Doc No: N2143=07-0003
Date: 2007-1-11
Reply to: M.J. Kronenburg
M.Kronenburg@inter.nl.net

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

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 [4,10].
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.

1.3 General Requirements [integer.requirements]

1. An object of class `integer` 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 class `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. This way the class `integer` 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 [2,3,4,10].

4. The user must have bit and element 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 and element access functions (2.6) are uniquely defined. The bit and element access functions retrieve and assign as if the integer is padded with zeros to infinity, and therefore their position argument is never out of range. Because of this, and because `integer` is not a container like `vector`, an iterator is not required.

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 or modular integer is a, works like a and is substitutable as a normal integer (1.5), the from `integer` derived classes `unsigned_integer` (1.7.1) and the `modular_integer` with a static modulus (1.7.2) are provided, and users can derive others. In these (possibly template) derived classes, certain member functions are overridden (possibly in terms of the base class ones), that is the class `integer` is a run-time polymorphic class [5]. The member operators are non-virtual, following the NVI design pattern [8]. This design ensures that any expression with mixed compile-time and run-time types derived from `integer` is possible (1.7).

7. For defining user-provided memory management functions, an abstract class `integer_allocator` is provided. A class derived from class `integer` can then be used with a static allocator object derived from `integer_allocator` (1.6), providing run-time polymorphism [5] between different allocator objects. The allocation functions may be serialized already, but if not, a mutex can be added to the static allocator object as a derived data member. More (template) allocated integer classes can share the same static allocator object. In this document the `allocated_integer` (1.7.3) 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. This requirement implies that the non-negative binary representation of the absolute value (see item 4) should be stored in a contiguous memory block, because then optionally an assembler module can be used for the basic arithmetic operations for substantial performance improvement.

9. The serialization of `integer` objects in a multithreaded environment can be done by using an `integer` data member in a class and adding a mutex to it. For serialization usually only a limited set of the arithmetic operations are used. In the `integer` class implementation, no static variables can be used that can change [8]. As the static allocator pointer must be assigned before an allocated integer object can be created, and the allocator object pointed to can only be deleted after all allocated integer objects are deleted, and thus remains constant during the lifetime of all allocated integer objects, this static variable does not violate this requirement.

1.4 Usage of Integer Classes [integer.usage]

```
class heap_allocator : public integer_allocator
{ private:
    HANDLE the_heap;
  public:
    heap_allocator()
    { the_heap = HeapCreate( 0, 0, 0 ); };
    ~heap_allocator()
    { HeapDestroy( the_heap ); };
    void * allocate( integer::size_type n )
    { return HeapAlloc( the_heap, 0, n ); };
    void * reallocate( void * p, integer::size_type n )
    { return HeapReAlloc( the_heap, 0, p, n ); };
    void deallocate( void * p )
    { HeapFree( the_heap, 0, p ); };
};

heap_allocator global_heap_a;           // create heap
heap_allocator global_heap_b;           // create second heap

int main()
{ modular_integer<1>::set_modulus( 16 );
  modular_integer<2>::set_modulus( 11 );
  allocated_integer<1>::set_allocator( & global_heap_a );
  allocated_integer<2>::set_allocator( & global_heap_b );
  integer x( 4 );
  modular_integer<1> z( 10 );
  allocated_integer<1> a( 3 );
  integer * p = new allocated_integer<2>( -7 );
  integer * q = new modular_integer<2>( 3 );
  z += x * a;                           // z becomes 6 = 22 mod 16
  x -= z;                               // x becomes -2
  a *= x;                               // a becomes -6
  *q += a;                              // *q becomes 8 = -3 mod 11
  a.swap( *p );                         // swaps a and *p by value
// ...
  delete p;
  delete q;
  modular_integer<1>::set_modulus( 0 ); // cleanup static memory
```

```
  modular_integer<2>::set_modulus( 0 ); // cleanup static memory
}
```

This mixing of different types of integer in expressions and run-time polymorphism is enabled by deriving the unsigned, modular and allocated integers from base class `integer`.
All static integer objects must be (default) initialized to zero, and all static non-zero integer objects must be assigned to zero before the end of the program, because the order of initialization and deletion of the static default allocator and the static integers is undefined as they reside in different translation units [basic.start.init]. This is also true for the static modulus and offset of any modular integer class.

1.5 Design Decisions [integer.design]

The infinite precision integer is represented as a C++ class with the name `integer`. This class encapsulates a pointer to a contiguous memory block holding the binary representation of the integer, the size of that memory block, and a sign. The internal representation is binary so that the arithmetic operations can be performed efficiently (in particular using an assembler module). The type of the pointer and the type of element access is `data_type`, which is defined as `unsigned int` [basic.fundamental]. Therefore the size of the memory block is aligned on `sizeof(data_type)` bytes. On some (possibly 64-bit) systems this alignment may be on two times that value for performance reasons, but then `data_type` and the class behaviour should remain identical. When the value of an object is zero, the pointer to memory, the size and the sign are zero. This means that the allocated memory of an integer object can be released by assigning zero to it.
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 size of the binary representations is automatically adjusted to hold the resulting integer value of the arithmetic operations as if the arguments of the arithmetic operations were padded with zero-bits to infinity. For the bit and element access functions this means that when a bit or element of a position larger than the integers size is retrieved, the result is zero, and when a non-zero bit or element of a position larger than the integers size is assigned, the integers size is extended to hold that bit or element. Therefore the bit or element position argument is never out of range. Because of this behaviour, the element access function `operator[]` returns a separate object of proxy class `reference`. Such a proxy class is also used for bit access of the `bitset` class [template.bitset]. This also means that the bitwise not, or `operator~`, is omitted in the interface, because the zero bits to infinity cannot be set to one.
An unsigned, modular or allocated integer is a, works like a and is substitutable as a normal integer. The question "can it have two?" [5] is in this case clearly answered with "no". Therefore the unsigned, modular and allocated integer classes are derived from base class `integer`. This

makes expressions like `a += b;` possible with `a` and `b` any type of integer, just like when `a` would be base type `unsigned int` and `b` base type `int`, or vice versa.
The set of unsigned integers is a subset of the integers, and the set of modular integers with some modulus is a subset of the integers. This makes the following argument important:
"Due to its characteristics, dynamic polymorphism in C++ is best at:
* Uniform manipulation based on superset/subset relationships: Different classes that hold a superset/subset (base/derived) relationship can be treated uniformly." [8].
Because the class `integer` is an arithmetic class, the performance hit of using run-time (dynamic) polymorphism, that is implicitly using a vtable, is negligible (1.9,[14]).
The basic mathematical functions are virtual, but the mathematical interface member operators are not virtual, using the NonVirtual Interface (NVI) design pattern [8]. Users always call the public non-virtual interface functions or operators, but in the implementation of the constructors, destructors and member functions, which is completely behind the interface, always the protected virtual member functions are called.
The unsigned, modular and allocated integers are thus designed as (possibly template) classes derived from class integer with some virtual member functions [class.virtual], providing runtime polymorphism [5]. Thus a pointer of type `integer *` may point to a normal, an unsigned, a modular or an allocated integer, which all behave like an integer (virtual functions) but with slightly different implementations. This may be compared with a driver, which is an abstract class interface with only virtual functions. The integer and the driver have in common that they are both very close to the hardware.
The virtual functions of classes derived from class integer augment the corresponding functions of the base class [14], for example: the unsigned subtraction is a normal subtraction followed by a check whether the result is negative.
Because these types are basic types, only basic language elements are used, and not other design patterns [9] like singletons, object factories and smart pointers, which should be used at a higher abstraction level.
The modular and allocated integer classes have a template non-type `int` parameter named `tag`. This way modular integers with different combinations of static modulus and static offset, and allocated integers with different (possibly shared) static allocators can be used.
The interface of the class `integer` and its derived types are specified in detail, because the user needs to know how the derived integer types exactly differ from the base integer type, and because they are implemented and used in performance critical environments.
In the internal representation, the sign and the non-negative binary representation of the absolute value are stored separately, so that the bit and element access functions (2.6) are uniquely defined.
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]. Therefore it is not necessary to add overloads of the member functions and operators for the arithmetic base types [over.match.best]. The extra memory allocation of

the converting constructor can be avoided by declaring such constants or variables as `integer` themselves.
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 (2.12).
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 `while( x )`, without leading to ambiguities in expressions [11]. On some (older) compilers this conversion operator may however lead to an ambiguity when using the boolean operators with arithmetic base types. This ambiguity may then be avoided by overloading the boolean operators for arithmetic base type `int`:

```
bool operator==( const integer & x, int y )
{ return x == integer( y ); };
```

Boolean comparisons against zero can sometimes be optimized with the `integer` member function `sign()` (2.6.2), for example with the boolean comparison `x.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 (2.13).
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, 2.12.11). 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.

1.6 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 an allocator object derived from `integer_allocator` is static. The allocated elements of the `integer_allocator` are always

bytes, so a template type parameter is not needed for this class.
The `allocated_integer` class is derived from class `integer`, and has a static function `set_allocator()` which takes as argument a pointer to `integer_allocator`, and which sets the static allocator pointer of that class (1.7.3). This way not only the type of allocator is provided, but a pointer to a static allocator object of a certain allocator type, thus enabling the use of more static allocator objects of the same type (see example, 1.4). This way also sharing static allocator objects between different allocated integer classes is possible. For all allocations, reallocations and deallocations in all integer member functions (2.8), the static allocator to be used is determined by calling the virtual `do_get_allocator()` member function (2.5.8), which returns the static allocator pointer. This way, run-time polymorphism [5] between different allocators is provided.
The `integer` member function `do_swap()` (2.5.12) should only swap the pointers to the allocated memory if the allocators of those pointers are identical, otherwise the allocated data should be swapped. For this also the `do_get_allocator()` member function (2.5.8) is used, comparing not the allocator types but the actual allocator objects. Classes derived from class `integer` with a static allocator must therefore also override and implement `do_get_allocator()` and `do_swap()`.
As classes derived from class `integer_allocator` are thus only used as static allocators, and not as non-static class data members, in the derived allocator constructor, memory management initialization can be done, such as creating a heap handle, and in the derived allocator destructor, memory management cleanup can be done. When the allocation functions used do not provide serialization, a mutex can be added as a derived data member to the allocator class derived from `integer_allocator`, providing serialization of memory access in a multithreaded environment. More than one static allocator objects of the same allocator type can be created, thus for example creating several heaps of the same type, and more allocated integer template classes may share the same static allocator object.

1.7 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 (possibly in terms of the base class ones), but leaving the non-virtual member functions and operators and the non-member functions and operators unchanged. Examples are an `unsigned_integer`, or a `modular_integer` with a static modulus, or an `allocated_integer` with a specific static allocator (see below). This way expressions with variables of mixed compile-time and run-time types are possible (1.4).
The `integer` member functions that can be overridden are protected and virtual, and in their overridden derived implementation, the base class versions can be used with the scope resolution operator [class.derived], in this case with `integer::`. The overridden derived virtual member functions thus augment the base virtual member functions [14]. The `integer` member functions

and operators that are non-virtual must not be redefined in a derived class.
The `integer` virtual member functions 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 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` (copy) constructor (2.3), and converting the object to the derived class.

3 It must override the virtual (copy) constructors `do_create()` and `do_clone()` (2.5.9) returning a pointer of the derived class and calling the derived (copy) constuctor

4 It must override `assign( const integer & )` (2.8.8), converting a (possibly temporary) object to the derived class.

5 It must override the swap function `do_swap()` (2.5.12).

6 It must override virtual member functions when they have another meaning for the derived class, possibly having as argument a reference to class `integer`, and preferably augmenting by calling the `integer::` functions in their implementations [14].

7 When it uses a static allocator (see below), it must override `do_get_allocator()` (2.5.8).

8 It must not redefine non-virtual member functions or operators

9 It must not redefine non-member functions or operators

The conversion to the derived class in the derived constructors and derived overridden virtual functions, 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 in the range defined by the static offset and modulus, if not reduce the object modulo the static modulus. In this context this operation is called normalization, and is provided by the virtual member function `do_normalize()` (2.5.11).
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 function

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.

1.7.1 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 derived overridden virtual member functions first call the corresponding `integer` constructor or member function. 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 `do_normalize()` (2.18.5).
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_integer`s that give a negative result throws an exception. As negation is subtraction from zero, negation of non-zero `unsigned_integer`s 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 overridden `assign()` 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).

1.7.2 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 in a range given by a static offset and a static modulus; the inclusive minimum $x_{\min}$ is equal to the `offset` and the inclusive maximum $x_{\max}$ is equal to `offset` + `modulus` − 1. Conversely the `offset` is equal to $x_{\min}$ and the `modulus` is equal to $x_{\max} - x_{\min} + 1$. The offset and modulus are specified instead of the minimum and maximum, because then repeated computation of the modulus from the minimum and maximum is not necessary. For a mathematical modulo class, also called residue class ring, `offset` is zero, which is its default value when it is not set. A non-zero `offset` may be used for a modular integer that strictly mimics a base type signed int.

The class `modular_integer` is identical to the class `integer`, except that all derived constructors and all the derived overridden virtual member functions first call the corresponding `integer` constructor or member function. When the result is not within the range given above, the result is reduced modulo the static modulus using the `mod` function (2.11.8), which is provided by the

derived member function `do_normalize()` (2.21.5), using the formula:
$x =$ `offset` $+ ((x -$ `offset`$)$ mod `modulus`$)$.
This way x always returns within the range given by the inclusive $x_{\min}$ and $x_{\max}$ above. When for example `modulus`$=2^n$ and `offset`$=0$, then `normalize()` means in this case: take only the first n bits of the result.
The class `modular_integer` is a template class with an template non-type `int` parameter named `tag`, so that more modular integer classes with different combinations of static modulus and static offset can be used. The user can also change the type of the modulus (and offset) by supplying a second template class type parameter named `modT`, for example for using a modular integer with only an allocated modulus, or an allocated modular integer with a differently allocated modulus. This template `modT` parameter defaults to `integer` for the modular integer, and to `allocated_integer` with the same `tag` for the allocated modular integer.
The static modulus and the static offset of the class `modular_integer` can be set with static functions (2.19), and must be set before objects of this class are constructed:
```
modular_integer<1>::set_modulus( 123 );
```
When the static modulus is not set, its value is zero, and the class `modular_integer` behaves as the class `integer`. After deletion of the modular integer objects it is wise to release the static memory by setting the static variables to zero:
```
modular_integer<1>::set_modulus( 0 );
```
As mentioned above, also a different `modT` type for the modulus can be used, as in:
```
typedef modular_integer<1,allocated_integer<3> > my_mod_class;
my_mod_class::set_modulus( 123 );
```
or vice versa:
```
typedef allocated_modular_integer<1,integer> my_mod_class;
my_mod_class::set_modulus( 123 );
```
This way maximum flexibility for using default or non-default allocated modular integers with default or non-default allocated static modulus is possible. For the allocated integer class with that tag used as static modulus type, the allocator must of course be set (and possibly shared with the modular integer class itself) before objects of that modular integer class can be used.
Mathematically equivalent expressions always yield identical results, although the temporary `integer` objects may have different values. The expression (assuming `offset==0`):
```
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 a modular integer with modulus 2^n and offset 0, 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 strictly 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. A signed base type `int` for which `numeric_limits` flag `is_modulo` is `true`, is in fact a modular integer with modulus 2^n and offset -2^{n-1}.

1.7.3 The class `allocated_integer` [integer.allocated]

A class derived from class integer that is specified in this document is the class `allocated_integer`.
All `allocated_integer` constructors, destructor and member functions are equivalent to the `integer` constructors, destructor and member functions, except that the static allocator pointer is used for allocations, reallocations and deallocations in the `integer::` member functions by calling the overridden derived `do_get_allocator()` (2.24.2). This means that all objects of class `allocated_integer` only have memory allocated, reallocated and freed with this allocator.

The class `allocated_integer` is a template class with an template non-type `int` parameter named `tag`, so that more allocated integer classes with different static allocator pointers can be used. The static allocator pointers of different allocated integer classes may point to the same static allocator object, thus sharing the same static allocator.

The static allocator pointer is set with the a static function (2.22), which must be called before objects of this class are constructed. This function takes a pointer parameter, which may be a pointer to a global allocator (1.4):
`allocated_integer<1>::set_allocator( & global_heap );`
or it may be shared, for example with the default allocator:
`allocated_integer<1>::set_allocator( integer::get_default_allocator() );`
When the static allocator pointer is not set, its value is zero, and then `do_get_allocator()` throws an exception. This static function does not change ownership of the pointer; the allocator object pointed to by the pointer must be constructed before allocated integer objects are created, and must be deleted after all allocated integer objects have been deleted. Therefore it is wise to make an allocator object global or static, and to release the memory of static allocated integer objects by assigning zero to them (1.4).

For completeness, also the derived classes `allocated_unsigned_integer` and `allocated_modular_integer` are provided.

The memory allocation, reallocation and deallocation functions used by the allocator object may be already serialized for usage in a multithreaded environment, but if not, serialization may be implemented by adding a mutex as a derived data member of the allocator class derived from `integer_allocator`.

1.8 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
M(N)	multiplication of two infinite precision integers
D(N)	division and/or remainder of two infinite precision integers
G(N)	greatest common divisor of two infinite precision integers

On most systems M(N)<D(N)<G(N). These basic time complexities depend on the choice of algorithm [1,2,3,4,10], but for large N in general M(N) and D(N) should be subquadratic, while G(N) may be quadratic.
The time complexity of division with Newton iteration is D(N) = O(M(N)) [2,12]. The time complexity of square root with Newton iteration is also O(M(N)) [12]. The time complexity of radix conversion from decimal to binary is O(M(N)log(N)) and from binary to decimal O(D(N)log(N)) [2].
The time complexities of the member functions `is_zero()`, `sign()`, `negate()` and `abs()` are only 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 [1,2,3,4,10].

1.9 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 [1,2,3,10]:

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 M(N). The squaring algorithm is somewhat faster than the multiplication algorithm. 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, linked in from a separate assembler module.

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 M(N).

5. Square root:
   For the square root, Newton iteration is used, which is order 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 of size 1 or 2, the base type `int` or `long long int` (sometimes called `__int64`) should be used, that is: not all base type variables should be replaced with `integer` variables.

8. Virtual functions:
   The cost of making functions virtual, that is implicitly using virtual table lookups [7], is in all cases negligible [14]. This is because the `integer` class is an arithmetic class, where most arithmetic operations require access to the binary data of an `integer` object, and some checks like for example on carry or zero result. This makes one vtable indirection relatively insignificant.

9. Memory allocations:
   The memory allocations of the default allocator can be replaced with special memory al-

location functions by deriving a class from class `integer_allocator` and using the class `allocated_integer`. A limited number of `integer` objects with limited values for example can be allocated in a memory pool. The class body itself however, where the memory pointer, the size and the sign reside, are allocated with the C++ default allocation functions, which are implementation defined. These body allocations can be avoided by avoiding declaring `integer` objects in inner loops, or even by declaring static (arrays of) `integer` objects. These static objects must however be explicitly assigned zero at the end of the program, so that static memory is released before the program exits (see 1.4).

10. Avoiding temporaries:
    When using a statement like:
    `x += 3;`
    or even with a varying base type `int i`:
    `x += i;`
    a temporary `integer` is constructed via the converting constructor [class.conv.ctor], which requires a memory allocation. In many cases it is therefore better to make the argument itself an `integer`:
    `const integer three( 3 );`
    or:
    `integer i;`
    which may save a memory allocation. The boolean expression:
    `if( x >= 0 )`
    may create a temporary with value zero, and therefore may be less efficient than:
    `if( x.sign() >= 0 )`
    and:
    `while( x != 0 )`
    may be less efficient than:
    `while( x )`
    For the same reason it may be useful to avoid binary operators:
    `x = a + b;`
    may be less efficient than:
    `x = a;`
    `x += b;`

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. 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.

4. The required complexities are provided where N is the number of decimals or bits, and where M(N) is the multiplication complexity, D(N) is the division complexity and G(N) is the greatest common divisor complexity (1.8).

2.2 Synopsis [integer.synopsis]

```
namespace std {
 namespace tr2 {

class integer_allocator;

class integer {
  static integer_allocator * the_default_allocator; // for exposition only
public:
```

```
// types:
  typedef unsigned int data_type;
  typedef std::size_t  size_type;
  typedef int          sign_type;
  typedef unsigned int radix_type;
// 2.3 constructors and destructor:
  integer();
  integer( int );
  integer( unsigned int );
  integer( long int );
  integer( unsigned long int );
  integer( long long int );
  integer( unsigned long long int );
  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 member functions (NVI design pattern):
  static integer_allocator * get_default_allocator();
  integer_allocator * get_allocator() const;
  integer * create() const;
  integer * clone() const;
  integer & normalize();
  integer & swap( integer & );
// 2.6 bit and element access member functions:
  sign_type sign() const;
  bool is_zero() const;
  bool is_odd() const;
  size_type highest_bit() const;
  size_type lowest_bit() const;
  bool get_bit( size_type ) const;
  integer & set_bit( size_type, bool );
  integer get_sub( size_type, size_type ) const;
```

```
  size_type size() const;
  class reference                                  // proxy for element access
  { friend class integer;                          // see also [template.bitset]
      reference();
      reference( integer *, size_type );
    public:
      ~reference();
      operator data_type() const;                  // for x = a[i];
      reference & operator=( data_type );          // for a[i] = x;
      reference & operator=( const reference & ); // for a[i] = a[j];
  };
  reference operator[]( size_type );
  const reference operator[]( size_type ) const;
// 2.7 arithmetic member operators (NVI design pattern):
  integer & negate();
  integer & abs();
  integer & operator++();
  integer & operator--();
  integer & operator=( const integer & );
  integer & operator+=( const integer & );
  integer & operator-=( const integer & );
  integer & operator*=( const integer & );
  integer & operator/=( const integer & );
  integer & operator%=( const integer & );
  integer & operator<<=( size_type );
  integer & operator>>=( size_type );
  integer & operator&=( const integer & );
  integer & operator|=( const integer & );
  integer & operator^=( const integer & );
protected:
// 2.5 copying member functions:
  virtual integer_allocator * do_get_allocator() const;
  virtual integer * do_create() const;
  virtual integer * do_clone() const;
  virtual integer & do_normalize();
  virtual integer & do_swap( integer & );
// 2.8 arithmetic member functions:
  virtual data_type get_element( size_type );
  virtual integer & set_element( size_type, data_type );
  virtual integer & do_negate();
```

```
  virtual integer & do_abs();
  virtual integer & increment();
  virtual integer & decrement();
  virtual integer & assign( const integer & );
  virtual integer & add( const integer & );
  virtual integer & subtract( const integer & );
  virtual integer & multiply( const integer & );
  virtual integer & divide( const integer & );
  virtual integer & remainder( const integer & );
  virtual integer & shift_left( size_type );
  virtual integer & shift_right( size_type );
  virtual integer & bitwise_and( const integer & );
  virtual integer & bitwise_or( const integer & );
  virtual integer & bitwise_xor( const integer & );
};

// 2.9 arithmetic non-member operators:
integer operator++( integer &, int );
integer operator--( integer &, int );
integer operator+( const integer & );
integer operator-( const integer & );
integer operator+( const integer &, const integer & );
integer operator-( const integer &, const integer & );
integer operator*( const integer &, const integer & );
integer operator/( const integer &, const integer & );
integer operator%( const integer &, const integer & );
integer operator<<( const integer &, integer::size_type );
integer operator>>( const integer &, integer::size_type );
integer operator&( const integer &, const integer & );
integer operator|( const integer &, const integer & );
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 & );
// 2.11 arithmetic non-member functions:
integer abs( const integer & );
```

```
integer sqr( const integer & );
integer sqrt( const integer & );
void sqrtrem( const integer &, integer &, integer & );
void divrem( const integer &, const integer &, integer &, integer & );
integer pow( const integer &, const integer & );
integer mod( const integer &, const integer & );
integer powmod( const integer &, const integer &, const integer & );
integer invmod( const integer &, const integer & );
integer gcd( const integer &, const integer & );
integer lcm( const integer &, const integer & );
integer extgcd( const integer &, const integer &, integer &, integer & );
void swap( integer &, integer & );
// 2.12 conversion non-member functions:
int to_int( const integer & );
unsigned int to_unsigned_int( const integer & );
long int to_long_int( const integer & );
unsigned long int to_unsigned_long_int( const integer & );
long long int to_long_long_int( const integer & );
unsigned long long int to_unsigned_long_long_int( const integer & );
float to_float( const integer & );
double to_double( const integer & );
long double to_long_double( const integer & );
std::string to_string( const integer & );
std::string to_string( const integer &, integer::radix_type );
// 2.13 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 & );
// 2.14 random non-member functions:
integer random( const integer &, const integer & );

class integer_allocator {
protected:
// 2.15 allocator constructors and destructor
   integer_allocator();
   virtual ~integer_allocator();
public:
```

```
// 2.16 allocator member functions
   virtual void * allocate( integer::size_type ) = 0;
   virtual void * reallocate( void *, integer::size_type ) = 0;
   virtual void deallocate( void * ) = 0;
};

class unsigned_integer : public integer {
public:
// 2.17 unsigned integer constructors and destructor
  unsigned_integer();
  unsigned_integer( int );
  unsigned_integer( unsigned int );
  unsigned_integer( long int );
  unsigned_integer( unsigned long int );
  unsigned_integer( long long int );
  unsigned_integer( unsigned long long int );
  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();
// 2.18 unsigned integer member functions
protected:
  integer_allocator * do_get_allocator() const;
  unsigned_integer * do_create() const;
  unsigned_integer * do_clone() const;
  unsigned_integer & do_normalize();
  unsigned_integer & do_swap( integer & );
  unsigned_integer & set_element( size_type, data_type );
  unsigned_integer & do_negate();
  unsigned_integer & do_abs();
  unsigned_integer & increment();
  unsigned_integer & decrement();
  unsigned_integer & assign( const integer & );
  unsigned_integer & add( const integer & );
  unsigned_integer & subtract( const integer & );
```

```
  unsigned_integer & multiply( const integer & );
  unsigned_integer & divide( const integer & );
  unsigned_integer & remainder( const integer & );
  unsigned_integer & shift_left( size_type );
  unsigned_integer & shift_right( size_type );
  unsigned_integer & bitwise_and( const integer & );
  unsigned_integer & bitwise_or( const integer & );
  unsigned_integer & bitwise_xor( const integer & );
};

template<int tag, class modT = integer>
class modular_integer : public integer {
private:
  static modT the_modulus, the_offset; // for exposition only
public:
// 2.19 modular integer static functions
  static void set_modulus( const integer & );
  static void set_offset( const integer & );
  static const modT & get_modulus();
  static const modT & get_offset();
// 2.20 modular integer constructors and destructor
  modular_integer();
  modular_integer( int );
  modular_integer( unsigned int );
  modular_integer( long int );
  modular_integer( unsigned long int );
  modular_integer( long long int );
  modular_integer( unsigned long long int );
  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();
// 2.21 modular integer member functions
protected:
  integer_allocator * do_get_allocator() const;
```

```
  modular_integer * do_create() const;
  modular_integer * do_clone() const;
  modular_integer & do_normalize();
  modular_integer & do_swap( integer & );
  modular_integer & set_element( size_type, data_type );
  modular_integer & do_negate();
  modular_integer & do_abs();
  modular_integer & increment();
  modular_integer & decrement();
  modular_integer & assign( const integer & );
  modular_integer & add( const integer & );
  modular_integer & subtract( const integer & );
  modular_integer & multiply( const integer & );
  modular_integer & divide( const integer & );
  modular_integer & remainder( const integer & );
  modular_integer & shift_left( size_type );
  modular_integer & shift_right( size_type );
  modular_integer & bitwise_and( const integer & );
  modular_integer & bitwise_or( const integer & );
  modular_integer & bitwise_xor( const integer & );
};

template<int tag>
class allocated_integer : public integer {
private:
  static integer_allocator * the_allocator; // for exposition only
public:
// 2.22 allocated integer static functions
  static void set_allocator( integer_allocator * );
// 2.23 allocated integer constructors and destructor
  allocated_integer();
  allocated_integer( int );
  allocated_integer( unsigned int );
  allocated_integer( long int );
  allocated_integer( unsigned long int );
  allocated_integer( long long int );
  allocated_integer( unsigned long long int );
  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();
// 2.24 allocated integer member functions
protected:
  integer_allocator * do_get_allocator() const;
  allocated_integer * do_create() const;
  allocated_integer * do_clone() const;
  allocated_integer & do_normalize();
  allocated_integer & do_swap( integer & );
  allocated_integer & set_element( size_type, data_type );
  allocated_integer & do_negate();
  allocated_integer & do_abs();
  allocated_integer & increment();
  allocated_integer & decrement();
  allocated_integer & assign( const integer & );
  allocated_integer & add( const integer & );
  allocated_integer & subtract( const integer & );
  allocated_integer & multiply( const integer & );
  allocated_integer & divide( const integer & );
  allocated_integer & remainder( const integer & );
  allocated_integer & shift_left( size_type );
  allocated_integer & shift_right( size_type );
  allocated_integer & bitwise_and( const integer & );
  allocated_integer & bitwise_or( const integer & );
  allocated_integer & bitwise_xor( const integer & );
};

template<int tag>
class allocated_unsigned_integer : public integer {
private:
  static integer_allocator * the_allocator; // for exposition only
public:
// 2.25 allocated unsigned integer static functions
  static void set_allocator( integer_allocator * );
// 2.26 allocated unsigned integer constructors and destructor
  allocated_unsigned_integer();
```

```
  allocated_unsigned_integer( int );
  allocated_unsigned_integer( unsigned int );
  allocated_unsigned_integer( long int );
  allocated_unsigned_integer( unsigned long int );
  allocated_unsigned_integer( long long int );
  allocated_unsigned_integer( unsigned long long int );
  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();
// 2.27 allocated unsigned integer member functions
protected:
  integer_allocator * do_get_allocator() const;
  allocated_unsigned_integer * do_create() const;
  allocated_unsigned_integer * do_clone() const;
  allocated_unsigned_integer & do_normalize();
  allocated_unsigned_integer & do_swap( integer & );
  allocated_unsigned_integer & set_element( size_type, data_type );
  allocated_unsigned_integer & do_negate();
  allocated_unsigned_integer & do_abs();
  allocated_unsigned_integer & increment();
  allocated_unsigned_integer & decrement();
  allocated_unsigned_integer & assign( const integer & );
  allocated_unsigned_integer & add( const integer & );
  allocated_unsigned_integer & subtract( const integer & );
  allocated_unsigned_integer & multiply( const integer & );
  allocated_unsigned_integer & divide( const integer & );
  allocated_unsigned_integer & remainder( const integer & );
  allocated_unsigned_integer & shift_left( size_type );
  allocated_unsigned_integer & shift_right( size_type );
  allocated_unsigned_integer & bitwise_and( const integer & );
  allocated_unsigned_integer & bitwise_or( const integer & );
  allocated_unsigned_integer & bitwise_xor( const integer & );
};
```

```
template<int tag, class modT = allocated_integer<tag> >
class allocated_modular_integer : public integer {
private:
  static modT the_modulus, the_offset;      // for exposition only
  static integer_allocator * the_allocator; // for exposition only
public:
// 2.28 allocated modular integer static functions
  static void set_modulus( const integer & );
  static void set_offset( const integer & );
  static const modT & get_modulus();
  static const modT & get_offset();
  static void set_allocator( integer_allocator * );
// 2.29 allocated modular integer constructors and destructor
  allocated_modular_integer();
  allocated_modular_integer( int );
  allocated_modular_integer( unsigned int );
  allocated_modular_integer( long int );
  allocated_modular_integer( unsigned long int );
  allocated_modular_integer( long long int );
  allocated_modular_integer( unsigned long long int );
  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();
// 2.30 allocated modular integer member functions
protected:
  integer_allocator * do_get_allocator() const;
  allocated_modular_integer * do_create() const;
  allocated_modular_integer * do_clone() const;
  allocated_modular_integer & do_normalize();
  allocated_modular_integer & do_swap( integer & );
  allocated_modular_integer & set_element( size_type, data_type );
  allocated_modular_integer & do_negate();
  allocated_modular_integer & do_abs();
  allocated_modular_integer & increment();
```

```
  allocated_modular_integer & decrement();
  allocated_modular_integer & assign( const integer & );
  allocated_modular_integer & add( const integer & );
  allocated_modular_integer & subtract( const integer & );
  allocated_modular_integer & multiply( const integer & );
  allocated_modular_integer & divide( const integer & );
  allocated_modular_integer & remainder( const integer & );
  allocated_modular_integer & shift_left( size_type );
  allocated_modular_integer & shift_right( size_type );
  allocated_modular_integer & bitwise_and( const integer & );
  allocated_modular_integer & bitwise_or( const integer & );
  allocated_modular_integer & bitwise_xor( const integer & );
};

} }
```

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]. Overloading the functions and operators 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 int arg );`

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

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

12 *Complexity:* O(N)

2.3.6

`integer( unsigned long int arg );`

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

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

15 *Complexity:* O(N)

2.3.7

`integer( long long int arg );`

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

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

18 *Complexity:* O(N)

2.3.8

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

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

20 *Postconditions:* `to_unsigned_long_long_int( 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, where the hexadecimal letters can be uppercase or lowercase; 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_argument_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_argument_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, where the hexadecimal letters can be uppercase or lowercase; 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_argument_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_argument_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 lead to ambiguities in expressions [11].

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 [11].

2.5 Copying Member Functions [integer.copy]

2.5.1 Rationale

The static member function `get_default_allocator()` can be used to share the default allocator with an `allocated_integer` template class. The member function `do_get_allocator()` is used by all member functions and operators for memory allocation, and is used by `do_swap()` to see if the pointers or the values must be swapped. The non-static non-virtual member functions are part of the NonVirtual Interface (NVI) design pattern [8]. This means that they are non-virtual, are inherited by any class derived from `integer`, and only call the corresponding virtual member function. Only the virtual functions must be overridden in a derived class.

2.5.2

```
static integer_allocator * get_default_allocator();
```

1 *Returns:* `the_default_allocator`.

2 *Complexity:* O(1)

2.5.3

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

3 *Returns:* `do_get_allocator();`

4 *Notes:* As all member functions call `do_get_allocator()` if necessary, in principle users never need to call this member function.

2.5.4

```
integer * create() const;
```

5 *Returns:* `do_create();`

6 *Notes:* This is the virtual constructor. An assertion on equality of `typeid` of the result and `*this` may be done here [8].

2.5.5

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

7 *Returns:* `do_clone();`

8 *Notes:* This is the virtual copy constructor. An assertion on equality of `typeid` of the result and `*this` may be done here [8].

2.5.6

```
integer & normalize();
```

9 *Returns:* `do_normalize()`

10 *Notes:* As all member functions call `do_normalize()` if necessary, in principle users never need to call this member function.

2.5.7

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

11 *Effects:* `do_swap( arg );`

12 *Returns:* `*this`.

2.5.8

`virtual integer_allocator * do_get_allocator() const;`

13 *Returns:* `the_default_allocator`.

14 *Complexity:* O(1)

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

2.5.9

`virtual integer * do_create() const;`

16 *Returns:* `new integer();`

17 *Postconditions:* `*create() == integer()`.

18 *Complexity:* O(1)

19 *Notes:* Derived classes must override this member function, calling the derived constructor.

2.5.10

`virtual integer * do_clone() const;`

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

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

22 *Complexity:* O(N)

23 *Notes:* Derived classes must override this member function, calling the derived copy constructor.

2.5.11

`virtual integer & do_normalize();`

24 *Effects:* None.

25 *Returns:* `*this`.

26 *Complexity:* O(1)

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

2.5.12

```
virtual integer & do_swap( integer & arg );
```

28 *Effects:*
```
if( do_get_allocator() != arg.do_get_allocator() )
{ integer temp( arg );
  arg.assign( *this );
  basic_swap( temp );
} else
{ basic_swap( arg );
  arg.do_normalize();
};
```

29 *Returns:* `*this`.

30 *Remarks:* `basic_swap()` swaps the pointers, the signs and the sizes.

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

32 *Notes:* Derived classes must override this member function.

2.6 Bit and Element Access Member Functions

2.6.1 Rationale [integer.access.mem.funs]

The member function `sign()` may be used for efficient comparisons against zero, like in `x.sign() >= 0`. The proxy class `reference` is used for the correct behaviour when the element position assigned or retrieved with `operator[]()` is larger than the integers current size (1.5). These non-virtual functions and operators must not be redefined in a derived class.

2.6.2

```
sign_type sign() const;
```

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

2 *Complexity:* O(1)

2.6.3

```
bool is_zero() const;
```

3 *Returns:* `*this == 0`

4 *Complexity:* O(1)

2.6.4

`bool is_odd() const;`

5 *Returns:* `get_bit(0)`

6 *Complexity:* O(1)

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

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

15 *Complexity:* O(1) amortized

2.6.7

`bool get_bit( size_type` *`bitpos`* `) const;`

16 *Returns:* `true` if bit number *`bitpos`* is set, and `false` when it is not set. When *`bitpos`*`>highest_bit()` the result is `false`. Therefore *`bitpos`* is never out of range.

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

18 *Complexity:* O(1)

19 *Notes:* This is implemented by obtaining the corresponding element by calling `get_element()`, and returning the corresponding bit in that element.

2.6.8

`integer & set_bit( size_type` *`bitpos`*`, bool` *`val`* `);`

20 *Effects:* If *`val`*`==true`, bit number *`bitpos`* is set, and if *`val`*`==false`, it is cleared. If *`val`*`==false` and *`bitpos`*`>highest_bit()`, nothing changes. If *`val`*`==true` and *`bitpos`*`>highest_bit()`, the size is extended to hold the set bit. Therefore *`bitpos`* is never out of range.

21 *Returns:* `*this;`

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

23 *Complexity:* O(1)

24 *Notes:* This is implemented by obtaining the corresponding element by calling `get_element()`, changing the corresponding bit in that element to *`val`*, and putting the element back by calling `set_element()`.

2.6.9

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

25 *Returns:* The sub integer with the *`nbits`* bits starting with bit number *`startbit`*. When *`startbit`*`>highest_bit()` or *`nbits`*`==0` the result is 0. Therefore *`startbit`* and *`nbits`* are never out of range.

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

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

2.6.10

`size_type size() const;`

28 *Returns:* the size of the value of the integer.

29 *Complexity:* O(1)

30 *Notes:* The size of an integer is the minimum number of contiguous `data_type` elements to hold its binary value. When the value is zero, the size is zero.

2.6.11

`reference::reference( integer * ref, size_type pos );`

31 *Effects:* Constructs an object of class `reference`:
`: the_ref( ref ), the_pos( pos ).`

2.6.12

`reference::operator data_type() const;`

32 *Returns:* `the_ref->get_element( the_pos );`

33 *Notes:* This is used for expressions like `x=a[i];`.

2.6.13

`reference & reference::operator=( data_type val );`

34 *Effects:* `the_ref->set_element( the_pos, val );`

35 *Returns:* `*this;`

36 *Notes:* This is used for expressions like `a[i]=x;`.

2.6.14

`reference & reference::operator=( const reference & arg );`

37 *Effects:*
```
data_type temp = arg;
*this = temp;
```

38 *Returns:* `*this;`

39 *Notes:* This is used for expressions like `a[i]=a[j];`.

2.6.15

`reference operator[]( size_type pos );`

40 *Returns:* `reference( this, pos );`

41 *Complexity:* O(1)

2.6.16

`const reference operator[]( size_type` *pos* `) const;`

42 *Returns:* `reference( this, pos );`

43 *Complexity:* O(1)

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

2.7.1 Rationale

These member functions and operators are part of the NonVirtual Interface (NVI) design pattern [8]. This means that they are non-virtual, are inherited by any class derived from `integer`, and only call the corresponding virtual member function. They must not be redefined in a derived class.

2.7.2

`integer & negate();`

1 *Effects:* `do_negate();`

2 *Returns:* `*this;`

2.7.3

`integer & abs();`

3 *Effects:* `do_abs();`

4 *Returns:* `*this;`

2.7.4

`integer & operator++();`

5 *Effects:* `increment();`

6 *Returns:* `*this;`

7 *Remarks:* unary prefix operator.

2.7.5

`integer & operator--();`

8 *Effects:* `decrement();`

9 *Returns:* `*this;`

10 *Remarks:* unary prefix operator.

2.7.6

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

33 *Effects:* `assign( rhs );`

34 *Returns:* `*this;`

2.7.7

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

11 *Effects:* `add( rhs );`

12 *Returns:* `*this;`

2.7.8

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

13 *Effects:* `subtract( rhs );`

14 *Returns:* `*this;`

2.7.9

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

15 *Effects:* `multiply( rhs );`

16 *Returns:* `*this;`

2.7.10

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

17 *Effects:* `divide( rhs );`

18 *Returns:* `*this;`

19 *Throws:* `division_by_zero_error` if *rhs*==0.

2.7.11

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

20 *Effects:* `remainder( rhs );`

21 *Returns:* `*this;`

22 *Throws:* `division_by_zero_error` if *rhs*==0.

2.7.12

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

23 *Effects:* `shift_left( rhs );`

24 *Returns:* `*this;`

25 *Notes:* `x <<= n` is equivalent to `x *=` 2^n.

2.7.13

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

26 *Effects:* `shift_right( rhs );`

27 *Returns:* `*this;`

28 *Notes:* `x >>= n` is equivalent to `x /=` 2^n.

2.7.14

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

29 *Effects:* `bitwise_and( rhs );`

30 *Returns:* `*this;`

2.7.15

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

31 *Effects:* `bitwise_or( rhs );`

32 *Returns:* `*this;`

2.7.16

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

33 *Effects:* `bitwise_xor( rhs );`

34 *Returns:* `*this;`

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

2.8.1 Rationale

These virtual member functions can be overridden in a derived class when they have another meaning, possibly augmenting by calling a base version of the same virtual function [14]. The functions that change the integer data use `do_get_allocator()` for all memory allocations, reallocations and deallocations.

2.8.2

```
virtual data_type get_element( size_type pos );
```

1 *Returns:* The element at position *pos* of `*this`. When *pos*`>=size()`, the return value is zero. Therefore *pos* is never out of range.

2 *Remarks:* The element numbering starts with zero. The result is independent of sign.

3 *Complexity:* O(1)

4 *Notes:* This member function is virtual, but normally does not need to be overridden. This member function is used for retrieving a bit or element by the `get_bit()`, `set_bit()` and `operator[]()` members.

2.8.3

`virtual integer & set_element( size_type` *pos*`, data_type` *val* `);`

5 *Effects:* Sets element at position *pos* of `*this` to *val*, overwriting the old value. When *val*`==0` and *pos*`>=size()`, nothing changes. When *val*`!=0` and *pos*`>=size()`, the size is extended for holding *val*. Therefore *pos* is never out of range. When *val*`==0` and *pos*`==size()-1`, the size is minimized for holding the new integer value.

6 *Returns:* `*this`.

7 *Remarks:* The element numbering starts with zero. The sign, when non-zero, is preserved, and the absolute value of the result is independent of sign.

8 *Complexity:* O(1)

9 *Notes:* This virtual member function is used for changing a bit or element by the `set_bit()` and `operator[]()` members.

2.8.4

`virtual integer & do_negate();`

10 *Effects:* Negates the sign of `*this` and stores the result in `*this`.

11 *Returns:* `*this`.

12 *Complexity:* O(1)

2.8.5

`virtual integer & do_abs();`

13 *Effects:* `if( sign() < 0 ) do_negate();`

14 *Returns:* `*this`.

15 *Complexity:* O(1)

2.8.6

`virtual integer & increment();`

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

17 *Returns:* `*this`.

18 *Complexity:* O(1) amortized

2.8.7

```
virtual integer & decrement();
```

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

20 *Returns:* `*this`.

21 *Complexity:* O(1) amortized

2.8.8

```
virtual integer & assign( const integer & rhs );
```

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

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

37 *Returns:* `*this`.

38 *Complexity:* O(N)

2.8.9

```
virtual integer & add( const integer & rhs );
```

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

23 *Returns:* `*this`.

24 *Complexity:* O(N)

2.8.10

```
virtual integer & subtract( const integer & rhs );
```

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

26 *Returns:* `*this`.

27 *Complexity:* O(N)

2.8.11

`virtual integer & multiply( const integer & rhs );`

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

29 *Returns:* `*this`.

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

2.8.12

`virtual integer & divide( const integer & rhs );`

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

32 *Returns:* `*this`.

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

34 *Notes:* *`rhs`*`!=0` because `operator/=` throws an error if *`rhs`*`==0`.

2.8.13

`virtual integer & remainder( const integer & rhs );`

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

36 *Returns:* `*this`.

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

38 *Notes:* *`rhs`*`!=0` because `operator%=` throws an error if *`rhs`*`==0`.

2.8.14

`virtual integer & shift_left( size_type rhs );`

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

40 *Returns:* `*this`.

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

2.8.15

```
virtual integer & shift_right( size_type rhs );
```

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

43 *Returns:* `*this`.

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

45 *Complexity:* O(N)

2.8.16

```
virtual integer & bitwise_and( const integer & rhs );
```

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

47 *Returns:* `*this`.

48 *Complexity:* O(N)

49 *Notes:* The sign of the result is computed as if the sign was also a bit, which is one when negative and zero otherwise.

2.8.17

```
virtual integer & bitwise_or( const integer & rhs );
```

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

51 *Returns:* `*this`.

52 *Complexity:* O(N)

53 *Notes:* The sign of the result is computed as if the sign was also a bit, which is one when negative and zero otherwise.

2.8.18

```
virtual integer & bitwise_xor( const integer & rhs );
```

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

55 *Returns:* `*this`.

56 *Complexity:* O(N)

57 *Notes:* The sign of the result is computed as if the sign was also a bit, which is one when negative and zero otherwise.

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.

2.9.2

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

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

2 *Remarks:* unary postfix operator

3 *Notes:* The possibly derived overridden member increment function is called.

2.9.3

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

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

5 *Remarks:* unary postfix operator

6 *Notes:* The possibly derived overridden member decrement function is called.

2.9.4

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

7 *Returns:* `arg;`

8 *Remarks:* unary operator

2.9.5

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

9 *Returns:* `integer( arg ).negate();`

10 *Remarks:* unary operator

11 *Notes:* The `integer` member negation function is called.

2.9.6

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

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

13 *Notes:* The `integer` member addition function is called.

2.9.7

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

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

15 *Notes:* The `integer` member subtraction function is called.

2.9.8

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

16 *Returns:* `integer( lhs ) *= rhs;`.

17 *Notes:* The `integer` member multiplication function is called.

2.9.9

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

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

19 *Throws:* `division_by_zero_error` if *rhs* is zero.

20 *Notes:* The `integer` member division function is called.

2.9.10

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

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

22 *Throws:* `division_by_zero_error` if *rhs* is zero.

23 *Notes:* The `integer` member remainder function is called.

2.9.11

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

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

25 *Notes:* The `integer` member left shift function is called.

2.9.12

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

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

27 *Notes:* The `integer` member right shift function is called.

2.9.13

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

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

29 *Notes:* The `integer` member bitwise and function is called.

2.9.14

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

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

31 *Notes:* The `integer` member bitwise or function is called.

2.9.15

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

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

33 *Notes:* The `integer` member bitwise xor function 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 Arithmetic Non-Member Functions [integer.arith.nonmem.funs]

2.11.1 Rationale

The non-member function `swap()` calls the corresponding virtual member function. The other arithmetic non-member functions treat their arguments as `integer`.

2.11.2

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

1 *Returns:* `integer( arg ).abs();`

2 *Complexity:* O(N)

3 *Notes:* `x = abs(x);` is equivalent to `x.abs();`.

2.11.3

`integer sqr( const integer & arg );`

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

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

6 *Notes:* `x = sqr(x);` is equivalent to `x *= x;`.

2.11.4

`integer sqrt( const integer & arg );`

7 *Returns:* trunc($\sqrt{arg}$), where the trunc function truncates toward zero.

8 *Throws:* `invalid_argument_error` if *`arg`* `< 0`.

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

2.11.5

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

10 *Effects:* *`res`* `= sqrt(` *`arg`* `);` *`rem`* `=` *`arg`* `- sqr(` *`res`* `);`.

11 *Throws:* `invalid_argument_error` if *`arg`* `< 0`.

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

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

2.11.6

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

14 *Effects:* *`quot`* `=` *`lhs`* `/` *`rhs`*`;` *`rem`* `=` *`lhs`* `%` *`rhs`*`;`.

15 *Throws:* `division_by_zero_error` if *`rhs`* `== 0`.

16 *Complexity:* D(N) = O(M(N))

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

2.11.7

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

18 *Returns:* x^n.

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

20 *Complexity:* O(M(floor$(n/2)N_x$))

2.11.8

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

21 *Returns:* `y == 0 ? x : x` mod `y`, where $x \bmod y = x - y * \text{floor}(x/y)$, and where the floor function truncates downward, that is toward minus infinity.

22 *Complexity:* D(N) = O(M(N))

23 *Notes:* This is not always equal to the remainder, see 2.8.13.

2.11.9

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

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

25 *Throws:* `invalid_argument_error` if `n < 0`.

26 *Complexity:* O(log(n)M(N_y))

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

2.11.10

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

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

29 *Throws:* `invalid_argument_error` if `y <= 0`, and `division_by_zero_error` if `x == 0`.

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

31 *Notes:* the modular inverse exists when `gcd(x,y)==1`, that is when `x` and `y` are relatively prime [4].

2.11.11

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

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

33 *Complexity:* $\mathrm{G}(N) = \mathrm{O}(N^2)$

2.11.12

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

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

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

2.11.13

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

36 *Effects:* computes the greatest common divisor of `x` and `y`, and computes `a` and `b` such that `x * a + y * b == gcd( x, y )`.

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

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

2.11.14

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

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

40 *Complexity:* $\mathrm{O}(1)$ ($\mathrm{O}(N)$ when allocators differ, $\mathrm{O}(\mathrm{D}(N))$ when `x` or `y` is modular)

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

2.12.1 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.

2.12.2

`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)

2.12.3

`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)

2.12.4

`long int to_long_int( const integer & arg );`

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

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

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

12 *Complexity:* O(1)

2.12.5

`unsigned long int to_unsigned_long_int( const integer & arg );`

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

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

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

16 *Complexity:* O(1)

2.12.6

`long long int to_long_long_int( const integer & arg );`

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

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

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

20 *Complexity:* O(1)

2.12.7

`unsigned long long int to_unsigned_long_long_int( const integer & arg );`

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

22 *Postconditions:* `integer( to_unsigned_long_long_int( 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)

2.12.8

`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)

2.12.9

`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)

2.12.10

```
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)

2.12.11

```
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))

2.12.12

```
std::string to_string( const integer & arg, integer::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_argument_error` when not `2 <= radix <= 36`.

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

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

2.13.1 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.

2.13.2

```
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_argument_error` 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))

2.13.3

```
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))

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

2.14.1 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.

2.14.2

```
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` if *min* > *max*.

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

4 *Complexity:* O(N)

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

2.15.1 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` (1.6).

2.15.2

```
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.

2.15.3

```
virtual ~integer_allocator();
```

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

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

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

2.16.1 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` (1.6).

2.16.2

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

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

2 *Throws:* `allocation_error` when allocation fails.

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

2.16.3

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

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

5 *Throws:* `allocation_error` when reallocation fails.

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

2.16.4

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

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

8 *Throws:* `allocation_error` when deallocation fails.

2.17 Unsigned Integer Constructors and Destructor

2.17.1 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 overridden derived `do_normalize()` (2.18.5).

2.17.2

```
unsigned_integer();
```

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

2.17.3

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

2 *Effects:* Constructs an object of class `unsigned_integer`:
`: integer( arg )`,
`do_normalize();`

2.17.4

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

3 *Effects:* Constructs an object of class `unsigned_integer`:
`: integer(` *arg* `)`.

2.17.5

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

4 *Effects:* Constructs an object of class `unsigned_integer`:
`: integer(` *arg* `)`,
`do_normalize();`

2.17.6

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

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

2.17.7

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

6 *Effects:* Constructs an object of class `unsigned_integer`:
`: integer(` *arg* `)`,
`do_normalize();`

2.17.8

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

7 *Effects:* Constructs an object of class `unsigned_integer`:
`: integer(` *arg* `)`.

2.17.9

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

8 *Effects:* Constructs an object of class `unsigned_integer`:
`: integer(` *arg* `)`,
`do_normalize();`

2.17.10

```
explicit unsigned_integer( double arg );
```

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

2.17.11

```
explicit unsigned_integer( long double arg );
```

10 *Effects:* Constructs an object of class `unsigned_integer`:
```
: integer( arg ),
do_normalize();
```

2.17.12

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

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

2.17.13

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

12 *Effects:* Constructs an object of class `unsigned_integer`:
```
: integer( arg, radix ),
do_normalize();
```

2.17.14

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

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

2.17.15

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

14 *Effects:* Constructs an object of class `unsigned_integer`:
```
: integer( arg, radix ),
do_normalize();
```

2.17.16

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

15 *Effects:* Copy constructs an object of class `unsigned_integer`:
```
: integer( arg ),
do_normalize();
```

2.17.17

```
~unsigned_integer();
```

16 *Effects:* None.

2.18 Unsigned Integer Member Functions [integer.unsigned.funs]

2.18.1 Rationale

The class `unsigned_integer` is derived from class `integer`. The `unsigned_integer` arithmetic member functions are equivalent to the `integer` member functions, except that when the result is negative, an exception is thrown, by calling the overridden derived `do_normalize()` (2.18.5).

2.18.2

```
integer_allocator * do_get_allocator() const;
```

1 *Returns:* `integer::do_get_allocator();`

2.18.3

```
unsigned_integer * do_create() const;
```

2 *Returns:* `new unsigned_integer();`

3 *Postconditions:* `*do_create() == unsigned_integer()`.

2.18.4

`unsigned_integer * do_clone() const;`

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

5 *Postconditions:* `*do_clone() == *this`.

2.18.5

`unsigned_integer & do_normalize();`

39 *Effects:* None.

40 *Throws:* `unsigned_is_negative` if `sign() < 0`.

41 *Returns:* `*this`.

2.18.6

`unsigned_integer & do_swap( integer & arg );`

6 *Effects:*

```
if( do_get_allocator() != arg.do_get_allocator() )
{ unsigned_integer temp( arg );
  arg.assign( *this );
  basic_swap( temp );
} else
{ basic_swap( arg );
  do_normalize();
  arg.do_normalize();
};
```

7 *Returns:* `*this`.

8 *Remarks:* `basic_swap()` swaps the pointers, the signs and the sizes.

2.18.7

`unsigned_integer & set_element( size_type pos, data_type val );`

9 *Effects:*

```
integer::set_element( pos, val );
```

10 *Returns:* `*this`.

11 *Notes:* As the sign never becomes negative, `normalize()` does not need to be called.

2.18.8

```
unsigned_integer & do_negate();
```

12 *Effects:*
```
integer::do_negate();
do_normalize();
```

13 *Returns:* `*this`.

14 *Notes:* The end effect of this function is that if `*this != 0` an exception is thrown.

2.18.9

```
unsigned_integer & do_abs();
```

15 *Effects:* None.

16 *Returns:* `*this`.

2.18.10

```
unsigned_integer & increment();
```

17 *Effects:*
```
integer::increment();
```

18 *Returns:* `*this`.

19 *Notes:* As the sign never becomes negative, `normalize()` does not need to be called.

2.18.11

```
unsigned_integer & decrement();
```

20 *Effects:*
```
integer::decrement();
do_normalize();
```

21 *Returns:* `*this`.

2.18.12

```
unsigned_integer & assign( const integer & rhs );
```

22 *Effects:*
```
integer::assign( rhs );
do_normalize();
```

23 *Returns:* `*this`.

2.18.13

```
unsigned_integer & add( const integer & rhs );
```

24 *Effects:*
```
integer::add( rhs );
do_normalize();
```

25 *Returns:* `*this`.

2.18.14

```
unsigned_integer & subtract( const integer & rhs );
```

26 *Effects:*
```
integer::subtract( rhs );
do_normalize();
```

27 *Returns:* `*this`.

2.18.15

```
unsigned_integer & multiply( const integer & rhs );
```

28 *Effects:*
```
integer::multiply( rhs );
do_normalize();
```

29 *Returns:* `*this`.

2.18.16

```
unsigned_integer & divide( const integer & rhs );
```

30 *Effects:*
```
integer::divide( rhs );
do_normalize();
```

31 *Returns:* `*this`.

2.18.17

```
unsigned_integer & remainder( const integer & rhs );
```

32 *Effects:*
```
integer::remainder( rhs );
do_normalize();
```

33 *Returns:* `*this`.

2.18.18

```
unsigned_integer & shift_left( size_type rhs );
```

34 *Effects:*
```
integer::shift_left( rhs );
```

35 *Returns:* `*this`.

36 *Notes:* As the sign never becomes negative, `normalize()` does not need to be called.

2.18.19

```
unsigned_integer & shift_right( size_type rhs );
```

37 *Effects:*
```
integer::shift_right( rhs );
```

38 *Returns:* `*this`.

39 *Notes:* As the sign never becomes negative, `normalize()` does not need to be called.

2.18.20

```
unsigned_integer & bitwise_and( const integer & rhs );
```

40 *Effects:*
```
integer::bitwise_and( rhs );
do_normalize();
```

41 *Returns:* `*this`.

2.18.21

```
unsigned_integer & bitwise_or( const integer & rhs );
```

42 *Effects:*
```
integer::bitwise_or( rhs );
do_normalize();
```

43 *Returns:* `*this`.

2.18.22

```
unsigned_integer & bitwise_xor( const integer & rhs );
```

44 *Effects:*
```
integer::bitwise_xor( rhs );
do_normalize();
```

45 *Returns:* `*this`.

2.19 Modular Integer Static Functions [integer.modular.static]

2.19.1 Rationale

The template class `modular_integer` is derived from class `integer`. The `modular_integer` constructors and arithmetic member functions are equivalent to the `integer` constructors and member functions, except that when the result is not between its range given by the static modulus and the static offset, the result is reduced modulo the static modulus using the `mod` function (2.11.8), by calling the overridden derived `do_normalize()` (2.21.5). The static modulus and the static offset are `the_modulus` and `the_offset`. When for example `the_modulus`=2^n and `the_offset`=0, then `do_normalize()` means in this case: take only the first n bits of the result. The template parameter `tag` specifies a specific modulus and offset combination, which must be set by the user (1.4).
When `the_modulus` is zero, the template class `modular_integer` behaves as class `integer`.

2.19.2

```
template<int tag, class modT>
static void set_modulus( const integer & arg );
```

1 *Effects:* `the_modulus = arg;`.

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

3 *Notes:* When the modulus is not set, its value is zero, and the class `modular_integer` behaves as class `integer`.

2.19.3

```
template<int tag, class modT>
static void set_offset( const integer & arg );
```

4 *Effects:* `the_offset = arg;`.

5 *Remarks:* When called, this function must be called before objects of this class are constructed.

6 *Notes:* When the offset is not set, its value is zero.

2.19.4

```
template<int tag, class modT>
static const modT & get_modulus();
```

7 *Returns:* `the_modulus;`.

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

2.19.5

```
template<int tag, class modT>
static const modT & get_offset();
```

9 *Returns:* `the_offset;`.

10 *Notes:* The return value is `const` because changing the offset for existing objects is undefined.

2.20 Modular Integer Constructors and Destructor

2.20.1 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 its range given by the static modulus and the static offset, the result is reduced modulo the static modulus using the `mod` function (2.11.8), by calling the overridden derived `do_normalize()` (2.21.5). When for example `the_modulus`=2^n and `the_offset`=0, then `do_normalize()` means in this case: take only the first n bits of the result.

2.20.2

```
template<int tag, class modT>
modular_integer();
```

1 *Effects:* Constructs an object of class `modular_integer`:
```
: integer(),
do_normalize();
```

2.20.3

```
template<int tag, class modT>
modular_integer( int arg );
```

2 *Effects:* Constructs an object of class `modular_integer`:
```
: integer( arg ),
do_normalize();
```

2.20.4

```
template<int tag, class modT>
modular_integer( unsigned int arg );
```

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

2.20.5

```
template<int tag, class modT>
modular_integer( long int arg );
```

4 *Effects:* Constructs an object of class `modular_integer`:
`: integer( arg ),`
`do_normalize();`

2.20.6

```
template<int tag, class modT>
modular_integer( unsigned long int arg );
```

5 *Effects:* Constructs an object of class `modular_integer`:
`: integer( arg ),`
`do_normalize();`

2.20.7

```
template<int tag, class modT>
modular_integer( long long int arg );
```

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

2.20.8

```
template<int tag, class modT>
modular_integer( unsigned long long int arg );
```

7 *Effects:* Constructs an object of class `modular_integer`:
`: integer( arg ),`
`do_normalize();`

2.20.9

```
template<int tag, class modT>
explicit modular_integer( float arg );
```

8 *Effects:* Constructs an object of class `modular_integer`:
`: integer( arg ),`
`do_normalize();`

2.20.10

```
template<int tag, class modT>
explicit modular_integer( double arg );
```

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

2.20.11

```
template<int tag, class modT>
explicit modular_integer( long double arg );
```

10 *Effects:* Constructs an object of class `modular_integer`:
```
: integer( arg ),
do_normalize();
```

2.20.12

```
template<int tag, class modT>
explicit modular_integer( const char * arg );
```

11 *Effects:* Constructs an object of class `modular_integer`:
```
: integer( arg ),
do_normalize();
```

2.20.13

```
template<int tag, class modT>
explicit modular_integer( const char * arg, radix_type radix );
```

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

2.20.14

```
template<int tag, class modT>
explicit modular_integer( const std::string & arg );
```

13 *Effects:* Constructs an object of class `modular_integer`:
```
: integer( arg ),
do_normalize();
```

2.20.15

```
template<int tag, class modT>
explicit modular_integer( const std::string & arg, radix_type radix );
```

14 *Effects:* Constructs an object of class `modular_integer`:
`: integer( arg, radix ),`
`do_normalize();`

2.20.16

```
template<int tag, class modT>
modular_integer( const integer & arg );
```

15 *Effects:* Copy constructs an object of class `modular_integer`:
`: integer( arg ),`
`do_normalize();`

2.20.17

```
template<int tag, class modT>
~modular_integer();
```

16 *Effects:* None.

2.21 Modular Integer Member Functions [integer.modular.funs]

2.21.1 Rationale

The class `modular_integer` is derived from class `integer`. The `modular_integer` arithmetic member functions are equivalent to the `integer` member functions, except that when the result is not between its range given by the static modulus and the static offset, the result is reduced modulo the static modulus using the `mod` function (2.11.8), by calling the overridden derived `do_normalize()` (2.21.5). When for example `the_modulus`=2^n and `the_offset`=0, then `do_normalize()` means in this case: take only the first n bits of the result.

2.21.2

```
template<int tag, class modT>
integer_allocator * do_get_allocator() const;
```

46 *Returns:* `integer::do_get_allocator();`

2.21.3

```
template<int tag, class modT>
modular_integer<tag,modT> * do_create() const;
```

1 *Returns:* `new modular_integer<tag,modT>();`

2 *Postconditions:* `*do_create() == modular_integer<tag,modT>()`.

2.21.4

```
template<int tag, class modT>
modular_integer<tag,modT> * do_clone() const;
```

3 *Returns:* `new modular_integer<tag,modT>( *this );`

4 *Postconditions:* `*do_clone() == *this`.

2.21.5

```
template<int tag, class modT>
modular_integer<tag,modT> & do_normalize();
```

42 *Effects:*
`*this = the_offset + mod( *this - the_offset, the_modulus );`

43 *Returns:* `*this`.

2.21.6

```
template<int tag, class modT>
modular_integer<tag,modT> & do_swap( integer & arg );
```

5 *Effects:*

```
if( do_get_allocator() != arg.do_get_allocator() )
{ modular_integer<tag,modT> temp( arg );
  arg.assign( *this );
  basic_swap( temp );
} else
{ basic_swap( arg );
  do_normalize();
  arg.do_normalize();
};
```

6 *Returns:* `*this`.

7 *Remarks:* `basic_swap()` swaps the pointers, the signs and the sizes.

2.21.7

```
modular_integer & set_element( size_type pos, data_type val );
```

8 *Effects:*
```
integer::set_element( pos, val );
do_normalize();
```

9 *Returns:* `*this`.

2.21.8

```
template<int tag, class modT>
modular_integer<tag,modT> & do_negate();
```

10 *Effects:*
```
integer::do_negate();
do_normalize();
```

11 *Returns:* `*this`.

2.21.9

```
template<int tag, class modT>
modular_integer<tag,modT> & do_abs();
```

12 *Effects:*
```
integer::do_abs();
do_normalize();
```

13 *Returns:* `*this`.

2.21.10

```
template<int tag, class modT>
modular_integer<tag,modT> & increment();
```

14 *Effects:*
```
integer::increment();
do_normalize();
```

15 *Returns:* `*this`.

2.21.11

```
template<int tag, class modT>
modular_integer<tag,modT> & decrement();
```

16 *Effects:*
```
integer::decrement();
do_normalize();
```

17 *Returns:* `*this`.

2.21.12

```
template<int tag, class modT>
modular_integer<tag,modT> & assign( const integer & rhs );
```

18 *Effects:*
```
integer::assign( rhs );
do_normalize();
```

19 *Returns:* `*this`.

2.21.13

```
template<int tag, class modT>
modular_integer<tag,modT> & add( const integer & rhs );
```

20 *Effects:*
```
integer::add( rhs );
do_normalize();
```

21 *Returns:* `*this`.

2.21.14

```
template<int tag, class modT>
modular_integer<tag,modT> & subtract( const integer & rhs );
```

22 *Effects:*
```
integer::subtract( rhs );
do_normalize();
```

23 *Returns:* `*this`.

2.21.15

```
template<int tag, class modT>
modular_integer<tag,modT> & multiply( const integer & rhs );
```

24 *Effects:*
```
integer::multiply( rhs );
do_normalize();
```

25 *Returns:* `*this`.

2.21.16

```
template<int tag, class modT>
modular_integer<tag,modT> & divide( const integer & rhs );
```

26 *Effects:*
```
integer::divide( rhs );
do_normalize();
```

27 *Returns:* `*this`.

28 *Notes:* This is not proper modular division, for which `invmod()` must be used (2.11.10) [2,3].

2.21.17

```
template<int tag, class modT>
modular_integer<tag,modT> & remainder( const integer & rhs );
```

29 *Effects:*
```
integer::remainder( rhs );
do_normalize();
```

30 *Returns:* `*this`.

31 *Notes:* This is not proper modular remainder, for which `invmod()` must be used (2.11.10) [2,3].

2.21.18

```
template<int tag, class modT>
modular_integer<tag,modT> & shift_left( size_type rhs );
```

32 *Effects:*
```
integer::shift_left( rhs );
do_normalize();
```

33 *Returns:* `*this`.

2.21.19

```
template<int tag, class modT>
modular_integer<tag,modT> & shift_right( size_type rhs );
```

34 *Effects:*
```
integer::shift_right( rhs );
do_normalize();
```

35 *Returns:* `*this`.

2.21.20

```
template<int tag, class modT>
modular_integer<tag,modT> & bitwise_and( const integer & rhs );
```

36 *Effects:*
```
integer::bitwise_and( rhs );
do_normalize();
```

37 *Returns:* `*this`.

2.21.21

```
template<int tag, class modT>
modular_integer<tag,modT> & bitwise_or( const integer & rhs );
```

38 *Effects:*
```
integer::bitwise_or( rhs );
do_normalize();
```

39 *Returns:* `*this`.

2.21.22

```
template<int tag, class modT>
modular_integer<tag,modT> & bitwise_xor( const integer & rhs );
```

40 *Effects:*
```
integer::bitwise_xor( rhs );
do_normalize();
```

41 *Returns:* `*this`.

2.22 Allocated Integer Static Functions [integer.allocated.static]

2.22.1 Rationale

The template class `allocated_integer` is derived from class `integer`. The `allocated_integer` constructors, destructor and member functions are equivalent to the `integer` constructors, destructor and member functions, except that the static allocator pointer is used for allocations in the `integer::` member functions by calling the overridden derived `do_get_allocator()` (2.24.2). The allocator is pointed to by static allocator pointer `the_allocator`. The template parameter `tag` specifies a specific allocator pointer, which must be set by the user (1.4). More than one allocated template classes may share the same static allocator object.

2.22.2

```
template<int tag>
static void set_allocator( integer_allocator * arg );
```

1 *Effects:* `the_allocator = arg;`

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

3 *Notes:* This function does not change ownership of the allocator object, and the allocator object can only be deleted after all allocated integer objects are deleted or made zero. When the allocator pointer is not set, its value is zero, and then the `do_get_allocator()` member function throws an error.

2.23 Allocated Integer Constructors and Destructor

2.23.1 Rationale [integer.allocated.ctors]

The template 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 pointer is used for allocations in the `integer::` assignment function by calling the overridden derived `do_get_allocator()` (2.24.2). This way only the end result of the constructor is allocated with the non-default allocator, and not the temporary results used during construction itself. Here it is used that an integer with value zero does not have memory allocated (1.5).

2.23.2

```
template<int tag>
allocated_integer();
```

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

2.23.3

```
template<int tag>
allocated_integer( int arg );
```

2 *Effects:* Constructs an object of class `allocated_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.23.4

```
template<int tag>
allocated_integer( unsigned int arg );
```

3 *Effects:* Constructs an object of class `allocated_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.23.5

```
template<int tag>
allocated_integer( long int arg );
```

4 *Effects:* Constructs an object of class `allocated_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.23.6

```
template<int tag>
allocated_integer( unsigned long int arg );
```

5 *Effects:* Constructs an object of class `allocated_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.23.7

```
template<int tag>
allocated_integer( long long int arg );
```

6 *Effects:* Constructs an object of class `allocated_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.23.8

```
template<int tag>
allocated_integer( unsigned long long int arg );
```

7 *Effects:* Constructs an object of class `allocated_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.23.9

```
template<int tag>
explicit allocated_integer( float arg );
```

8 *Effects:* Constructs an object of class `allocated_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.23.10

```
template<int tag>
explicit allocated_integer( double arg );
```

9 *Effects:* Constructs an object of class `allocated_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.23.11

```
template<int tag>
explicit allocated_integer( long double arg );
```

10 *Effects:* Constructs an object of class `allocated_integer`:

```
: integer(),
assign( integer( arg ) );
```

2.23.12

```
template<int tag>
explicit allocated_integer( const char * arg );
```

11 *Effects:* Constructs an object of class `allocated_integer`:

```
: integer(),
assign( integer( arg ) );
```

2.23.13

```
template<int tag>
explicit allocated_integer( const char * arg, radix_type radix );
```

12 *Effects:* Constructs an object of class `allocated_integer`:

```
: integer(),
assign( integer( arg, radix ) );
```

2.23.14

```
template<int tag>
explicit allocated_integer( const std::string & arg );
```

13 *Effects:* Constructs an object of class `allocated_integer`:

```
: integer(),
assign( integer( arg ) );
```

2.23.15

```
template<int tag>
explicit allocated_integer( const std::string & arg, radix_type radix );
```

14 *Effects:* Constructs an object of class `allocated_integer`:

```
: integer(),
assign( integer( arg, radix ) );
```

2.23.16

```
template<int tag>
allocated_integer( const integer & arg );
```

15 *Effects:* Copy constructs an object of class `allocated_integer`:
```
: integer(),
assign( arg );
```

2.23.17

```
template<int tag>
~allocated_integer();
```

16 *Effects:* Destructs an object of class `allocated_integer`:
```
assign( integer() );
```

17 *Notes:* As an integer with value zero has no memory allocated, this assignment releases the memory.

2.24 Allocated Integer Member Functions [integer.allocated.funs]

2.24.1 Rationale

The template class `allocated_integer` is derived from class `integer`. The `allocated_integer` member functions are equivalent to the `integer` member functions, except that the static allocator pointer is used for allocations in the `integer::` member functions by calling the overridden derived `do_get_allocator()` (2.24.2).

2.24.2

```
template<int tag>
integer_allocator * do_get_allocator() const;
```

1 *Returns:* `the_allocator;`

2 *Throws:* `no_allocator_error` if `the_allocator == 0`

2.24.3

```
template<int tag>
allocated_integer<tag> * do_create() const;
```

3 *Returns:* `new allocated_integer<tag>();`

4 *Postconditions:* `*do_create() == allocated_integer<tag>()`.

2.24.4

```
template<int tag>
allocated_integer<tag> * do_clone() const;
```

5 *Returns:* `new allocated_integer<tag>( *this );`

6 *Postconditions:* `*do_clone() == *this`.

2.24.5

```
template<int tag>
allocated_integer & do_normalize();
```

7 *Effects:* None.

8 *Returns:* `*this`.

2.24.6

```
template<int tag>
allocated_integer<tag> & do_swap( integer & arg );
```

9 *Effects:*

```
if( do_get_allocator() != arg.do_get_allocator() )
{ allocated_integer<tag> temp( arg );
  arg.assign( *this );
  basic_swap( temp );
} else
{ basic_swap( arg );
  arg.do_normalize();
};
```

10 *Returns:* `*this`.

11 *Remarks:* `basic_swap()` swaps the pointers, the signs and the sizes.

2.24.7

```
template<int tag>
allocated_integer & set_element( size_type pos, data_type val );
```

12 *Effects:*
```
integer::set_element( pos, val );
```

13 *Returns:* `*this`.

2.24.8

```
template<int tag>
allocated_integer & do_negate();
```

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

15 *Returns:* `*this`.

2.24.9

```
template<int tag>
allocated_integer & do_abs();
```

16 *Effects:*
```
integer::do_abs();
```

17 *Returns:* `*this`.

2.24.10

```
template<int tag>
allocated_integer & increment();
```

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

19 *Returns:* `*this`.

2.24.11

```
template<int tag>
allocated_integer & decrement();
```

20 *Effects:*
`integer::decrement();`

21 *Returns:* `*this`.

2.24.12

```
template<int tag>
allocated_integer & assign( const integer & rhs );
```

22 *Effects:*
`integer::assign( rhs );`

23 *Returns:* `*this`.

2.24.13

```
template<int tag>
allocated_integer & add( const integer & rhs );
```

24 *Effects:*
`integer::add( rhs );`

25 *Returns:* `*this`.

2.24.14

```
template<int tag>
allocated_integer & subtract( const integer & rhs );
```

26 *Effects:*
`integer::subtract( rhs );`

27 *Returns:* `*this`.

2.24.15

```
template<int tag>
allocated_integer & multiply( const integer & rhs );
```

28 *Effects:*
`integer::multiply( rhs );`

29 *Returns:* `*this`.

2.24.16

```
template<int tag>
allocated_integer & divide( const integer & rhs );
```

30 *Effects:*
`integer::divide( rhs );`

31 *Returns:* `*this`.

2.24.17

```
template<int tag>
allocated_integer & remainder( const integer & rhs );
```

32 *Effects:*
`integer::remainder( rhs );`

33 *Returns:* `*this`.

2.24.18

```
template<int tag>
allocated_integer & shift_left( size_type rhs );
```

34 *Effects:*
`integer::shift_left( rhs );`

35 *Returns:* `*this`.

2.24.19

```
template<int tag>
allocated_integer & shift_right( size_type rhs );
```

36 *Effects:*
integer::shift_right(*rhs*);

37 *Returns:* *this.

2.24.20

```
template<int tag>
allocated_integer & bitwise_and( const integer & rhs );
```

38 *Effects:*
integer::bitwise_and(*rhs*);

39 *Returns:* *this.

2.24.21

```
template<int tag>
allocated_integer & bitwise_or( const integer & rhs );
```

40 *Effects:*
integer::bitwise_or(*rhs*);

41 *Returns:* *this.

2.24.22

```
template<int tag>
allocated_integer & bitwise_xor( const integer & rhs );
```

42 *Effects:*
integer::bitwise_xor(*rhs*);

43 *Returns:* *this.

2.25 Allocated Unsigned Integer Static Functions

2.25.1 Rationale [integer.allocated.unsigned.static]

The template class `allocated_unsigned_integer` is derived from class `integer`. The `allocated_unsigned_integer` constructors, destructor and member functions are equivalent to the `integer` constructors, destructor and member functions, except that when the result is negative, an exception is thrown, by calling the overridden derived `do_normalize()` (2.27.5), and that the static allocator pointer is used for allocations in the `integer::` member functions by calling the overridden derived `do_get_allocator()` (2.27.2). The allocator is pointed to by static allocator pointer `the_allocator`. The template parameter `tag` specifies a specific allocator pointer, which must be set by the user (1.4). More than one allocated template classes may share the same static allocator object.

2.25.2

```
template<int tag>
static void set_allocator( integer_allocator * arg );
```

1 *Effects:* `the_allocator = arg;`

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

3 *Notes:* This function does not change ownership of the allocator object, and the allocator object can only be deleted after all allocated integer objects are deleted or made zero. When the allocator pointer is not set, its value is zero, and then the `do_get_allocator()` member function throws an error.

2.26 Allocated Unsigned Integer Constructors and Destructor

2.26.1 Rationale [integer.allocated.unsigned.ctors]

The template class `allocated_unsigned_integer` is derived from class `integer`. The `allocated_unsigned_integer` constructors and destructor are equivalent to the `integer` constructors and destructor, except that when the result is negative, an exception is thrown, by calling the overridden derived `do_normalize()` (2.27.5), and that the static allocator pointer is used for allocations in the `integer::` assignment function by calling the overridden derived `do_get_allocator()` (2.27.2). This way only the end result of the constructor is allocated with the non-default allocator, and not the temporary results used during construction itself. Here it is used that an integer with value zero does not have memory allocated (1.5).

2.26.2

```
template<int tag>
allocated_unsigned_integer();
```

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

2.26.3

```
template<int tag>
allocated_unsigned_integer( int arg );
```

2 *Effects:* Constructs an object of class `allocated_unsigned_integer`:
`: integer()`,
`assign( integer( arg ) );`

2.26.4

```
template<int tag>
allocated_unsigned_integer( unsigned int arg );
```

3 *Effects:* Constructs an object of class `allocated_unsigned_integer`:
`: integer()`,
`assign( integer( arg ) );`

2.26.5

```
template<int tag>
allocated_unsigned_integer( long int arg );
```

4 *Effects:* Constructs an object of class `allocated_unsigned_integer`:
`: integer()`,
`assign( integer( arg ) );`

2.26.6

```
template<int tag>
allocated_unsigned_integer( unsigned long int arg );
```

5 *Effects:* Constructs an object of class `allocated_unsigned_integer`:
`: integer()`,
`assign( integer( arg ) );`

2.26.7

```
template<int tag>
allocated_unsigned_integer( long long int arg );
```

6 *Effects:* Constructs an object of class `allocated_unsigned_integer`:

```
: integer(),
assign( integer( arg ) );
```

2.26.8

```
template<int tag>
allocated_unsigned_integer( unsigned long long int arg );
```

7 *Effects:* Constructs an object of class `allocated_unsigned_integer`:

```
: integer(),
assign( integer( arg ) );
```

2.26.9

```
template<int tag>
explicit allocated_unsigned_integer( float arg );
```

8 *Effects:* Constructs an object of class `allocated_unsigned_integer`:

```
: integer(),
assign( integer( arg ) );
```

2.26.10

```
template<int tag>
explicit allocated_unsigned_integer( double arg );
```

9 *Effects:* Constructs an object of class `allocated_unsigned_integer`:

```
: integer(),
assign( integer( arg ) );
```

2.26.11

```
template<int tag>
explicit allocated_unsigned_integer( long double arg );
```

10 *Effects:* Constructs an object of class `allocated_unsigned_integer`:

```
: integer(),
assign( integer( arg ) );
```

2.26.12

```
template<int tag>
explicit allocated_unsigned_integer( const char * arg );
```

11 *Effects:* Constructs an object of class `allocated_unsigned_integer`:

```
: integer(),
assign( integer( arg ) );
```

2.26.13

```
template<int tag>
explicit allocated_unsigned_integer( const char * arg, radix_type radix );
```

12 *Effects:* Constructs an object of class `allocated_unsigned_integer`:

```
: integer(),
assign( integer( arg, radix ) );
```

2.26.14

```
template<int tag>
explicit allocated_unsigned_integer( const std::string & arg );
```

13 *Effects:* Constructs an object of class `allocated_unsigned_integer`:

```
: integer(),
assign( integer( arg ) );
```

2.26.15

```
template<int tag>
explicit allocated_unsigned_integer( const std::string & arg, radix_type radix );
```

14 *Effects:* Constructs an object of class `allocated_unsigned_integer`:

```
: integer(),
assign( integer( arg, radix ) );
```

2.26.16

```
template<int tag>
allocated_unsigned_integer( const integer & arg );
```

15 *Effects:* Copy constructs an object of class `allocated_unsigned_integer`:

```
: integer(),
assign( arg );
```

2.26.17

```
template<int tag>
~allocated_unsigned_integer();
```

16 *Effects:* Destructs an object of class `allocated_unsigned_integer`:
`assign( integer() );`

17 *Notes:* As an integer with value zero has no memory allocated, this assignment releases the memory.

2.27 Allocated Unsigned Integer Member Functions

2.27.1 Rationale [integer.allocated.unsigned.funs]

The template class `allocated_unsigned_integer` is derived from class `integer`. The `allocated_unsigned_integer` member functions are equivalent to the `integer` member functions, except that when the result is negative, an exception is thrown, by calling the overridden derived `do_normalize()` (2.27.5), and that the static allocator pointer is used for allocations in the `integer::` member functions by calling the overridden derived `do_get_allocator()` (2.27.2).

2.27.2

```
template<int tag>
integer_allocator * do_get_allocator() const;
```

1 *Returns:* `the_allocator;`

2 *Throws:* `no_allocator_error` if `the_allocator == 0`

2.27.3

```
template<int tag>
allocated_unsigned_integer<tag> * do_create() const;
```

3 *Returns:* `new allocated_unsigned_integer<tag>();`

4 *Postconditions:* `*do_create() == allocated_unsigned_integer<tag>()`.

2.27.4

```
template<int tag>
allocated_unsigned_integer<tag> * do_clone() const;
```

5 *Returns:* `new allocated_unsigned_integer<tag>( *this );`

6 *Postconditions:* `*do_clone() == *this`.

2.27.5

```
template<int tag>
allocated_unsigned_integer & do_normalize();
```

7 *Effects:* None.

8 *Throws:* `unsigned_is_negative` if `sign() < 0`.

9 *Returns:* `*this`.

2.27.6

```
template<int tag>
allocated_unsigned_integer<tag> & do_swap( integer & arg );
```

10 *Effects:*

```
if( do_get_allocator() != arg.do_get_allocator() )
{ allocated_unsigned_integer<tag> temp( arg );
  arg.assign( *this );
  basic_swap( temp );
} else
{ basic_swap( arg );
  do_normalize();
  arg.do_normalize();
};
```

11 *Returns:* `*this`.

12 *Remarks:* `basic_swap()` swaps the pointers, the signs and the sizes.

2.27.7

```
template<int tag>
allocated_unsigned_integer & set_element( size_type pos, data_type val );
```

13 *Effects:*
```
integer::set_element( pos, val );
```

14 *Returns:* `*this`.

15 *Notes:* As the sign never becomes negative, `normalize()` does not need to be called.

2.27.8

```
template<int tag>
allocated_unsigned_integer & do_negate();
```

16 *Effects:*
```
integer::do_negate();
do_normalize();
```

17 *Returns:* `*this`.

18 *Notes:* The end effect of this function is that if `*this != 0` an exception is thrown.

2.27.9

```
template<int tag>
allocated_unsigned_integer & do_abs();
```

19 *Effects:* None.

20 *Returns:* `*this`.

2.27.10

```
template<int tag>
allocated_unsigned_integer & increment();
```

21 *Effects:*
```
integer::increment();
```

22 *Returns:* `*this`.

23 *Notes:* As the sign never becomes negative, `normalize()` does not need to be called.

2.27.11

```
template<int tag>
allocated_unsigned_integer & decrement();
```

24 *Effects:*
```
integer::decrement();
do_normalize();
```

25 *Returns:* `*this`.

2.27.12

```
template<int tag>
allocated_unsigned_integer & assign( const integer & rhs );
```

26 *Effects:*
```
integer::assign( rhs );
do_normalize();
```

27 *Returns:* `*this`.

2.27.13

```
template<int tag>
allocated_unsigned_integer & add( const integer & rhs );
```

28 *Effects:*
```
integer::add( rhs );
do_normalize();
```

29 *Returns:* `*this`.

2.27.14

```
template<int tag>
allocated_unsigned_integer & subtract( const integer & rhs );
```

30 *Effects:*
```
integer::subtract( rhs );
do_normalize();
```

31 *Returns:* `*this`.

2.27.15

```
template<int tag>
allocated_unsigned_integer & multiply( const integer & rhs );
```

32 *Effects:*
```
integer::multiply( rhs );
do_normalize();
```

33 *Returns:* `*this`.

2.27.16

```
template<int tag>
allocated_unsigned_integer & divide( const integer & rhs );
```

34 *Effects:*
```
integer::divide( rhs );
do_normalize();
```

35 *Returns:* `*this`.

2.27.17

```
template<int tag>
allocated_unsigned_integer & remainder( const integer & rhs );
```

36 *Effects:*
```
integer::remainder( rhs );
do_normalize();
```

37 *Returns:* `*this`.

2.27.18

```
template<int tag>
allocated_unsigned_integer & shift_left( size_type rhs );
```

38 *Effects:*
```
integer::shift_left( rhs );
```

39 *Returns:* `*this`.

40 *Notes:* As the sign never becomes negative, `normalize()` does not need to be called.

2.27.19

```
template<int tag>
allocated_unsigned_integer & shift_right( size_type rhs );
```

41 *Effects:*
```
integer::shift_right( rhs );
```

42 *Returns:* `*this`.

43 *Notes:* As the sign never becomes negative, `normalize()` does not need to be called.

2.27.20

```
template<int tag>
allocated_unsigned_integer & bitwise_and( const integer & rhs );
```

44 *Effects:*
```
integer::bitwise_and( rhs );
do_normalize();
```

45 *Returns:* `*this`.

2.27.21

```
template<int tag>
allocated_unsigned_integer & bitwise_or( const integer & rhs );
```

46 *Effects:*
```
integer::bitwise_or( rhs );
do_normalize();
```

47 *Returns:* `*this`.

2.27.22

```
template<int tag>
allocated_unsigned_integer & bitwise_xor( const integer & rhs );
```

48 *Effects:*
```
integer::bitwise_xor( rhs );
do_normalize();
```

49 *Returns:* `*this`.

2.28 Allocated Modular Integer Static Functions

2.28.1 Rationale [integer.allocated.modular.static]

The template class `allocated_modular_integer` is derived from class `integer`. The `allocated_modular_integer` constructors, destructor and member functions are equivalent to the `integer` constructors, destructor and member functions, except that when the result is not between its range given by the static modulus and the static offset, the result is reduced modulo the static modulus using the `mod` function (2.11.8), by calling the overridden derived `do_normalize()` (2.30.5), and that the static allocator pointer is used for allocations in the `integer::` member functions by calling the overridden derived `do_get_allocator()` (2.30.2). The static modulus and the static offset are `the_modulus` and `the_offset`. When for example `the_modulus`=2^n and `the_offset`=0, then `do_normalize()` means in this case: take only the first n bits of the result. The allocator is pointed to by static allocator pointer `the_allocator`. The template parameter `tag` specifies a specific combination of modulus, offset and allocator pointer, which must be set by the user (1.4). More than one allocated template classes may share the same static allocator object.
When `the_modulus` is zero, the template class `allocated_modular_integer` behaves as class `allocated_integer`.

2.28.2

```
template<int tag, class modT>
static void set_modulus( const integer & arg );
```

1 *Effects:* `the_modulus = arg;`.

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

3 *Notes:* When the modulus is not set, its value is zero, and the class `modular_integer` behaves as class `integer`.

2.28.3

```
template<int tag, class modT>
static void set_offset( const integer & arg );
```

4 *Effects:* `the_offset = arg;`.

5 *Remarks:* When called, this function must be called before objects of this class are constructed.

6 *Notes:* When the offset is not set, its value is zero.

2.28.4

```
template<int tag, class modT>
static const modT & get_modulus();
```

7 *Returns:* `the_modulus;`.

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

2.28.5

```
template<int tag, class modT>
static const modT & get_offset();
```

9 *Returns:* `the_offset;`.

10 *Notes:* The return value is `const` because changing the offset for existing objects is undefined.

2.28.6

```
template<int tag, class modT>
static void set_allocator( integer_allocator * arg );
```

11 *Effects:* `the_allocator = arg;`

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

13 *Notes:* This function does not change ownership of the allocator object, and the allocator object can only be deleted after all allocated integer objects are deleted or made zero. When the allocator pointer is not set, its value is zero, and then the `do_get_allocator()` member function throws an error.

2.29 Allocated Modular Integer Constructors and Destructor

2.29.1 Rationale [integer.allocated.modular.ctors]

The template class `allocated_modular_integer` is derived from class `integer`. The `allocated_modular_integer` constructors and destructor are equivalent to the `integer` constructors and destructor, except that when the result is not between its range given by the static modulus and the static offset, the result is reduced modulo the static modulus using the `mod` function (2.11.8), by calling the overridden derived `do_normalize()` (2.30.5), and that

the static allocator pointer is used for allocations in the `integer::` assignment function by calling the overridden derived `do_get_allocator()` (2.30.2). This way only the end result of the constructor is allocated with the non-default allocator, and not the temporary results used during construction itself. Here it is used that an integer with value zero does not have memory allocated (1.5).

2.29.2

```
template<int tag, class modT>
allocated_modular_integer();
```

1 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer() );
```

2.29.3

```
template<int tag, class modT>
allocated_modular_integer( int arg );
```

2 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.29.4

```
template<int tag, class modT>
allocated_modular_integer( unsigned int arg );
```

3 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.29.5

```
template<int tag, class modT>
allocated_modular_integer( long int arg );
```

4 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.29.6

```
template<int tag, class modT>
allocated_modular_integer( unsigned long int arg );
```

5 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.29.7

```
template<int tag, class modT>
allocated_modular_integer( long long int arg );
```

6 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.29.8

```
template<int tag, class modT>
allocated_modular_integer( unsigned long long int arg );
```

7 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.29.9

```
template<int tag, class modT>
explicit allocated_modular_integer( float arg );
```

8 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.29.10

```
template<int tag, class modT>
explicit allocated_modular_integer( double arg );
```

9 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.29.11

```
template<int tag, class modT>
explicit allocated_modular_integer( long double arg );
```

10 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.29.12

```
template<int tag, class modT>
explicit allocated_modular_integer( const char * arg );
```

11 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.29.13

```
template<int tag, class modT>
explicit allocated_modular_integer( const char * arg, radix_type radix );
```

12 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg, radix ) );
```

2.29.14

```
template<int tag, class modT>
explicit allocated_modular_integer( const std::string & arg );
```

13 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg ) );
```

2.29.15

```
template<int tag, class modT>
explicit allocated_modular_integer( const std::string & arg, radix_type radix );
```

14 *Effects:* Constructs an object of class `allocated_modular_integer`:
```
: integer(),
assign( integer( arg, radix ) );
```

2.29.16

```
template<int tag, class modT>
allocated_modular_integer( const integer & arg );
```

15 *Effects:* Copy constructs an object of class `allocated_modular_integer`:
`: integer(),`
`assign( arg );`

2.29.17

```
template<int tag, class modT>
~allocated_modular_integer();
```

16 *Effects:* Destructs an object of class `allocated_modular_integer`:
`integer::assign( integer() );`

17 *Notes:* As an integer with value zero has no memory allocated, this assignment releases the memory. The `integer::` function is used, because modular reduction with a non-zero offset may change zero into a non-zero value.

2.30 Allocated Modular Integer Member Functions

2.30.1 Rationale [integer.allocated.modular.funs]

The template class `allocated_modular_integer` is derived from class `integer`. The `allocated_modular_integer` member functions are equivalent to the `integer` member functions, except that when the result is not between its range given by the static modulus and the static offset, the result is reduced modulo the static modulus using the `mod` function (2.11.8), by calling the overridden derived `do_normalize()` (2.30.5), and that the static allocator pointer is used for allocations in the `integer::` member functions and operators by calling the overridden derived `do_get_allocator()` (2.30.2).

2.30.2

```
template<int tag, class modT>
integer_allocator * do_get_allocator() const;
```

1 *Returns:* `the_allocator;`

2 *Throws:* `no_allocator_error` if `the_allocator == 0`

2.30.3

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> * do_create() const;
```

3 *Returns:* `new allocated_modular_integer<tag,modT>();`

4 *Postconditions:* `*do_create() == allocated_modular_integer<tag,modT>()`.

2.30.4

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> * do_clone() const;
```

5 *Returns:* `new allocated_modular_integer<tag,modT>( *this );`

6 *Postconditions:* `*do_clone() == *this`.

2.30.5

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & do_normalize();
```

7 *Effects:*
`*this = the_offset + mod( *this - the_offset, the_modulus );`

8 *Returns:* `*this`.

9 *Remarks:* When evaluating this expression, temporary results must be of the same type as `*this`.

2.30.6

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & do_swap( integer & arg );
```

10 *Effects:*

```
if( do_get_allocator() != arg.do_get_allocator() )
{ allocated_modular_integer<tag,modT> temp( arg );
  arg.assign( *this );
  basic_swap( temp );
} else
{ basic_swap( arg );
  do_normalize();
  arg.do_normalize();
};
```

11 *Returns:* `*this`.

12 *Remarks:* `basic_swap()` swaps the pointers, the signs and the sizes.

2.30.7

```
allocated_modular_integer & set_element( size_type pos, data_type val );
```

13 *Effects:*
```
integer::set_element( pos, val );
do_normalize();
```

14 *Returns:* `*this`.

2.30.8

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & do_negate();
```

15 *Effects:*
```
integer::do_negate();
do_normalize();
```

16 *Returns:* `*this`.

2.30.9

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & do_abs();
```

17 *Effects:*
```
integer::do_abs();
do_normalize();
```

18 *Returns:* `*this`.

2.30.10

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & increment();
```

19 *Effects:*
```
integer::increment();
do_normalize();
```

20 *Returns:* `*this`.

2.30.11

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & decrement();
```

21 *Effects:*
```
integer::decrement();
do_normalize();
```

22 *Returns:* `*this`.

2.30.12

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & assign( const integer & rhs );
```

23 *Effects:*
```
integer::assign( rhs );
do_normalize();
```

24 *Returns:* `*this`.

2.30.13

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & add( const integer & rhs );
```

25 *Effects:*
```
integer::add( rhs );
do_normalize();
```

26 *Returns:* `*this`.

2.30.14

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & subtract( const integer & rhs );
```

27 *Effects:*
```
integer::subtract( rhs );
do_normalize();
```

28 *Returns:* `*this`.

2.30.15

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & multiply( const integer & rhs );
```

29 *Effects:*
```
integer::multiply( rhs );
do_normalize();
```

30 *Returns:* `*this`.

2.30.16

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & divide( const integer & rhs );
```

31 *Effects:*
```
integer::divide( rhs );
do_normalize();
```

32 *Returns:* `*this`.

33 *Notes:* This is not proper modular division, for which `invmod()` must be used (2.11.10) [2,3].

2.30.17

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & remainder( const integer & rhs );
```

34 *Effects:*
```
integer::remainder( rhs );
do_normalize();
```

35 *Returns:* `*this`.

36 *Notes:* This is not proper modular remainder, for which `invmod()` must be used (2.11.10) [2,3].

2.30.18

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & shift_left( size_type rhs );
```

37 *Effects:*
```
integer::shift_left( rhs );
do_normalize();
```

38 *Returns:* `*this`.

2.30.19

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & shift_right( size_type rhs );
```

39 *Effects:*
```
integer::shift_right( rhs );
do_normalize();
```

40 *Returns:* `*this`.

2.30.20

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & bitwise_and( const integer & rhs );
```

41 *Effects:*
```
integer::bitwise_and( rhs );
do_normalize();
```

42 *Returns:* `*this`.

2.30.21

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & bitwise_or( const integer & rhs );
```

43 *Effects:*
```
integer::bitwise_or( rhs );
do_normalize();
```

44 *Returns:* `*this`.

2.30.22

```
template<int tag, class modT>
allocated_modular_integer<tag,modT> & bitwise_xor( const integer & rhs );
```

45 *Effects:*
```
integer::bitwise_xor( rhs );
do_normalize();
```

46 *Returns:* `*this`.

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] H. Sutter and A. Alexandrescu, C++ Coding Standards, Addison-Wesley (2005).

[9] A. Alexandrescu, Modern C++ Design, Addison-Wesley (2001).

[10] M. Welschenbach, Cryptography in C and C++, second edition, Apress (2005).

[11] M. Wilson, Imperfect C++, Addison-Wesley (2005).

[12] http://numbers.computation.free.fr/Constants/Algorithms/inverse.html

[13] http://www.swox.com/gmp

[14] http://www.parashift.com/c++-faq-lite/virtual-functions.html