Assignment 2

`Answers.txt` Due:	10:00 PM,	Tuesday, September 16, 2003
Runnable Code Due:	10:00 PM,	Wednesday, September 17, 2003
Final Code Due:	10:00 PM,	Thursday, September 18, 2003

Preliminaries

As with all assignments in this course, part of your grade will be based on your coding style and on the way your written work is presented. The *Homework Policies* handout discusses these issues in depth.

Before You Begin

Set up your assignment directory on turing by executing

```
cs70checkout cs70ass2
```

This command should check out the following files and directories:

```
cs70ass2/SCHEDULE
cs70ass2/Answers.txt
cs70ass2/Makefile
cs70ass2/bigint.hpp
cs70ass2/bigint.cpp
cs70ass2/factorial.cpp
cs70ass2/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

Unless otherwise stated, answers to the questions in this section should 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.

W1. If you compile the code (by typing `make`), and run the `factorial` program,

(a) What happens if you type in `0` for the number?

(b) What did the programmer do to cause this behavior?

W2. The code contains numerous instances of the expression *this.

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

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

W3. The *BigInt* class declares its constructor as follows (in `bigint.hpp`):

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

Your answer for the first part 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**.

W4. Explain what each of the following 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. 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.

W6. 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? (Your answer will require more than one line of code. 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.)

W7. There are no explicit constructor calls to copy *BigInt* objects, yet the copy constructor is called in some situations. When calculating 7! using the provided code,

(a) How many times is the copy constructor invoked? (Hint: You can use your answer to the previous question to write a copy constructor that prints a message each time the copy constructor is run.)

(b) For each copy-constructor call, explain why the copy was made. (You must list the copies in the order they occur.)

(c) How could you eliminate the first copy-constructor call?

W8. What do the following member functions in the *vector<int>* class do?

(a) `push_back`

(b) `size`

(c) `capacity`

W9. 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? (What did the programmer probably mean?)

(b) Give two possible outcomes for this coding error in practice. (Unlike the friendly version of *vector* found in Weiss, most implementations *do not* throw an exception in this case.)

W10. Internally, the *BigInt* class represents a number as a *vector<int>* called `chunks_`:

(a) 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

i. Just after the opening brace of the constructor?

ii. Just before the closing brace of the constructor?

(b) How many times is the `BigInt(`*int* `value)` constructor executed when calculating 7!?

(c) Many of these `BigInt` constructor calls could be eliminated. How?

W11. The *BigInt* class represents zero in an "interesting" way. What is it?

W12. Compiling and executing `./mytests`, which (as provided) contains the code

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

prints out `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 of the functionality of 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.)

Talk to me if you want to take things further for extra credit.

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.

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

C1. Revise the implementation of factorial so that it uses *= and +=, instead of * and + while keeping the code otherwise unchanged. (Think about whether or not this is a "bogus efficiency" change or not—I may ask you about it in Wednesday's class.)

C2. Replace the placeholder code for the print member function with correct code. Test your new code.

In this part, (and only this part) you may assume that CHUNK_SIZE is a multiple of 10, but you may not assume it is exactly 10.

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

C3. Add two private member functions to the BigInt class:

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

You can and should use these functions to make the rest of your code simple and elegant.

The getChunk function should return chunks_[i] or 0 if there is no such chunk. The setChunk function should ensure that chunks_[i] exists (increasing the size of the vector if necessary) and then set chunks_[i] to the supplied value.

C4. 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.

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

C5. 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.

C6. Modify the code for *= so that it calls lhs.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!).

C7. Modify the code for *= so that it operates correctly for all *BigInt*s.

You may not change the text messages output by your `factorial` program, nor may you change the interface of the *BigInt* class beyond that specified.

You can and should modify `mytests.cpp` to perform any tests that you think will either confirm that your code works correctly or uncover bugs. What the `mytests` program outputs is up to you. During grading we will examine `mytests.cpp` to see how well you have checked the various boundary cases that arise. When we grade this file, we will not expect the same level of attention to style as your other code—it is in most respects your private file for your own experiments and tests. It will need to be readable enough for us to determine what you were testing, however.

When constructing test cases, make sure to think of boundary cases where your code could fail and test them. Bugs due to forgotten boundary cases are the most common reason for not getting full credit on the coding part of this assignment.

Remember to submit your code using the electronic submission system described in the *Electronic Submission* handout. You are not required to include a `README` file with this assignment.

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:: on 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
```