# CBigInt CLASS: AN IMPLEMENTATION OF BIG INTEGERS IN C++[*]

*Shyamal Suhana Chandra*
*University of Pittsburgh*
*Pittsburgh, PA 15260*
*shyamalc@gmail.com*

*Kailash Chandra*
*Computer Science Information Systems*
*Kelce College of Business*
*Pittsburg State University*
*Pittsburg, KS 66762*
*kchandra@pittstate.edu*
*(620)235-4539*

**ABSTRACT**

A dynamic implementation of big integers in C++ is presented. A class that implements big integers of unlimited precision is developed and used to introduce the students to dynamic data structures, several algorithms, and several advanced features of C++. The big integer class is implemented using a doubly-linked list and various features of C++, such as, constructors, destructors, function overloading, operator overloading, and container classes. Several incremental programming assignments are also presented that can be used during a course focused on data structures.

## INTRODUCTION

This paper describes a C++ implementation of big integers along with functions and operators that can be used to manipulate them just like other built-in types of values. This implementation of big integers was also used to introduce students in a data structures course to dynamic data structures and several advanced features of C++ programming

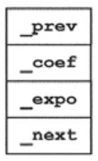
[*] Copyright © 2005 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

language. It was important for the students to be exposed to operator overloading and function overloading. The big integer class and its associated functions were developed in an incremental fashion with students being required to provide test functions for each function or operator defined for the class. Students were also required to implement various simple functions and build on them to define other functions. This process exposed the students to modular programming, software reuse, and handling of interface problems at an early stage.

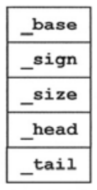
There are various references that encourage introducing data structures in the classroom while keeping it interesting and challenging at the same time [1-4]. These authors have introduced and implemented different data structures using various unique features of different programming languages. Most of the implementations [5-8] of big integers are static and dense list based, we have chosen it to be dynamic linked list based. There are various reasons for choosing the dynamic representation of big integers in addition to some extra features implemented. Here are some of them:

1. Unlike static implementations, the dynamic implementation presented has no limit on the precision, only limit is placed by the hardware, in this case, the available main memory address space.
2. Although there are some memory and processing overhead in terms of pointers with each digit, but only the digits necessary to represent a value are stored, unlike the fixed storage requirement irrespective of the number of significant digits in case of static implementations.
3. On top of this, only the non-zero digits are stored to save memory. This feature is possible only in dynamic representation.
4. It is possible to change the base of a number in place. This feature is easily implemented only in dynamic representation.
5. Using the scheme presented, it is possible to allow mixed-base calculations easily. For example, one can add a number given in base 2 to a number given in base 8.
6. Shift operations are very quick, for a shift to right, only a digit from the right is deleted and weight of each digit is changed, and for a left shift, only the weight of each digit is changed.
7. Also, in our implementation, each digit is 16 bits long, making the base of our scheme as 2^16, requiring very few digits and thus not so much of overhead in terms of pointers even for very large integer values. This can easily be changed to 32 bits or 64 bits.

There are two object types used in our implementation, `CDigit` and `CBigInt`, as represented by the following diagrams:



CDigit



CBigInt

CDigit type of objects are used to store the digits making a big integer. It has the digit in a given base, the weight of the digit represented by the exponent of the base, the address of the digit to the left, and the address of the digit to the right. CBigInt type of objects are used to store big integers as a doubly-linked list of CDigit type of objects. It holds the base in which the number is being stored, sign of the number, size to represent the number of non-zero digits (unless there is only one zero digit) in the number, head to store the address of the first digit, and tail to store the address of the last digit. A big integer such as 400000020502 can be represented by the following expression where b is the base of the big integer:

$$4000000000020502 = 4b^{15} + 2b^{4} + 5b^{2} + 2$$

In general, any big integer of size n can be represented by the following expression:

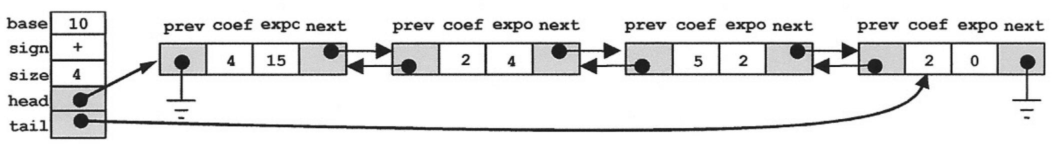
$$c_{n-1}b^{n-1} + c_{n-2}b^{n-2} + c_{n-3}b^{n-3} \ldots + c_0 b^0$$

where the coefficients $c_n \neq 0$ and represent the digits in a big integer. The exponents are nonnegative integers. Each $c_i b^i$ represents a weighted digit of the big integer. We have developed two classes, CDigit to implement the digits in a big integer and CBigInt to implement the big integers. Each big integer is represented as a doubly linked list of digits.

The class CDigit has four private data members: _prev to hold the address of the previous digit, _coef to hold the coefficient of the digit, _expo to hold the exponent of base, and _next to hold the address of the next digit. In addition to several constructors and functions, the operators =, ==, !=, <<, >>, +, -, *, ++, --, and / have been implemented.

The class CBigInt has five private data members: _base to hold the, _sign to hold the, _size to hold the number of terms in the polynomial, _head to hold the address of the first node, and _tail to hold the address of the last node. In addition to several constructors and other useful functions, the operators =, ==, !=, <<, >>, +, -, *, ++, --, /, +=, -=, *=, and /= have been implemented. We have also implemented mixed base arithmetic operations. Only non-zero digits are allowed in the proposed big integer implementation. The terms are maintained in decreasing order of exponent. An example of the representation for the integer 4000000000020502 and corresponding doubly-linked list representation would be as follows:

```
+4*b^15 + 2*b^4 + 5*b^2 + 2*b^0
```



As evident from the example above, there are two types of objects used in this dynamic representation: (1) the control block that holds the base, the sign, the number of non-zero digits, the address of the first digit, and the address of the last digit; and (2) digit nodes that hold the coefficient, the exponent of the base, pointer to the previous digit, and pointer to the next digit. The following sections describe some basic features of this implementation, provide a brief description of the CDigit and CBigInt classes along with their members, and list several small student projects that can be based on these classes.

## DOUBLY-LINKED LIST DATA STRUCTURE

Doubly-linked list is a bidirectional linear list in which each element contains pointers to the next and previous elements in the list. As mentioned earlier, two classes are defined to implement the big integers. The first is `CDigit` to hold the digits in a big integer and the second is `CBigInt` to hold the doubly-linked list of objects of `CDigit` type. The `CDigit` class was defined outside the `CBigInt` class so that it can be used outside the `CBigInt` class and also to demonstrate the ability of C++ to define new classes based on existing classes. The class `CDigit` has no destructor function member as the objects of this type have no resources to be released.

## GLOBAL DECLARATIONS AND DEFINITIONS

Following is a list of some type definitions and constants. Types are defined in order to make the implementation portable. For example, when 64-bit processors become available, we will need to only redefine the types: `uShort, sShort, uLong, sLong`, and `uChar,` through these statements and they will take effect for the entire implementation. The constants defined below have a similar effect, they can be modified if and when necessary and change the entire implementation. These changes could be made as the changes occur in the hardware and/or software platforms.

```
typedef unsigned short uShort;
typedef signed short   sShort;
typedef unsigned long  uLong;
typedef signed long    sLong;
typedef unsigned char  uChar;
const uShort BASE      = 10; //65536
const uShort MIN_VALUE = 0;
const uShort MAX_VALUE = BASE-1;
const uShort MAX_EXPO  = 10; //65536
const uShort MAX_SIZE  = MAX_EXPO+1;
const uShort UNDEFINED = 911;
```

## CDigit CLASS DEFINITION

Following is a list of all the data members and some of the important function members of the class `CDigit`.

```
class CDigit
  {
  private:
    CDigit *_prev;
    uShort  _coef;
    uShort  _expo;
    CDigit *_next;
  public:
    CDigit(void);
    CDigit(char ch);
    CDigit(const CDigit &d2);
    CDigit(uShort coef, uShort expo=0);
    bool   operator   == (const CDigit &d2) const;
    bool   operator   != (const CDigit &d2) const;
    bool operator     <  (const CDigit &d2) const;
    bool operator     >  (const CDigit &d2) const;
    bool operator     >= (const CDigit &d2) const;
    bool operator     <= (const CDigit &d2) const;
    CDigit operator   +  (const CDigit &d2) const;
    CDigit operator   *  (const CDigit &d2) const;
    CDigit operator   -  (const CDigit &d2) const;
    CDigit operator   /  (const CDigit &d2) const;
    CDigit & operator =  (const CDigit &d2);
    CDigit & operator++();
    CDigit & operator--();
    CDigit & operator++(int dummy);
    CDigit & operator--(int dummy);
    friend class CBigInt;
    friend ostream & operator << (ostream &bob, const CDigit &d2);
  };
```

## CBigInt CLASS DEFINITION

Following is a list of all the data members and some of the important function members of the class `CBigInt`.

```
class CBigInt
  {
  private:
    uShort  _base;
    char    _sign;
    uShort  _size;
    CDigit *_head;
    CDigit *_tail;
  public:
    CBigInt(uShort base = BASE);
    CBigInt(char ch, uShort base = BASE);
    CBigInt(const CBigInt &bi2);
    CBigInt(const char *decStr, uShort base = BASE);
    ~CBigInt(void);
    char *toStr(uShort base=BASE) const;
    CBigInt & operator = (const CBigInt &bi2);
    bool operator     ==  (const CBigInt &bi2) const;
    bool operator     !=  (const CBigInt &bi2) const;
    bool operator      <  (const CBigInt &bi2) const;
    bool operator      <= (const CBigInt &bi2) const;
    bool operator      >  (const CBigInt &bi2) const;
    bool operator      >= (const CBigInt &bi2) const;
    CBigInt & operator +  (const CBigInt &bi2) const;
    CBigInt & operator -  (const CBigInt &bi2) const;
    CBigInt & operator *  (const CBigInt &bi2) const;
    CBigInt & operator /  (const CBigInt &bi2) const;
    CBigInt & operator %  (const CBigInt &bi2) const;
    CBigInt & operator * (const CDigit &d2) const;
    CBigInt & operator + (const CDigit &d2) const;
    CBigInt & operator++();
    CBigInt & operator--();
    CBigInt & operator++(int dummy);
    CBigInt & operator--(int dummy);
    friend ostream & operator << (ostream &bob, const CBigInt &bi);
  };
```

## PROPOSED PROGRAMMING PROJECTS

The approach for incorporating the big integer classes in the data structures course involves the use of several mini projects. This approach makes the learning process gradual and the potential problems and difficulties are discovered at an early stage in the process. The mini projects help most of the students and reinforces key ideas through repetition. This approach has proven to be successful with various students achieving a reasonable level of success in completing the mini projects within the specified time. There are many possible mini projects that can be assigned using these concepts. Below are brief descriptions of some of the possible programming projects. As one will notice, it is suggested that a test function should be required for each function developed.

1. Implement `CDigit` class with default constructor, copy constructor, constructor with given digit and its weight through the exponent of the base value, overload insertion operator << for output, and test functions for each member function.
2. Add the definition of `CBigInt` to the previous project with a default constructor, a constructor to create a random big integer, overloaded insertion operator << for output, and test functions for each member functions.
3. Add `toStr` and `~CBigInt` destructor to properly release the memory, member functions to the previous project. Make sure there is also a test function for each function added.
4. Overload the operators `==, !=, <, <=, >,` and `>=` to the previous project. Also write test functions to thoroughly test the implementations of each new member function.
5. Overload the operators addition `(+),` subtraction `(-),` assignment `(=),` multiplication `(*)` to the previous project and also add a test function for each overloaded operator.
6. Overload divide `(/)` operator and develop a test function, to the previous project.

## CONCLUSION AND SUMMARY

We have discussed the implementation of the big integer class using a doubly-linked list, various simple functions, and overloaded operators so that the big integers can be handled just like any other primitive type value. With a slight modification to the current implementation, by including just the position of a decimal point, it would be easily possible to handle the integer and non-integer big numbers. By choosing the dynamic implementation, we can achieve almost limitless precision, store only the digits necessary to represent a value, store only the non-zero digits, possibility to change the base of a number in place, allow mixed-base calculations easily, and speedy shift operations. Along with these, a set of mini projects has been presented that will step-by-step involve the students in learning about algorithm development, the dynamic data structures, reinforce student knowledge concerning class definition and usage, and introduce students to more advanced features of C++ such as operator and function overloading.

## REFERENCES

[1] Makinen, Erkki, Programming Projects on Chess, *SIGCSE Bulletin*, Vol. 28, No. 4, December 1996, pp 41-44.

[2] Meisalo, V., E. Sutinen, and J. Tarhio, CLAP: teaching data structures in a creative way, *SIGCSE Bulletin*, Vol. 29, No. 3, September 1997, pp 117-119.

[3] Gurwitz, Chaya, Teaching Linked Lists: *Computer Science Education*, Vol. 9, No. 1, 1999, pp 36-42.

[4] Pifile, Richard, Using a Model Train to Illustrate a Doubly Linked List in a Microprocessor Laboratory, *Computers in Education Journal*, Vol. 7, No. 1, Jan-May 1997, pp 23-26.

[5] LiDIA Manual, A library for computational number theory, Version 2.0a, http://www.wschnei.de/number-theory-software/lidia/html/lidiaman.html, retrieved January 24, 2005.

[6] Class BigInteger, JavaTM 2 Platform, Sun Microsystems, http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigInteger.html, retrieved January 24, 2005.

[7] BigInt - arbitrary precision integer via the gnu gmp library, http://coldstore.sourceforge.net/coldstore/kdoc/BigInt.html, retrieved January 24, 2005.

[8] Tan, Chew Keong, The Code Project, C# BigInteger Class, C# Programming, http://www.codeproject.com/csharp/BigInteger.asp, retrieved January 24, 2005.