

A Proposal to add the Infinite Precision Integer to the C++ Standard Library

M.J. Kronenburg
e-mail: M.Kronenburg@inter.nl.net

1 July 2004

Document numbers N1692 04-0132.

1 Motivation

The need of integers not fitting into the data width of the processor increases. For example, the range of the `int` on a 32-bit machine is $\pm 2^{31} \simeq \pm 10^9$. For exceeding this range, a C++ class is created that concatenates many `unsigned ints` and a sign into an integer class whose data width is only limited by available memory size. The mathematical operators are overloaded. The range of this integer class is $\pm 2^{8m}$ where m is the maximum available memory size in bytes. With the current memory sizes, the range is practically unlimited, it is an *infinite precision integer*, and its proposed class type name is `integer`.

Currently a number of implementations of the *infinite precision integer* exist that give a good overview of support, performance and design issues:

1. The `Integer` class in the Gnu C++ library
(C++, limited to about 10^5 decimals [7]).
2. The Gnu Multiple Precision Arithmetic Library
(C with assembler, unlimited [6]).
3. The `Integer` class developed by myself.
(C++ with assembler, unlimited)

Below these implementations are referred to as implementations 1, 2 and 3.

2 Impact on the Standard

There is no impact on the standard, as the class is fully self-contained with its own memory management.

3 Design Decisions

There is a number of design decisions, and the alternatives are mostly present in the implementations listed above. Therefore below the design and other differences between the three implementations are listed.

3.1 Class Structure

The proposed C++ class would be the `integer` class. The arithmetic `int` functions and operators are overloaded. In addition to the functions and operators overloaded from `int`, the greatest common divisor and least common multiple (gcd and lcm) must be present, and also some functions for bit level operations as `setbit`, `clearbit`, `getbit`, `highestbit`, `lowestbit` and so on. In

implementation 1, there is a C++ class `Integer` which contains a `struct` for the internal integer representation. This means that every class function or operator calls some function operating on the `struct` of the internal representation. Implementation 2 is implemented in C, so for usage in C++ a C++-wrapper is needed. In implementation 3, the `Integer` class is a simple C++ class without a `struct` for internal representation. There is a constructor from `string`, so that any number can be specified as a string with decimals of arbitrary length.

3.2 Data Structure

The data in all implementations mentioned above is a contiguous binary block of variable length. The sign is contained in a separate field, and can have values -1 (less than zero), 0 (zero) and 1 (greater than zero). An alternative would be to put the data in an STL vector, but as many `integer` operations are performed at bit level, the dependency on the vector container implementation would be unacceptable. Therefore the `integer` does not use any STL container or function.

3.3 Data Granularity

For the operations on the `integer` data, its granularity must be defined, that is how wide are the data chunks that are operated on. On 32-bit processors, implementation 1 uses short 16-bit granularity, because this means that all shift and carry operations can be done using 32-bit `ints` and no assembler is needed. On 32-bit processors, the other implementations use 32-bit granularity, which means that assembler is unavoidable. Currently it is not clear to me if 64-bit processors can handle 64-bit granularity `integers`.

3.4 Performance

The performance of all arithmetic operations is much better using 32-bit granularity and assembler. For the multiplication of two large `integers`, a number of algorithms exist (N is number of decimals or bits):

Algorithm	implementation	order	decimals	reference
basecase	1,2,3	N^2	$1 - 10^2$	
Karatsuba	2,3	$N^{1.585}$	$10^2 - 10^3$	[1,4]
3-way Toom-Cook	2	$N^{1.465}$	$> 10^3$	[1,4]
16-way Toom-Cook	3	$N^{1.239}$	$> 10^3$	[1]
Schönhage NTT	2	$N \log N \log \log N$	$> 10^3$	[2,3,4]
Strassen FFT	-	$N \log^2 N$	$> 10^3$	[1,3,5]

(NTT is Number Theoretic Transform, FFT is Fast Fourier Transform). The Strassen FFT algorithm uses floating-point arithmetic, which means that its accuracy cannot be mathematically guaranteed for very large arguments [4], and it is therefore not used in the quoted implementations, but its performance is the best of all. Implementation 1 only uses basecase multiplication, and therefore for large arguments its performance is poor. In implementation 3 I found that 16-way Toom-Cook is faster than Schönhage NTT (up to some very large argument not known to me).

For division and remainder recursive algorithms lead to better performance for large arguments, and the same is true for the instream and outstream, which means conversion from binary to decimal notation and vice versa.

In some cases the performance may not be of any interest, but users that start using the *infinite precision integer* may tend to test the class for large arguments and compare results with the well known commercial computer algebra programs. This may be even more the case when on top of the *infinite precision integer* the *arbitrary precision real* is defined.

3.5 Assembler Usage

Implementation 1 does not use assembler, and runs therefore on any platform. Implementation 2 uses assembler, and is targeted for Unix-like platforms running a C compiler and its assembler [4]. Implementation 3 uses assembler, and is targeted for Borland C++ Builder and its Turbo Assembler.

3.6 Conclusion

When a choice between the existing three implementations listed above must be made, the following may be considered. As implementation 1 is limited to about 10^5 decimals (it is not clear to me why this limit exists), this implementation does not seem useful, as the aim of the *infinite precision integer* is to be able to use an unlimited number of decimals. For large **integers** its performance is poor. Implementation 2 may be preferred on Unix-like platforms, all that seems to be needed is a C++-wrapper. Implementation 3 may be preferred on Windows-like Intel platforms. Another option is to collaborate with development of a commercial computer algebra program.

4 Proposed Text

4.1 Requirements

The requirements to the **integer** class are provided by the proposed interface in the header file. The assertions and pre/post-conditions are self-evident, the required average complexity is provided, where N is the number of decimals or bits:

```

class integer
{
private:
    unsigned int *data, *maxdata;
    signed char  thesign;
public:
    integer(); // Complexity: // 1
    integer( int ); // 1
    integer( double ); // 1
    integer( const char * ); // < N^2 (see 3.4)
    integer( const string & ); // < N^2 (see 3.4)
    integer( const integer & ); // N
    virtual ~integer(); // 1

    const unsigned int size() const; // 1
    integer &operator=( const integer & ); // N
    integer &negate(); // 1
    integer &abs(); // 1
    integer &operator++(); // 1
    integer &operator--(); // 1
    const integer operator++( int ); // N
    const integer operator--( int ); // N
    integer &operator|=( const integer & ); // N
    integer &operator&=( const integer & ); // N
    integer &operator^=( const integer & ); // N
    integer &operator<<=( unsigned int ); // N
    integer &operator>>=( unsigned int ); // N
    integer &operator+=( const integer & ); // N
    integer &operator-=( const integer & ); // N
    integer &operator*=( const integer & ); // < N^2 (see 3.4)
    integer &operator/=( const integer & ); // < N^2 (see 3.4)
    integer &operator%=( const integer & ); // < N^2 (see 3.4)
    const integer operator-() const; // N
    const integer operator<<( unsigned int ) const; // N
    const integer operator>>( unsigned int ) const; // N
};

const bool operator==( const integer &, const integer & ); // N
const bool operator!=( const integer &, const integer & ); // N
const bool operator>( const integer &, const integer & ); // N
const bool operator>=( const integer &, const integer & ); // N
const bool operator<( const integer &, const integer & ); // N
const bool operator<=( const integer &, const integer & ); // N

const integer operator|( const integer &, const integer & ); // N

```

```

const integer operator&( const integer &, const integer & ); // N
const integer operator^( const integer &, const integer & ); // N

const integer operator+( const integer &, const integer & ); // N
const integer operator-( const integer &, const integer & ); // N
const integer operator*( const integer &, const integer & ); // < N^2 (see 3.4)
const integer operator/( const integer &, const integer & ); // < N^2 (see 3.4)
const integer operator%( const integer &, const integer & ); // < N^2 (see 3.4)

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

ostream & operator<<( ostream &, const integer & ); // < N^2 (see 3.4)
istream & operator>>( istream &, integer & ); // < N^2 (see 3.4)

const int sign( const integer & ); // 1
const bool even( const integer & ); // 1
const bool odd( const integer & ); // 1

const bool getbit( const integer &, unsigned int ); // 1
void setbit( integer &, unsigned int ); // 1
void clearbit( integer &, unsigned int ); // 1
const unsigned int lowestbit( const integer & ); // 1
const unsigned int highestbit( const integer & ); // 1

const integer abs( const integer & ); // N
const integer sqr( const integer & ); // < N^2 (see 3.4)
const integer pow( const integer &, const Integer & ); // ?
const integer factorial( const integer & ); // ?

// floor of the square root, like int sqrt( int )
const integer sqrt( const integer & ); // ?
// random integer >= first and < second argument
const integer random( const integer &, const integer & ); // ?

const int toint( const integer & ); // 1
const double todouble( const integer & ); // 1

```

For error handling, a separate `exception` class may be created:

```

class integer_exception : public exception
{ public:
    enum type_of_error {
        error_unknown, error_overflow,
        error_divbyzero, error_memalloc, ...
    };
};

```

```
};  
integer_exception( type_of_error = error_unknown, ... );  
virtual const char * what () const;  
private:  
    type_of_error error_type;  
    string error_description;  
};
```

5 Unresolved Issues

1. The required complexities of `gcd`, `lcm`, `sqrt`, `pow` and `factorial` are currently not yet clear.
2. The overall performance of the *infinite precision integer* should be comparable with the overall performance of the well known commercial computer algebra programs.
3. On top of the *infinite precision integer*, the *arbitrary precision real* can be defined.

6 References

1. D.E. Knuth, The Art of Computer Programming, Volume 2 (1998).
2. P. Zimmermann, An implementation of Schönhage's multiplication algorithm (1992).
3. A. Schönhage and V. Strassen, Computing 7 (1971) 281.
4. Free Software Foundation, Gnu MP manual ed. 4.1.2 (2002).
5. <http://numbers.computation.free.fr/Constants/Algorithms/fft.html>
6. <http://www.swox.com/gmp>
7. http://www.math.utah.edu/docs/info/libg++_20.html