# Assignment 2

**Written Part Due:** 11:59 pm, Wednesday, September 20, 2006
**Coding Part Due:** 11:59 pm, Wednesday, September 20, 2006
**Review Part Due:** 5:00 pm, Friday, September 22, 2006

## Preliminaries

Get the files for this assignment by executing `svn update` from your `cs70` directory. This command should check out the following files:

```
hw2/HW2-Answers.txt
hw2/HW2-Rubric.txt
hw2/Answers.txt
hw2/Makefile
hw2/bigint.hpp
hw2/bigint.cpp
hw2/factorial.cpp
hw2/mytests.cpp
```

The assignment directory contains a C++ implementation of a *BigInt* class that provides arbitrary-sized integers (on most 32-bit systems *int* can only store values in the range $-2^{31} \cdots 2^{31}-1$), and a simple test program that calculates factorials. Your goal is to modify the program so that it can calculate large factorials with complete accuracy (e.g., 50!, which is a 64-digit number). At present, the *BigInt* class is only partially complete and cannot manipulate numbers larger than $2^{31}-1$, and so can only calculate factorials up to 12!.

## Written Questions

Most answers to the questions in this section can be one sentence (or, in some cases, one line of code) only. You can make modifications to the code for the assignment to answer these questions, but should always return the code to its previous (working) state before moving on to the next question.

Your answers must go into the file `Answers.txt` . Be sure to include your names, and to use good grammar and spelling.

W1. What do the following member functions in the *vector<int>* class do?

(a) `push_back`
(b) `size`
(c) `capacity`
(d) `resize`
(e) `reserve`

W2. If `v` is of type *vector<int>*, the statement `v[v.size()] = 1;` has *undefined behavior* according to the C++ standard.

(a) Why is this code incorrect?

(b) Explain two different goals the programmer might have hoped this statement would achieve, and for each one give the correct (one-line) statement.

(c) Give two different run-time outcomes possible when the above bad statement executes. (Unlike the friendly version of *vector* found in Weiss, most implementations *do not* throw an exception in this case.)

W3. The code in `bigint.cpp` contains numerous instances of the expression `*this`.

(a) What does `this` mean in C++?

(b) What does `*this` mean in C++?

(c) If you saw code for a member function referring to `(*this).chunks_` or to `this->chunks_`, you could (and almost always should) replace this code with what simpler expression?

W4. Explain what each of the following global-function declarations mean, highlighting the differences between them:

(a) `BigInt factorial(BigInt n)`

(b) `BigInt factorial(const BigInt n)`

(c) `BigInt factorial(BigInt& n)`

(d) `BigInt factorial(const BigInt& n)`

W5. Explain what each of the following *member*-function declarations mean, highlighting the differences between them:

(a) `BigInt operator+(const BigInt& rhs);`

(b) `BigInt operator+(const BigInt& rhs) const;`

W6. The *BigInt* class declares its constructor as follows (in `bigint.hpp`):

```
explicit BigInt(int value = 0);
```

Your answer should probably begin "Because it isn't a..."

(a) Why doesn't the constructor specify a return type?

(b) What does the keyword **explicit** mean?

(c) If the **explicit** keyword were removed, the factorial function could be simplified by deleting some parts of the code and leaving the rest unchanged.

i. What are these simplifications?

ii. Why can these simplifications be made?

iii. Give one reason for removing **explicit** and simplifying the code.

iv. Give one reason for *not* removing **explicit**.

W7. If we were defining our own copy constructor for the *BigInt* class, it would have to be defined as `BigInt(const BigInt& orig)` and not `BigInt(BigInt orig)`. Explain why the second form would generate an infinite loop.

In CS 70, we require that you either write your versions of these functions, disable them, or explicitly state in a comment in the class definition why the compiler's defaults are okay.

W8. In C++, the compiler will automatically provide every class with a copy constructor and an assignment operator if you do not declare them. If you were to write the *BigInt* copy constructor explicitly, what would its definition be? (To receive full credit, your code should be as simple as possible, with no code inside the braces. Hint: The *vector<int>* class has a copy constructor.)

W9. The *BigInt* class represents a number as a *vector<int>* called `chunks_`:

Consider what happens when we write the declaration *BigInt* `x(27183142)`. When this declaration is encountered during execution, the `BigInt(`*int* `value)` constructor is run. What is the size and the contents of the vector, listed in order, starting at zero

(a) Just after the opening brace of the constructor?

(b) Just before the closing brace of the constructor?

W10. The *BigInt* class represents zero in an "interesting" way. What is it?

W11. Compiling and executing `./mytests`, which (as provided) contains the code

```
BigInt x(1234);
BigInt y(5678);
cout << "Before multiply: x = " << x << ", y = " << y << endl;
x *= y;
cout << "After multiply: x =  " << x << ", y = " << y << endl;
```

prints out

```
Before multiply: x = 34, y = 78
After multiply:  x = 2652, y = 78
```

This incorrect output is due to temporary code in the *BigInt* class—the code implementing both the `*=` operator and the `print` member function is just a placeholder for code that you will write later.

(a) What would the above code output if `print` were fixed to operate correctly and `*=` were left unchanged?

(b) What would the above code output if `*=` were fixed to operate correctly and `print` were left unchanged?

(c) The placeholder code actually operates correctly provided that the *BigInt* is within a certain range. What is that range?

(d) Given these limitations, why does the program print `3628800` when asked to calculate the factorial of 10 (i.e., the correct answer)?

## Coding Component

The code for this assignment is written using an *evolutionary* style. The key idea is to design the basics of the program, and then begin to implement that design using milestones at which portions of the incomplete program can be shown to work. The stages of construction for this project are

1. Creating the `bigint.hpp` header file, declaring the most essential operations of the `bigint.hpp` class
2. Creating a preliminary implementation of the *BigInt* class that has very limited functionality
3. Creating a test program (in this case, `factorial.cpp`) to test some of the functionality of the *BigInt* class
4. Testing the preliminary code to ensure that the basic foundations are working
5. Implementing more functionality in the *BigInt* class (with frequent testing)
6. Enhancing the tests to test more functionality, paying attention to testing boundary cases
7. Repeat from step 5 until finished

The rationale behind this development strategy is simple—quick gratification ("I have a program that works and does something!"), and ease of debugging ("Oops, I just broke something, what did I do?"). Or, in other words, you begin by making something trivial that works, then mold it into the final code. (This strategy is not always the best choice, especially if you begin coding before you have adequately considered the overall design of your program.)

The code you have been supplied with is at stage 4—a preliminary program exists and it is possible to test the code. In this assignment, you will be doing some work at stage 5—you will not be finishing the evolutionary process in this assignment (although you can for fun, if you like).

### Coding Questions

These coding questions are designed so that you can submit your code after completing each question. You will, however, only be graded on your final submission.

Your final submission should not change the text messages from the `factorial` program (i.e., no extra debugging output should appear). You should not change the interface of the *BigInt* class beyond that specified, or to add private "helper" member functions.

**Note:** You do *not* need to handle negative integers in this assignment. You are also *not* required to implement division or subtraction.

C1. Replace the placeholder code for the print member function with correct code. (There are some handy tips in the *Tricky Stuff* section at the end of the assignment that you may want to read first.) In this part, (and only this part) you may assume that CHUNK_SIZE is a power of 10. You may not assume it is exactly 10. If you do assume it is a power of 10 for printing, you must document your assumption in the comments.

If you do assume it is a power of 10, you should document it your comments.

Extend `mytests.cpp` to test this code. Be sure to look at extremes and boundary-cases, and to consider what different sorts of numbers might be worth testing.

C2. You can imagine that your integer is *implicitly* prefixed by an infinite number of leading zeros. Since we know they're all zeros, we can avoid keeping these zero-valued chunks in memory. To simulate this view of the number as an infinite sequence of digits (where we can get or set any digit in the sequence), add the following two private member functions to the BigInt class:

```
int getChunk(int index) const;
void setChunk(int index, int value);
```

The getChunk function should return chunks_[i], or 0 if there is no such chunk (i.e., if we're past the non-zero chunks). The setChunk function should ensure that chunks_[i] exists (increasing the size of the vector if necessary and filling in any intermediate zeros) and then set chunks_[i] to the supplied value.

You can and should use these functions later where it simplifies your code.

C3. Replace the placeholder code for the += member function with correct code. You'll have to devise the algorithm for performing addition, but it is basically the same algorithm that you learned in grade school to add large base-ten numbers (using a pencil and paper), except that in this case you have base-CHUNK_SIZE numbers.

(There may be more "interesting" test cases here than for print (e.g., single-chunk number plus multi-chunk number, multi-chunk number plus single-chunk number, etc.).

Test your new code by adding suitable test cases to `mytests.cpp`.

C4. Using setChunk and getChunk, write a private member function *void* easyMultiply(*int* v) that multiplies a *BigInt* by a small integer v that has a value between 0 and CHUNK_SIZE–1. (You may assume that $\text{CHUNK\_SIZE}^2$ will fit in an *int*.) One possible algorithm is:

> Multiply the number in chunks_ by v:
> > If v is zero, set the object to the representation of zero. Otherwise,
> > Initialize carry to zero.
> > For each chunk (going from least significant to most significant):
> > > Multiply the chunk value by v, and add the carry.
> > > The new chunk value and carry are this value
> > > > modulo CHUNK_SIZE and divided by CHUNK_SIZE respectively
>
> > If there is a non-zero carry left over, put it into a new leading chunk.

but variations are possible (e.g., a single loop that goes until you've both run out of chunks and have a carry of zero).

C5. Modify the code for *= so that it calls easyMultiply(rhs.chunks_[0]), and test the factorial program. This version of *= is still merely a placeholder for a complete implementation, but should be sufficient to calculate very large factorials (up to 99!).

C6. Modify the code for *= so that it operates correctly for all *BigInt*s, again using easyMultiply. (You may add other private member functions as well if this makes the multiplication code easier.)

C7. Add a less-than operation for *BigInt* values. The class interface in `bigint.hpp` must be extended by the specification

```
bool operator<(const BigInt& rhs) const;
```

(with some comments), and your code must be extended by the definition of this member function:

```
bool BigInt::operator<(const BigInt& rhs) const
{
  ⋮
}
```

### Tricky Stuff

Depending on how you code part C2 of this assignment, you may want to print integers with leading zeros. Usually, C++ prints out numbers without any leading zeros, but the language provides two facilities that can help, the first is the setw *ostream* manipulator (which controls the "field width" for printed numbers), and the second is the fill *ostream* method (which sets the character used for padding a number out to the field width set with setw). The short test program below shows how these two features can be used.

Sometimes example code in CS 70 violates the style rules for assignments. For example, this code contains some magic numbers, which would be bad style in a CS 70 homework submission (except for `mytests.cpp`).

```
#include <iostream>
#include <iomanip>
using namespace std;                          // Avoid having to say std::cout, etc.

int main()
{
    int exampleInt = 17;

    cout << setw(4) << exampleInt << endl;
    const char oldFill = cout.fill('0');      // Be nice, save old fill char...
    cout << setw(4) << exampleInt << endl;
    cout.fill(oldFill);                       // Restore it
    cout << setw(5) << exampleInt << endl;
    cout << exampleInt << endl;
}
```

When run, this program outputs

```
   17
0017
    17
17
```

Note that we have followed good practice by restoring the old fill character after we were done printing leading zeros.