Assignment 2

Documents Due: 2:45 pm, Tuesday, February 5, 2002
Code Due: 9:30 am, Thursday, February 7, 2002

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 and also explains how to submit your work. If you do not have a copy of this handout, you can obtain a copy by visiting the course website at http://www.cs.hmc.edu/courses/2002/spring/cs70/.

Before You Begin

Download the sample source code included with this assignment from the Homework area of the course website. The file archive to download is `assign2.tgz`. On *turing*, you can download and unpack the file archive by executing

```
curl -O http://www.cs.hmc.edu/courses/2002/spring/cs70/homework/assign2.tgz
tar xzvf assign2.tgz
```

This archive contains the following files and directories:

```
cs70ass2/Makefile
cs70ass2/bigint.cpp
cs70ass2/bigint.hpp
cs70ass2/factorial.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. The eventual goal is for the program to be able to 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. The *BigInt* class declares its constructor as follows (in `bigint.hpp`):

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

(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** and simplifying the code.

W2. 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)`

W3. 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.

W4. In C++, the compiler will automatically provide every class with a copy constructor and an assignment operator if you do not declare them.[1] If you were to write the copy constructor explicitly, what would its definition be? (The answer will be 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.)

W5. 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?

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

W6. What do the following member functions in the *vector<int>* class do?

(a) `push_back`

(b) `size`

(c) `capacity`

W7. 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.)

W8. 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)` executed when calculating 7! ?

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

W9. Executing

```
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 will be written 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)?

W10. The coding part of this assignment does *not* require you to write code to handle negative numbers. Suggest two possible ways in which negative integers *could* be represented in the *BigInt* class.

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 coding is started before the design has been seriously considered.)

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 evolution process in this assignment (although you can for fun, if you like).

(*Questions begin on next page.*)

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 (i.e. /=).

C1. Add the following operators to the class as member functions:

```
BigInt operator+ (const BigInt& rhs) const;
BigInt operator– (const BigInt& rhs) const;
BigInt operator* (const BigInt& rhs) const;
BigInt operator/ (const BigInt& rhs) const;
```

These operators *must* be implemented trivially in terms of calls to +=, –=, etc. The implementation of each of the above operators (the code inside the braces) is expected to be about two lines.

C2. Confirm that these new operators work correctly by revising the implementation of factorial so that it uses * and – (instead of *= and –=).

C3. Replace the placeholder code for the print member function with correct code. Test your new code.

C4. Add 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. 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.

C5. Replace the placeholder code for the += and –= member functions with correct code. Test your new code.

C6. 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).

C7. 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!).

C8. Modify the code for *= so that it operates correctly for all *BigInt*s.

You should submit your code using the electronic submission system described in the *Homework Policies* handout. You are not required to include a README file with this assignment.