equivalent to the `integer` member functions and operators, except that the static allocator is activated, the `integer` member function or operator is called, and the static allocator is deactivated. A pointer to the static allocator is set with the static function `set_allocator()`, and is given here the name *`_stat_alloc`*. An integer with value zero does not have memory allocated (1.4).

2.26.2

```
allocated_integer * clone() const;
```

1 *Returns:* `new allocated_integer( *this );`

2 *Postconditions:* `*clone() == *this`.

3 *Complexity:* $O(N)$

2.26.3

`allocated_integer & operator=( const integer & rhs );`

4 *Effects:*
```
_stat_alloc->activate();
integer::operator=( rhs );
_stat_alloc->deactivate();
```

5 *Returns:* `*this`.

6 *Complexity:* O(N)

2.26.4

`integer_allocator * get_allocator() const;`

7 *Returns:* `_stat_alloc`

8 *Complexity:* O(1)

2.26.5

`void normalize();`

9 *Effects:* None.

10 *Complexity:* O(1)

2.26.6

`void swap( integer & arg );`

11 *Effects:*
```
if( get_allocator() != arg.get_allocator() )
{ allocated_integer temp( arg );
  arg.operator=( *this );
  _swap( temp );
} else
{ _swap( arg );
  arg.normalize();
};
```

12 *Remarks:* `_swap()` swaps the pointers, the signs and the lengths.

13 *Complexity:* O(1) (O(N) when allocators differ, O(D(N)) when *arg* is modular)

2.26.7

`allocated_integer & negate();`

14 *Effects:*
`integer::negate();`

15 *Returns:* `*this`.

16 *Complexity:* O(1)

17 *Notes:* As this only affects the sign, memory allocation is never needed.

2.26.8

`allocated_integer & abs();`

18 *Effects:*
`integer::abs();`

19 *Returns:* `*this`.

20 *Complexity:* O(1)

21 *Notes:* As this only affects the sign, memory allocation is never needed.

2.26.9

`allocated_integer & operator++();`

22 *Effects:*
```
_stat_alloc->activate();
integer::operator++();
_stat_alloc->deactivate();
```

23 *Returns:* `*this`.

24 *Remarks:* unary prefix operator.

25 *Complexity:* O(1) amortized

2.26.10

`allocated_integer & operator--();`

26 *Effects:*
```
_stat_alloc->activate();
integer::operator--();
_stat_alloc->deactivate();
```

27 *Returns:* `*this`.

28 *Remarks:* unary prefix operator.

29 *Complexity:* O(1) amortized

2.26.11

`allocated_integer & operator+=( const integer & rhs );`

30 *Effects:*
```
_stat_alloc->activate();
integer::operator+=( rhs );
_stat_alloc->deactivate();
```

31 *Returns:* `*this`.

32 *Complexity:* O(N)

2.26.12

`allocated_integer & operator-=( const integer & rhs );`

33 *Effects:*
```
_stat_alloc->activate();
integer::operator-=( rhs );
_stat_alloc->deactivate();
```

34 *Returns:* `*this`.

35 *Complexity:* O(N)

2.26.13

`allocated_integer & operator*=( const integer & rhs );`

36 *Effects:*
```
_stat_alloc->activate();
integer::operator*=( rhs );
_stat_alloc->deactivate();
```

37 *Returns:* `*this`.

38 *Complexity:* O(M(N))

2.26.14

`allocated_integer & operator/=( const integer & rhs );`

39 *Effects:*
```
_stat_alloc->activate();
integer::operator/=( rhs );
_stat_alloc->deactivate();
```

40 *Returns:* `*this`.

41 *Complexity:* O(D(N))

2.26.15

`allocated_integer & operator%=( const integer & rhs );`

42 *Effects:*
```
_stat_alloc->activate();
integer::operator%=( rhs );
_stat_alloc->deactivate();
```

43 *Returns:* `*this`.

44 *Complexity:* O(D(N))

2.26.16

`allocated_integer & operator<<=( size_type rhs );`

45 *Effects:*
```
_stat_alloc->activate();
integer::operator<<=( rhs );
_stat_alloc->deactivate();
```

46 *Returns:* `*this`.

47 *Complexity:* O(N)

2.26.17

`allocated_integer & operator>>=( size_type rhs );`

48 *Effects:*
```
_stat_alloc->activate();
integer::operator>>=( rhs );
_stat_alloc->deactivate();
```

49 *Returns:* `*this`.

50 *Complexity:* O(N)

2.26.18

`allocated_integer & operator|=( const integer & rhs );`

51 *Effects:*
```
_stat_alloc->activate();
integer::operator|=( rhs );
_stat_alloc->deactivate();
```

52 *Returns:* `*this`.

53 *Complexity:* O(N)

2.26.19

`allocated_integer & operator&=( const integer & rhs );`

54 *Effects:*
```
_stat_alloc->activate();
integer::operator&=( rhs );
_stat_alloc->deactivate();
```

55 *Returns:* `*this`.

56 *Complexity:* O(N)

2.26.20

`allocated_integer & operator^=( const integer & rhs );`

57 *Effects:*
```
_stat_alloc->activate();
integer::operator^=( rhs );
_stat_alloc->deactivate();
```

58 *Returns:* `*this`.

59 *Complexity:* O(N)

2.27 Allocated Integer Static Functions

2.27.1 Rationale [integer.allocated.static]

The class `allocated_integer` is derived from class `integer`. The `allocated_integer` constructors, destructor, member functions and operators are equivalent to the `integer` constructors, destructor, member functions and operators, except that the static allocator is activated, the `integer` constructor, member function or operator is called, and the static allocator is deactivated. A pointer to this static allocator, which is mentioned as *`_stat_alloc`* above, is set with the static function `set_allocator()` (1.6.3).

2.27.2

`static void set_allocator( integer_allocator * arg );`

1 *Effects:* `_stat_alloc = arg`.

2 *Remarks:* This function must be called before objects of this class are constructed.

3 *Complexity:* O(1)

4 *Notes:* When the static allocator is not set, its value is the default allocator, and the class `allocated_integer` behaves as class `integer`. The pointer is deleted by the class `allocated_integer` as if it were a static variable (1.6.3).

2.28 Allocated Unsigned Integer Constructors and Destructor

2.28.1 Rationale [integer.allocated.unsigned.ctors]

The class `allocated_unsigned_integer` is derived from class `unsigned_integer`. The `allocated_unsigned_integer` constructors and destructor are equivalent to the `unsigned_integer` constructors and destructor, except that the static allocator is activated, the `unsigned_integer` constructor and assigment is called, and the static allocator is deactivated. A pointer to the static allocator is set with the static function `set_allocator()`, and is given here the name `_stat_alloc`. An integer with value zero does not have memory allocated (1.4).

2.28.2

`allocated_unsigned_integer();`

1 *Effects:* Constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`.

2 *Complexity:* O(1)

3 *Notes:* Here it is used that an integer with value zero does not have memory allocated (1.4).

2.28.3

`allocated_unsigned_integer( int arg );`

4 *Effects:* Constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`,
`_stat_alloc->activate();`
`integer::operator=( unsigned_integer( arg ) );`
`_stat_alloc->deactivate();`

5 *Complexity:* O(N)

2.28.4

`allocated_unsigned_integer( unsigned int` *arg* `);`

6 *Effects:* Constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`,
_stat_alloc`->activate();`
`integer::operator=( unsigned_integer(` *arg* `) );`
_stat_alloc`->deactivate();`

7 *Complexity:* O(N)

2.28.5

`allocated_unsigned_integer( long` *arg* `);`

8 *Effects:* Constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`,
_stat_alloc`->activate();`
`integer::operator=( unsigned_integer(` *arg* `) );`
_stat_alloc`->deactivate();`

9 *Complexity:* O(N)

2.28.6

`allocated_unsigned_integer( unsigned long` *arg* `);`

10 *Effects:* Constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`,
_stat_alloc`->activate();`
`integer::operator=( unsigned_integer(` *arg* `) );`
_stat_alloc`->deactivate();`

11 *Complexity:* O(N)

2.28.7

`allocated_unsigned_integer( long long` *arg* `);`

12 *Effects:* Constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`,
_stat_alloc`->activate();`

`integer::operator=( unsigned_integer( `*`arg`*` ) );`
`_stat_alloc``->deactivate();`

13 *Complexity:* O(N)

2.28.8

`allocated_unsigned_integer( unsigned long long `*`arg`*` );`

14 *Effects:* Constructs an object of class `allocated_unsigned_integer`, calling `unsigned_integer()`,
`_stat_alloc``->activate();`
`integer::operator=( unsigned_integer( `*`arg`*` ) );`
`_stat_alloc``->deactivate();`

15 *Complexity:* O(N)

2.28.9

`explicit allocated_unsigned_integer( float `*`arg`*` );`

16 *Effects:* Constructs an object of class `allocated_unsigned_integer`, calling `unsigned_integer()`,
`_stat_alloc``->activate();`
`integer::operator=( unsigned_integer( `*`arg`*` ) );`
`_stat_alloc``->deactivate();`

17 *Complexity:* O(N)

2.28.10

`explicit allocated_unsigned_integer( double `*`arg`*` );`

18 *Effects:* Constructs an object of class `allocated_unsigned_integer`, calling `unsigned_integer()`,
`_stat_alloc``->activate();`
`integer::operator=( unsigned_integer( `*`arg`*` ) );`
`_stat_alloc``->deactivate();`

19 *Complexity:* O(N)

2.28.11

`explicit allocated_unsigned_integer( long double arg );`

20 *Effects:* Constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`,
`_stat_alloc->activate();`
`integer::operator=( unsigned_integer( arg ) );`
`_stat_alloc->deactivate();`

21 *Complexity:* O(N)

2.28.12

`explicit allocated_unsigned_integer( const char * arg );`

22 *Effects:* Constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`,
`_stat_alloc->activate();`
`integer::operator=( unsigned_integer( arg ) );`
`_stat_alloc->deactivate();`

23 *Complexity:* O(M(N)log(N))

2.28.13

`explicit allocated_unsigned_integer( const char * arg, radix_type radix );`

24 *Effects:* Constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`,
`_stat_alloc->activate();`
`integer::operator=( unsigned_integer( arg, radix ) );`
`_stat_alloc->deactivate();`

25 *Complexity:* O(M(N)log(N))

2.28.14

`explicit allocated_unsigned_integer( const std::string & arg );`

26 *Effects:* Constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`,
`_stat_alloc->activate();`

`integer::operator=( unsigned_integer( `*`arg`*` ) );`
`_stat_alloc``->deactivate();`

27 *Complexity:* O(M(N)log(N))

2.28.15

`explicit allocated_unsigned_integer( const std::string & `*`arg`*`, radix_type `*`radix`*` );`

28 *Effects:* Constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`,
`_stat_alloc``->activate();`
`integer::operator=( unsigned_integer( `*`arg`*`, `*`radix`*` ) );`
`_stat_alloc``->deactivate();`

29 *Complexity:* O(M(N)log(N))

2.28.16

`allocated_unsigned_integer( const integer & `*`arg`*` );`

30 *Effects:* Copy constructs an object of class `allocated_unsigned_integer`,
calling `unsigned_integer()`,
`_stat_alloc``->activate();`
`integer::operator=( unsigned_integer( `*`arg`*` ) );`
`_stat_alloc``->deactivate();`

31 *Complexity:* O(N)

2.28.17

`~allocated_unsigned_integer();`

32 *Effects:* Destructs an object of class `allocated_unsigned_integer`:
`_stat_alloc``->activate();`
`integer::operator=( integer() );`
`_stat_alloc``->deactivate();`

33 *Complexity:* O(1)

34 *Notes:* Here it is used that an integer with value zero does not have memory allocated (1.4).

2.29 Allocated Unsigned Integer Member Functions and Operators

2.29.1 Rationale [integer.allocated.unsigned.funsops]

The class `allocated_unsigned_integer` is derived from class `unsigned_integer`. The `allocated_unsigned_integer` member functions and operators are equivalent to the `unsigned_integer` member functions and operators, except that the static allocator is activated, the `unsigned_integer` member function or operator is called, and the static allocator is deactivated. A pointer to the static allocator is set with the static function `set_allocator()`, and is given here the name *`_stat_alloc`*. An integer with value zero does not have memory allocated (1.4).

2.29.2

```
allocated_unsigned_integer * clone() const;
```

1 *Returns:* `new allocated_unsigned_integer( *this );`

2 *Postconditions:* `*clone() == *this`.

3 *Complexity:* O(N)

2.29.3

```
allocated_unsigned_integer & operator=( const integer & rhs );
```

4 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator=( rhs );
_stat_alloc->deactivate();
```

5 *Returns:* `*this`.

6 *Complexity:* O(N)

2.29.4

```
integer_allocator * get_allocator() const;
```

7 *Returns:* *`_stat_alloc`*

8 *Complexity:* O(1)

2.29.5

`void normalize();`

9 *Effects:*
```
unsigned_integer::normalize();
```

10 *Complexity:* O(1)

2.29.6

`void swap( integer & arg );`

11 *Effects:*
```
if( get_allocator() != arg.get_allocator() )
{ allocated_unsigned_integer temp( arg );
  arg.operator=( *this );
  _swap( temp );
} else
{ _swap( arg );
  allocated_unsigned_integer::normalize();
  arg.normalize();
};
```

12 *Remarks:* `_swap()` swaps the pointers, the signs and the lengths.

13 *Complexity:* O(1) (O(N) when allocators differ, O(D(N)) when *arg* is modular)

2.29.7

`allocated_unsigned_integer & negate();`

14 *Effects:*
```
unsigned_integer::negate();
```

15 *Returns:* `*this`.

16 *Complexity:* O(1)

17 *Notes:* As this only affects the sign, memory allocation is never needed.

2.29.8

`allocated_unsigned_integer & abs();`

18 *Effects:*
```
unsigned_integer::abs();
```

19 *Returns:* `*this`.

20 *Complexity:* O(1)

21 *Notes:* As this only affects the sign, memory allocation is never needed.

2.29.9

`allocated_unsigned_integer & operator++();`

22 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator++();
_stat_alloc->deactivate();
```

23 *Returns:* `*this`.

24 *Remarks:* unary prefix operator.

25 *Complexity:* O(1) amortized

2.29.10

`allocated_unsigned_integer & operator--();`

26 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator--();
_stat_alloc->deactivate();
```

27 *Returns:* `*this`.

28 *Remarks:* unary prefix operator.

29 *Complexity:* O(1) amortized

2.29.11

`allocated_unsigned_integer & operator+=( const integer & rhs );`

30 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator+=( rhs );
_stat_alloc->deactivate();
```

31 *Returns:* `*this`.

32 *Complexity:* O(N)

2.29.12

`allocated_unsigned_integer & operator-=( const integer & rhs );`

33 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator-=( rhs );
_stat_alloc->deactivate();
```

34 *Returns:* `*this`.

35 *Complexity:* O(N)

2.29.13

`allocated_unsigned_integer & operator*=( const integer & rhs );`

36 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator*=( rhs );
_stat_alloc->deactivate();
```

37 *Returns:* `*this`.

38 *Complexity:* O(M(N))

2.29.14

`allocated_unsigned_integer & operator/=( const integer & rhs );`

39 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator/=( rhs );
_stat_alloc->deactivate();
```

40 *Returns:* `*this`.

41 *Complexity:* O(D(N))

2.29.15

`allocated_unsigned_integer & operator%=( const integer & rhs );`

42 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator%=( rhs );
_stat_alloc->deactivate();
```

43 *Returns:* `*this`.

44 *Complexity:* O(D(N))

2.29.16

`allocated_unsigned_integer & operator<<=( size_type rhs );`

45 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator<<=( rhs );
_stat_alloc->deactivate();
```

46 *Returns:* `*this`.

47 *Complexity:* O(N)

2.29.17

`allocated_unsigned_integer & operator>>=( size_type rhs );`

48 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator>>=( rhs );
_stat_alloc->deactivate();
```

49 *Returns:* `*this`.

50 *Complexity:* O(N)

2.29.18

`allocated_unsigned_integer & operator|=( const integer & rhs );`

51 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator|=( rhs );
_stat_alloc->deactivate();
```

52 *Returns:* `*this`.

53 *Complexity:* O(N)

2.29.19

`allocated_unsigned_integer & operator&=( const integer & rhs );`

54 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator&=( rhs );
_stat_alloc->deactivate();
```

55 *Returns:* `*this`.

56 *Complexity:* O(N)

2.29.20

`allocated_unsigned_integer & operator^=( const integer & rhs );`

57 *Effects:*
```
_stat_alloc->activate();
unsigned_integer::operator^=( rhs );
_stat_alloc->deactivate();
```

58 *Returns:* `*this`.

59 *Complexity:* O(N)

2.30 Allocated Unsigned Integer Static Functions

2.30.1 Rationale [integer.allocated.unsigned.static]

The class `allocated_unsigned_integer` is derived from class `unsigned_integer`. The `allocated_unsigned_integer` constructors, destructor, member functions and operators are equivalent to the `unsigned_integer` constructors, destructor, member functions and operators, except that the static allocator is activated, the `unsigned_integer` constructor, member function or operator is called, and the static allocator is deactivated. A pointer to this static allocator, which is mentioned as *_stat_alloc* above, is set with the static function `set_allocator()` (1.6.3).

2.30.2

`static void set_allocator( integer_allocator * arg );`

1 *Effects:* `_stat_alloc = arg`.

2 *Remarks:* This function must be called before objects of this class are constructed.

3 *Complexity:* O(1)

4 *Notes:* When the static allocator is not set, its value is the default allocator, and the class `allocated_unsigned_integer` behaves as class `unsigned_integer`. The pointer is deleted by the class `allocated_unsigned_integer` as if it were a static variable (1.6.3).

2.31 Allocated Modular Integer Constructors and Destructor

2.31.1 Rationale [integer.allocated.modular.ctors]

The class `allocated_modular_integer` is derived from class `modular_integer`. The `allocated_modular_integer` constructors and destructor are equivalent to the `modular_integer` constructors and destructor, except that the static allocator is activated, the `modular_integer` constructor and assigment is called, and the static allocator is deactivated. A pointer to the static allocator is set with the static function `set_allocator()`, and is given here the name *`_stat_alloc`*. An integer with value zero does not have memory allocated (1.4).

2.31.2

`allocated_modular_integer();`

1 *Effects:* Constructs an object of class `allocated_modular_integer`,
calling `modular_integer()`.

2 *Complexity:* O(1)

3 *Notes:* Here it is used that an integer with value zero does not have memory allocated (1.4).

2.31.3

`allocated_modular_integer( int` *`arg`* `);`

4 *Effects:* Constructs an object of class `allocated_modular_integer`,
calling `modular_integer()`,
`_stat_alloc``->activate();`
`integer::operator=( modular_integer(` *`arg`* `) );`
`_stat_alloc``->deactivate();`

5 *Complexity:* O(N)

2.31.4

`allocated_modular_integer( unsigned int` *`arg`* `);`

6 *Effects:* Constructs an object of class `allocated_modular_integer`,
calling `modular_integer()`,
`_stat_alloc``->activate();`
`integer::operator=( modular_integer(` *`arg`* `) );`
`_stat_alloc``->deactivate();`

7 *Complexity:* O(N)

2.31.5

`allocated_modular_integer( long` *`arg`* `);`

8 *Effects:* Constructs an object of class `allocated_modular_integer`, calling `modular_integer()`,
`_stat_alloc``->activate();`
`integer::operator=( modular_integer(` *`arg`* `) );`
`_stat_alloc``->deactivate();`

9 *Complexity:* O(N)

2.31.6

`allocated_modular_integer( unsigned long` *`arg`* `);`

10 *Effects:* Constructs an object of class `allocated_modular_integer`, calling `modular_integer()`,
`_stat_alloc``->activate();`
`integer::operator=( modular_integer(` *`arg`* `) );`
`_stat_alloc``->deactivate();`

11 *Complexity:* O(N)

2.31.7

`allocated_modular_integer( long long` *`arg`* `);`

12 *Effects:* Constructs an object of class `allocated_modular_integer`, calling `modular_integer()`,
`_stat_alloc``->activate();`
`integer::operator=( modular_integer(` *`arg`* `) );`
`_stat_alloc``->deactivate();`

13 *Complexity:* O(N)

2.31.8

`allocated_modular_integer( unsigned long long` *`arg`* `);`

14 *Effects:* Constructs an object of class `allocated_modular_integer`, calling `modular_integer()`,
`_stat_alloc->activate();`
`integer::operator=( modular_integer( arg ) );`
`_stat_alloc->deactivate();`

15 *Complexity:* O(N)

2.31.9

`explicit allocated_modular_integer( float arg );`

16 *Effects:* Constructs an object of class `allocated_modular_integer`, calling `modular_integer()`,
`_stat_alloc->activate();`
`integer::operator=( modular_integer( arg ) );`
`_stat_alloc->deactivate();`

17 *Complexity:* O(N)

2.31.10

`explicit allocated_modular_integer( double arg );`

18 *Effects:* Constructs an object of class `allocated_modular_integer`, calling `modular_integer()`,
`_stat_alloc->activate();`
`integer::operator=( modular_integer( arg ) );`
`_stat_alloc->deactivate();`

19 *Complexity:* O(N)

2.31.11

`explicit allocated_modular_integer( long double arg );`

20 *Effects:* Constructs an object of class `allocated_modular_integer`, calling `modular_integer()`,
`_stat_alloc->activate();`
`integer::operator=( modular_integer( arg ) );`
`_stat_alloc->deactivate();`

21 *Complexity:* O(N)

2.31.12

`explicit allocated_modular_integer( const char * arg );`

22 *Effects:* Constructs an object of class `allocated_modular_integer`,
calling `modular_integer()`,
`_stat_alloc->activate();`
`integer::operator=( modular_integer( arg ) );`
`_stat_alloc->deactivate();`

23 *Complexity:* O(M(N)log(N))

2.31.13

`explicit allocated_modular_integer( const char * arg, radix_type radix );`

24 *Effects:* Constructs an object of class `allocated_modular_integer`,
calling `modular_integer()`,
`_stat_alloc->activate();`
`integer::operator=( modular_integer( arg, radix ) );`
`_stat_alloc->deactivate();`

25 *Complexity:* O(M(N)log(N))

2.31.14

`explicit allocated_modular_integer( const std::string & arg );`

26 *Effects:* Constructs an object of class `allocated_modular_integer`,
calling `modular_integer()`,
`_stat_alloc->activate();`
`integer::operator=( modular_integer( arg ) );`
`_stat_alloc->deactivate();`

27 *Complexity:* O(M(N)log(N))

2.31.15

`explicit allocated_modular_integer( const std::string & arg, radix_type radix );`

28 *Effects:* Constructs an object of class `allocated_modular_integer`,
calling `modular_integer()`,
`_stat_alloc->activate();`

`integer::operator=( modular_integer(` *`arg`*`,` *`radix`* `) );`
`_stat_alloc``->deactivate();`

29 *Complexity:* O(M(N)log(N))

2.31.16

`allocated_modular_integer( const integer &` *`arg`* `);`

30 *Effects:* Copy constructs an object of class `allocated_modular_integer`,
calling `modular_integer()`,
`_stat_alloc``->activate();`
`integer::operator=( modular_integer(` *`arg`* `) );`
`_stat_alloc``->deactivate();`

31 *Complexity:* O(N)

2.31.17

`~allocated_modular_integer();`

32 *Effects:* Destructs an object of class `allocated_modular_integer`:
`_stat_alloc``->activate();`
`integer::operator=( integer() );`
`_stat_alloc``->deactivate();`

33 *Complexity:* O(1)

34 *Notes:* Here it is used that an integer with value zero does not have memory allocated (1.4).

2.32 Allocated Modular Integer Member Functions and Operators

2.32.1 Rationale [integer.allocated.modular.funsops]

The class `allocated_modular_integer` is derived from class `modular_integer`. The `allocated_modular_integer` member functions and operators are equivalent to the `modular_integer` member functions and operators, except that the static allocator is activated, the `modular_integer` member function or operator is called, and the static allocator is deactivated. A pointer to the static allocator is set with the static function `set_allocator()`, and is given here the name *`_stat_alloc`*. An integer with value zero does not have memory allocated (1.4).

2.32.2

`allocated_modular_integer * clone() const;`

1 *Returns:* `new allocated_modular_integer( *this );`

2 *Postconditions:* `*clone() == *this`.

3 *Complexity:* O(N)

2.32.3

`allocated_modular_integer & operator=( const integer &` *`rhs`* `);`

4 *Effects:*

```
_stat_alloc->activate();
modular_integer::operator=( rhs );
_stat_alloc->deactivate();
```

5 *Returns:* `*this`.

6 *Complexity:* O(N)

2.32.4

`integer_allocator * get_allocator() const;`

7 *Returns:* *`_stat_alloc`*

8 *Complexity:* O(1)

2.32.5

`void normalize();`

9 *Effects:*

```
_stat_alloc->activate();
modular_integer::normalize();
_stat_alloc->deactivate();
```

10 *Complexity:* O(D(N))

2.32.6

```
void swap( integer & arg );
```

11 *Effects:*
```
if( get_allocator() != arg.get_allocator() )
{ allocated_modular_integer temp( arg );
  arg.operator=( *this );
  _swap( temp );
} else
{ _swap( arg );
  allocated_modular_integer::normalize();
  arg.normalize();
};
```

12 *Remarks:* `_swap()` swaps the pointers, the signs and the lengths.

13 *Complexity:* O(D(N))

2.32.7

```
allocated_modular_integer & negate();
```

14 *Effects:*
```
_stat_alloc->activate();
modular_integer::negate();
_stat_alloc->deactivate();
```

15 *Returns:* `*this`.

16 *Complexity:* O(1)

2.32.8

```
allocated_modular_integer & abs();
```

17 *Effects:*
```
_stat_alloc->activate();
modular_integer::abs();
_stat_alloc->deactivate();
```

18 *Returns:* `*this`.

19 *Complexity:* O(1)

2.32.9

`allocated_modular_integer & operator++();`

20 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator++();
_stat_alloc->deactivate();
```

21 *Returns:* `*this`.

22 *Remarks:* unary prefix operator.

23 *Complexity:* O(1) amortized

2.32.10

`allocated_modular_integer & operator--();`

24 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator--();
_stat_alloc->deactivate();
```

25 *Returns:* `*this`.

26 *Remarks:* unary prefix operator.

27 *Complexity:* O(1) amortized

2.32.11

`allocated_modular_integer & operator+=( const integer & rhs );`

28 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator+=( rhs );
_stat_alloc->deactivate();
```

29 *Returns:* `*this`.

30 *Complexity:* O(N)

2.32.12

`allocated_modular_integer & operator-=( const integer & rhs );`

31 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator-=( rhs );
_stat_alloc->deactivate();
```

32 *Returns:* `*this`.

33 *Complexity:* O(N)

2.32.13

`allocated_modular_integer & operator*=( const integer & rhs );`

34 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator*=( rhs );
_stat_alloc->deactivate();
```

35 *Returns:* `*this`.

36 *Complexity:* O(M(N))

2.32.14

`allocated_modular_integer & operator/=( const integer & rhs );`

37 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator/=( rhs );
_stat_alloc->deactivate();
```

38 *Returns:* `*this`.

39 *Complexity:* O(D(N))

2.32.15

`allocated_modular_integer & operator%=( const integer & rhs );`

40 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator%=( rhs );
_stat_alloc->deactivate();
```

41 *Returns:* `*this`.

42 *Complexity:* O(D(N))

2.32.16

`allocated_modular_integer & operator<<=( size_type rhs );`

43 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator<<=( rhs );
_stat_alloc->deactivate();
```

44 *Returns:* `*this`.

45 *Complexity:* O(N)

2.32.17

`allocated_modular_integer & operator>>=( size_type rhs );`

46 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator>>=( rhs );
_stat_alloc->deactivate();
```

47 *Returns:* `*this`.

48 *Complexity:* O(N)

2.32.18

`allocated_modular_integer & operator|=( const integer & rhs );`

49 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator|=( rhs );
_stat_alloc->deactivate();
```

50 *Returns:* `*this`.

51 *Complexity:* O(N)

2.32.19

`allocated_modular_integer & operator&=( const integer & rhs );`

52 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator&=( rhs );
_stat_alloc->deactivate();
```

53 *Returns:* `*this`.

54 *Complexity:* O(N)

2.32.20

`allocated_modular_integer & operator^=( const integer & rhs );`

55 *Effects:*
```
_stat_alloc->activate();
modular_integer::operator^=( rhs );
_stat_alloc->deactivate();
```

56 *Returns:* `*this`.

57 *Complexity:* O(N)

2.33 Allocated Modular Integer Static Functions

2.33.1 Rationale [integer.allocated.modular.static]

The class `allocated_modular_integer` is derived from class `modular_integer`. The `allocated_modular_integer` constructors, destructor, member functions and operators are equivalent to the `modular_integer` constructors, destructor, member functions and operators, except that the static allocator is activated, the `modular_integer` constructor, member function or operator is called, and the static allocator is deactivated. A pointer to this static allocator, which is mentioned as *`_stat_alloc`* above, is set with the static function `set_allocator()` (1.6.3).

2.33.2

```
static void set_allocator( integer_allocator * arg );
```

1 *Effects:* *`_stat_alloc`* `=` *`arg`*.

2 *Remarks:* This function must be called before objects of this class are constructed.

3 *Complexity:* O(1)

4 *Notes:* When the static allocator is not set, its value is the default allocator, and the class `allocated_modular_integer` behaves as class `modular_integer`. The pointer is deleted by the class `allocated_modular_integer` as if it were a static variable (1.6.3).

Chapter 3

References

[1] D.E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, third edition, Addison-Wesley (1998).

[2] J. von zur Gathen and J. Gerhard, Modern Computer Algebra, second edition, Cambridge University Press (2003).

[3] E. Bach and J. Shallit, Algorithmic Number Theory, Volume 1: Efficient Algorithms, MIT Press (1996).

[4] J.A. Buchmann, Introduction to Cryptography, Springer (2001).

[5] B. Stroustrup, The C++ Programming Language, third edition, Addison-Wesley (2000).

[6] S. D. Meyers, Effective C++, second edition, Addison-Wesley (1998).

[7] S. D. Meyers, More Effective C++, Addison-Wesley (1996).

[8] M. Welschenbach, Cryptography in C and C++, second edition, Apress (2005).

[9] `http://numbers.computation.free.fr/Constants/Algorithms/inverse.html`

[10] `http://www.swox.com/gmp`