Introduction
This proposal aims to provide the standardese for N1692. It is meant to be self-contained, but implementers may still benefit from the background provided in N1692.
If included in TR1, this would fit best in chapter 5, Numerical facilities. It is an extension from the second category (“New library components (types and functions) that are declared in entirely new headers”). However, in that context the type `long long` cannot be assumed and must be replaced by `long`.
Proposed changes:

A. Add `<integer>` to 17.4.1.2/2 Table 11

B. Change 26/2 as follows:
The following subclauses describe components for complex number types, numeric \((n\text{ at a time})\) arrays, generalized numeric algorithms, unlimited range integers and facilities included from the ISO C library, as summarized in Table 79:

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.1 Requirements</td>
<td></td>
</tr>
<tr>
<td>26.2 Complex numbers</td>
<td><code>&lt;complex&gt;</code></td>
</tr>
<tr>
<td>26.3 Numeric arrays</td>
<td><code>&lt;valarray&gt;</code></td>
</tr>
<tr>
<td>26.4 Generalized numeric operations</td>
<td><code>&lt;numeric&gt;</code></td>
</tr>
<tr>
<td>26.5 Unlimited range integer</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td>26.5.6 C library</td>
<td><code>&lt;cmath&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
</tbody>
</table>

C. Add the following paragraph after 26.4 Generalized numeric operations:

26.5 Unlimited range integer

1. The header `<integer>` defines a single class and numerous functions for representing and manipulating integral numbers whose range is limited by available memory.

2. Notes: all integer operations which return an (temporary) integer and many member operators may fail due to insufficient memory. Such errors are reported as overflow_error exceptions (as available memory defines the range of the integer class).

26.5.1 Header `<integer>` synopsis

```c
#include <cstddef> // for size_t

namespace std {
  class integer;

  // 26.5.6 comparisons
  bool operator==( integer const&, integer const& );
  bool operator!=( integer const&, integer const& );
  bool operator>( integer const&, integer const& );
  bool operator>=( integer const&, integer const& );
  bool operator<( integer const&, integer const& );
  bool operator<=( integer const&, integer const& );
```
// 26.5.7 arithmetic operators
// unary
integer const& operator+( integer const& );
integer operator-( integer const& );

// binary
integer operator+( integer const&, integer const& );
integer operator+( integer const&, long long );
integer operator+( long long, integer const&);
integer operator-( integer const&, integer const& );
integer operator-( integer const&, long long );
integer operator-( long long, integer const&);
integer operator*( integer const&, integer const& );
integer operator*( integer const&, long long );
integer operator*( integer const&, double);
integer operator*( long long, integer const&);
integer operator*( double, integer const&);
integer operator/( integer const&, integer const& );
integer operator/( integer const&, long long );
integer operator/( long long, integer const& );
integer operator/( integer const&, double );
integer operator%( integer const&, integer const& );
integer operator%( integer const&, long long);

// 26.5.8 bitwise operators
integer operator&( integer const&, integer const& );
integer operator|( integer const&, integer const& );
integer operator^( integer const&, integer const& );
integer operator<< ( integer const&, size_t );
integer operator>> ( integer const&, size_t );

// 26.5.9 iostream operators
ostream & operator<<( ostream &, integer const& );
istream & operator>>( istream &, integer & );

// 26.5.10 functions
bool getbit( integer const&, size_t );
void setbit( integer &, size_t );
void clearbit( integer &, size_t );
size_t lowestbit( integer const& );
size_t highestbit( integer const& );
bool even( integer const& );
bool odd( integer const& );
int sign( integer const& );

integer gcd( integer const&, integer const& );
integer lcm( integer const&, integer const& );

integer abs( integer const& );
integer sqr( integer const& );
integer sqrt( integer const& );

integer factorial( long long );
integer pow( integer const&, long long );
}

26.5.2 class integer

namespace std {
    class integer {
private:
    std::vector<long long> // for exposition only
    signed char thesign; // for exposition only

class public:
    // 26.5.4 member functions
    integer( );
    integer( integer const& );
    integer( long long );
    integer( string const& );
    integer( char const * );

    integer& operator=( integer const& );
    integer& operator=( long long );
    integer& operator=( char const * );
    integer& operator=( string const& );

    operator long long( ) const;

    size_t hex_digits( ) const;

    // 26.5.4 arithmetic operators
integer& operator+=( integer const& );
integer& operator+=( long long );
integer& operator-=( integer const& );
integer& operator-=( long long );
integer& operator++( );
integer& operator--( );

integer& operator*=( integer const& );
integer& operator*=( long long );
integer& operator*=( double );
integer& operator/=( integer const& );
integer& operator/=( long long );
integer& operator/=( double );
integer& operator%=( integer const& );
integer& operator%=( long long );

// 26.5.5 bitwise operators
integer& operator&=( integer const& );
integer& operator|=( integer const& );
integer& operator^=( integer const& );
integer& operator<<=( size_t );
integer& operator>>=( size_t );

1. The class integer represents an integer number. [Note: this is not a fundamental type, and therefore not a signed integer type in the sense of 3.9.1 —end Note]

26.5.3 integer member functions

integer::integer ( );
1. Effects: creates an object of class integer
2. Postcondition: *this==0

integer::integer ( integer const& rhs);
3. Effects: copies an object of class integer
4. Postcondition: *this ==rhs
5. Complexity: O(N)

integer::integer ( long long rhs);
6. Effects: creates an object of class integer with initial value rhs
7. Complexity: O(1)
integer::integer (string const & rhs);

8. Requires: rhs contains only characters from “+-0123456789” (no whitespace allowed).
9. Effects: creates an object of class integer with initial value rhs
10. Throws: invalid_argument if rhs contains characters other than the 12 allowed
11. Complexity: O(N^2)

integer::integer (char const* rhs);

12. Effects: As if integer::integer (string const&) was called with argument
(string(rhs))

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

13. Effects: assigns the value of rhs to *this
14. Postcondition: *this==rhs
15. Returns: *this
16. Complexity: O(N^2)

integer& integer::operator=(long long rhs);

17. Effects: assigns the value of rhs to *this
18. Postcondition: *this==rhs
19. Returns: *this
20. Complexity: O(N)

integer& integer::operator=(string const & rhs);

21. Effects: as if calling operator=(integer(rhs))

integer& integer::operator=(char const* rhs);

22. Effects: as if calling operator=(integer(rhs))

integer::operator long long( ) const;

23. Returns: an object of type long long, with the value of *this
24. Throws: out_of_range if objects of type long long cannot hold the value of *this
25. Notes: the out_of_range (logic_error) exception can be prevented by comparing
*this to integer(LONGLONG_MAX).

size_t integer::hex_digits( ) const;

26. Returns: the number of hex digits in the representation, excluding any sign but possibly including
leading zeros.
27. Note: This is intended to roughly represent the amount of allocated storage, in 4 bits units (nibbles).

26.5.4 integer arithmetic member operators [lib.integer.member.arithops]

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

1. Effects: Increments the value of *this by the value of rhs
2. Returns: *this
3. Complexity: O(N)

    `integer& integer::operator+=( long long rhs);`

4. Effects: Increments the value of *this by the value of rhs
5. Returns: *this
6. Complexity: amortized O(1)

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

7. Effects: Decrements the value of *this by the value of rhs
8. Returns: *this
9. Complexity: O(N)

    `integer& integer::operator-=( long long rhs);`

10. Effects: Decrements the value of *this by the value of rhs
11. Returns: *this
12. Complexity: O(N)

    `integer& integer::operator++( );`

13. Effects: Increments the value of *this by 1
14. Returns: *this
15. Complexity: amortized O(1)

    `integer& integer::operator--( );`

16. Effects: Decrements the value of *this by 1
17. Returns: *this
18. Complexity: amortized O(1)

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

19. Effects: Multiplies the value of *this by the value of rhs
20. Returns: *this
21. Complexity: < O(N^2)

    `integer& integer::operator*=( long long rhs);`
    `integer& integer::operator*=( double rhs);`

22. Effects: Multiplies the value of *this by the value of rhs
23. Returns: *this
24. Complexity: O(N)

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

25. Effects: Divides the value of *this by the value of rhs
26. Complexity: O(N^2)
27. Throws: `invalid_argument` if rhs == 0

2005-01-13

Michiel.Salters@LogicaCMG.com
28. Returns: \*this

```cpp
integer& integer::operator/=( long long rhs);
integer& integer::operator/=( double rhs);
```

29. Effects: Divides the value of \*this by the value of rhs

30. Returns: \*this

31. Throws: invalid_argument if rhs==0

32. Complexity: O(N)

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

33. Effects: Sets \*this to the reminder of (\*this)/rhs

34. Returns: \*this

35. Complexity: O(N^2)

```cpp
integer& integer::operator%=( long long rhs);
```

36. Effects: Sets \*this to the reminder of (\*this)/rhs

37. Returns: \*this

38. Complexity: O(N)

### 26.5.5 integer bitwise member operators

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

1. Effects: Clears all bits of the absolute value in \*this for which the corresponding bit in rhs is clear. Other bits and the sign of \*this are unchanged.

2. Note: if the representation of rhs uses fewer bits than \*this, all additional bits are considered clear i.e. rhs is logically padded with leading zeroes.

3. Returns: \*this

4. Complexity: O(N)

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

5. Effects: Sets all bits of the absolute value in \*this for which the corresponding bit in rhs is set. Other bits and the sign of \*this are unchanged

6. Note: if the position of the highest bit in rhs was greater than the position of the highest bit in \*this, \*this will grow

7. Returns: \*this

8. Complexity: O(N)

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

9. Effects: If the position of the highest bit in rhs was greater than the number of bits in \*this, leading zeroes are added to \*this. Toggles every bit in the (possibly extended) absolute value of \*this for which the corresponding bit in rhs is set. Other bits and the sign of \*this are unchanged.
10. Returns: *this
11. Complexity: O(N)

    int integer::operator<=( size_t rhs);
12. Effects: as if calling this->operator*=( 2 * rhs )
13. Complexity: O(N)

    int integer::operator>=( size_t rhs);
14. Effects: as if calling this->operator/=( 2 / rhs )
15. Complexity: O(N)

26.5.6 integer comparisons  [lib.integer.comp]

    bool operator==( integer const& lhs, integer const& rhs);
1. Returns: true if the sign and absolute value of lhs and rhs are equal.
2. Complexity: O(N)

    bool operator!=( integer const& lhs, integer const& rhs);
3. Returns: !(lhs==rhs)

    bool operator> ( integer const& lhs, integer const& rhs);
    bool operator< ( integer const& lhs, integer const& rhs);
    bool operator<=( integer const& lhs, integer const& rhs);
    bool operator>=( integer const& lhs, integer const& rhs);
4. Returns: the ordering of lhs and rhs, based on the sign of (lhs-rhs)

26.5.7 integer non-member arithmetic operators  [lib.integer.arithops]

    integer const& operator+( integer const& rhs);
1. Notes: unary operator
2. Returns: rhs

    integer operator-( integer const& rhs);
3. Notes: unary operator
4. Returns: integer(0) - rhs

    integer operator+( integer const& lhs,
                   integer const& rhs);
    integer operator+( long long lhs, integer const& rhs);
    integer operator+( integer const& lhs, long long rhs);
5. Returns: integer(lhs)+=rhs

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

2005-01-13  Michiel.Salters@LogicaCMG.com
integer operator-(long long lhs, integer const& rhs);
integer operator-(integer const& lhs, long long rhs);

6.  Returns: integer(lhs) -= rhs

integer operator*(integer const& lhs,
                integer const& rhs);
integer operator*(integer const& lhs, long long rhs);
integer operator*(integer const& lhs, double rhs);

7.  Returns: integer(lhs) *= rhs

integer operator*(long long lhs, integer const& rhs);
integer operator*(double lhs, integer const& rhs);

8.  Returns: integer(rhs) *= lhs

integer operator/(integer const& lhs,
                integer const& rhs);
integer operator/(long long lhs, integer const& rhs);
integer operator/(integer const& lhs, long long rhs);
integer operator/(integer const& lhs, double rhs);

9.  Returns: integer(lhs) /= rhs

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

39. Returns: Divides the value of lhs by the value of rhs
40.  Throws: invalid_argument if rhs==0

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

10. Returns: integer(lhs) %= rhs

26.5.8 integer non-member bitwise operators
    [lib.integer.bitops]

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

1. Returns: integer(lhs) &= rhs

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

2. Returns: integer(lhs) |= rhs

integer operator^(integer const& lhs,
                integer const& rhs);
3. **Returns:** `integer(lhs)^=rhs`

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

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

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

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

### 26.5.9 `integer` iostream operators

```
template< class charT, class traits>
basic_istream< charT, traits >&
operator>>(basic_istream< charT, traits >& is, integer& x);
```

1. **Requires:** at least one digit. If no digit is encountered, calls `is.setstate(ios::failbit)` (which may throw `ios::failure` (27.4.4.3))

2. **Effects:** extracts a sign (if present, else ‘+’ is assumed) followed by the longest possible sequence of decimal digits, converts them to an integer value as if `integer::integer(string const&)` was called, and assigns this value to `x`.

3. **Returns:** `is`

```
template< class charT, class traits>
basic_ostream< charT, traits >&
operator<<(basic_ostream< charT, traits >& os, integer const& x);
```

4. **Effects:** inserts the complex number `x` onto the stream `os` as if it were implemented as follows:

```cpp
template< class charT, class traits>
basic_ostream< charT, traits >&
operator<<(basic_ostream< charT, traits >& os, integer const& x) {
    if ( x<0 )
        return os << '-' << (-x);
    else
        return os << (x/10) << char(x%10);
}
```

### 26.5.10 `integer` functions

```
bool getbit( integer const& v, size_t s );
```

1. **Returns:** `((v>>s)&1)==1`

2. **Notes:** Does not throw, but returns 0 if `v<(1<<s)`.

```cpp
void setbit( integer& v, size_t s );
```

3. **Effects:** Sets the bit in position `s`
4. Postcondition: \( \text{getbit}(v, s) == \text{true} \)

\[
\text{void clearbit}( \text{integer} & v, \text{size}_t s);
\]

5. Effects: Clears the bit in position \( s \)

6. Postcondition: \( \text{getbit}(v, s) == \text{false} \)

\[
\text{size}_t \text{lowestbit}( \text{integer const} & v );
\]

7. Returns: the lowest value of \( s \) for which \( \text{getbit}(v, s) \) is true

\[
\text{size}_t \text{highestbit}( \text{integer const} & v );
\]

8. Returns: the highest value of \( s \) for which \( \text{getbit}(v, s) \) is true.

\[
\text{bool even}( \text{integer const} & v );
\]

9. Returns: \( !\text{getbit}(v, 0) \)

\[
\text{bool odd}( \text{integer const} & v );
\]

10. Returns: \( \text{getbit}(v, 0) \)

\[
\text{int sign}( \text{integer const} & v );
\]

11. Returns: \( -1 \) if \( v < 0 \), \( 0 \) if \( v == 0 \), and \( +1 \) if \( v > 1 \)

\[
\text{integer abs}( \text{integer const} & v );
\]

12. Returns: \( v * \text{sign}(v) \)

13. Complexity: \( \text{O}(1) \)

\[
\text{integer sqr}( \text{integer const} & v );
\]

14. Returns: \( v * v \)

15. Complexity: no worse than \( \text{operator}^*( \text{integer const} &, \text{integer const} & ) \)

\[
\text{integer sqrt}( \text{integer const} & v );
\]

16. Returns: the largest number \( R \) for which \( R * R <= v \)

\[
\text{integer gcd}( \text{integer const} & v1, \text{integer const} & v2);
\]

17. Returns: the greatest common divisor, i.e. the highest value of \( d \) for which \( (v1 \% d) \) and \( (v2 \% d) \) are both \( 0 \).

18. Notes: the signs of \( v1 \) and \( v2 \) are ignored; the returned value is always positive.

\[
\text{integer lcm}( \text{integer const} & v1, \text{integer const} & v2);
\]

19. Returns: the least common multiple, i.e. the lowest value of \( m \) for which \( (m \% v1) \) and \( (m \% v2) \) are both \( 0 \).

20. Notes: the signs of \( v1 \) and \( v2 \) are ignored; the returned value is always positive.
integer factorial( unsigned long long v );

Returns: ( v==0 ? 1 : v * factorial (v-1) )

Complexity: O(v)

integer pow( integer const base,
             unsigned long long exp );

Returns: ( v==0 ? 1 : base * factorial (exp-1) )

Complexity: O( log exp )

D. Renumber 26.5 C library to become 26.6:

26.6 C Library

1. Tables 80 and 81 describe headers <cmath> and <cstdlib> ...

Rationale:
The class would fit better after <complex> 26.2, but this would involve renumbering 26.3
<valarray> and 26.4 <numeric>. However, 26.5 (C library) is different enough that it should be
last. Therefore, this proposal inserts the paragraph after 26.4 <numeric>.

Just like std::string introduces a variable-length string to complement char[N], this class introduces
an integer class without a fixed size. Like std::string, no new literals are created. This class too
can use string literals in initializations, e.g.
    std::integer foo = "12345678901234567890";
Like std::string, such initializations can be optimized at compile time. The benefits are larger here,
as this involves a base conversion (decimal-binary), but it still is a QoI issue.

The extra complexity of an unsigned_integer class is considered too much for the savings of a
single sign bit.

The postfix operator++(int) must create a temporary, which is a rather expensive O(n) operation.
Therefore it is not included, nor is operator--(int).

Many functions have overloads taking long long int. These have better complexity bounds,
typically by a factor of N. Adding long int, plain int and/or short int overloads would
provide only marginally more efficient overloads. Adding double overloads would increase the ranges, but
there are no convincing cases known for integer::integer(double) or operator+=
(double). However, operator*=(double) and operator/=(double) are provided.
Closely related to this, it is unclear what the type of (double(0.5) + integer(1)) should be.
The exact implementation of `operator *=` is left to the implementation, but it is required to be better than straightforward multiplication (which would be $O(N^2)$), and the complexity bound specified is $< O(N^2)$). N1692 offers a number of alternatives.

N1692 offers no guidance on division. The Gnu GMP documentation suggests that $< O(N^2)$ is unreasonably expensive (scheduled for GMP 5.0, H2 2005), which is why the `operator /= (integer const&)` proposed here has a quadratic complexity bound. Of course, a high-quality implementation may use the better algorithms.

The `&=, |=, *=` and `^=` operators operate as if the shortest of their operands has leading zeroes added to match the length of the other operand. This ensures that expressions like `(integer ("1234567891234567890")) | 1)` don’t truncate their left hand side.

There is no overload of `operator <<` with signature `integer operator<<( long long lhs, size_t rhs);`. This would allow the expression `i << size_t(65)` but would require core changes.

The `sqr` function is added because the operation is common, but the obvious implementation $v^2$ uses the generic multiplication algorithm. The proposed wording allows this, but also allows optimized algorithms. E.g. using the conventional $O(N^2)$ algorithm, $123\times 123$ is $100*100 + 100*20 + 2*100 + 100*3 + 3*100 + 20*20 + 20*3 + 3*20 + 3*3$ (9 multiplies, 8 additions) whereas $sqr(123)$ can also be calculated as $100*100 + 2*100*20 + 2*100*3 + 20*20 + 2*20*3 + 3*3$ (6 multiplies, 3 bitshifts, 5 additions). (Of course, this implementation is not allowed for either `operator *` or `sqr` because they both should have better complexities.)

Like matrices, these numbers can become quite large, which means that temporaries may have significant overhead. Therefore, similar techniques could be applied to them. In particular, the common form $a*b+c$ could be optimized by returning an intermediate type from `operator *` and adding overloads for that type to `operator +`. This proposal does not allow this, but since no internal representation is mandated a run-time equivalent technique can be used.

Equivalent to the common small string optimization, implementers could choose not to use dynamically allocated memory if the value held is small enough (e.g. no larger then `LONG_LONG_MAX`). This can be enforced by requiring `integer::integer(long long)` to succeed. This proposal does not want to enforce that decision, although it silently assumes that behavior in the description of `operator long long( )`.

This class does not specify an `std::allocator` template argument. This is not an explicit decision, and the LWG might consider adding it.

- Copyright Michiel Salters / Nederlands Normalisatie Instituut.