CS 103 Lab – V for Vector

1 Introduction

As you know, each "int" in C++ can store up to 2^{31}-1, which is about 2 billion, and each "long" can store a number up to 2^{63}-1, which is about 9 quintillion. For mathematical and cryptographical applications, it is necessary to hold much larger numbers in memory. E.g., when you make a credit card transaction online, it is common to use numbers up to 2^{2048} in size (about 617 decimal digits). You will write a class that can support such operations.

In grade school, you learned how to add two numbers by long-hand arithmetic. You were forced to memorize the 10-by-10 "addition table" that tells you the sum of any two numbers between 0 and 9. But then, *you learned an algorithm* to compute any sum of an arbitrary pair of numbers! I.e., to be able to add 3-digit numbers, you didn't have to memorize a 1000-by-1000 table. Instead you used a sequence of operations that broke everything down into simple steps.

For example, to add A = 243 and B = 85, you would write the numbers one on top of each other, and add up the columns from right to left, performing "carries" as needed:

10^2 place	10^1 place	10^0 place	
			<- carry
2	4	3	<- A : 243
	8	5	<- B : 85
			<- sum

First you added the 10^0 place, writing 8 (3+5) in the sum's 10^0 place. Then you added the 10^1 place, giving 12 (4+8); but since this is too big, you would write a 2 in the sum's 10^1 place, and carry the 1. Finally, in the 10^2 place, you added the 1 (carry), 2 (A), and 0 (B), giving 3, with the sum being 328:

10^2 place	10^1 place	10^0 place	
1			<- carry
2	4	3	<- A : 243
	8	5	<- B : 85
3	**2**	**8**	<- sum

This algorithm works for any number of digits. You will implement it in C++ for this lab.

2 What you will learn

After completing this lab you should be able to:

- Utilize the string class and the vector template class
- Implement a simple numerical algorithm
- Understand the difference between public and private access in a class
- Understand the default copy constructor

3 Background Information and Notes

The specific operations that we want you to implement are defined for you in the header **bigint.h**:

```
class BigInt {
public:
   BigInt(string s);   // convert string to BigInt
   string to_string(); // get string representation
   void add(BigInt b); // add another BigInt to this one
private:
   // whatever you need
};
```

For example, this sample program (**test1.cpp**) prints out 55 and 68:

```
BigInt a("13");
BigInt b("42");
b.add(a);                       // increase b by a
cout << b.to_string() << endl; // prints 55
b.add(a);                       // increase b by a
cout << b.to_string() << endl; // prints 68
```

We want your library to support numbers that can be arbitrarily large. Therefore, you will represent each number as a long list of digits. E.g. `BigInt a("13")` should be thought of as "create a new BigInt that contains a sequence of two digits, 1 and 3."

The number of digits in a BigInt can grow. (We'll only deal with positive numbers in this lab.) So it is extremely convenient to use the **vector<T>** class, which represents a list of values of any type **T**, which can also grow over time. The vector class has an extensive API that we encourage you to look up online. For this assignment you can get by with 4 operations:

```
vector<T>()                    // default constructor: new empty vector
void vector.push_back(T t)     // add element to end of the list
T vector[int i]                // get the element at position i
size_type vector.size()        // current size (size_type is an unsigned int)
```

For future assignments, it can be useful to use other vector operations, so please skim the documentation online. You need to **#include <vector>** to use vectors.

Here is a sample program using these operations:

```
vector<string> words;              // declare, construct
words.push_back("Hello");          // push_back: add to end
words.push_back("World");
cout << words.size() << endl;      // size: get size
cout << words[0] << " " << words[1] << endl; // [i]: access
```

As you can see, it builds up the vector from the beginning to the end. It prints out the number **2**, then **Hello World**.

What makes the most sense for our BigInt library is to use a **vector<int>** as the data member: this will permit the numerical operations involved to be as simple as possible.

It is recommended that you use **position 0** to represent the **units digit**, and **position 1** to represent the **tens digit**, et cetera. So the label "**10^K place**" on *page 1* of this lab handout means index K of your vector. This will simplify the addition algorithm although it will complicate the constructor.

If you follow this approach, the constructor **BigInt(string s)** will be responsible for:

- converting the digits (characters of **s**) to ints (elements of the vector)
- reversing the order (since a string representation of a number puts the units digit at the end, but we want it at the beginning)

To convert a character to an int, a simple approach is to use the fact that the ASCII values of the ten digit characters, obtained by casting them to int, are as follows:

```
char c   '0'  '1'   '2'   '3'   '4'   '5'   '6'   '7'   '8'   '9'
(int)c   48   49    50    51    52    53    54    55    56    57
```

So to convert a char that is a digit (e.g., **'3'**) to an int of the same value (e.g., 3), first cast to int, and then subtract 48. (Or, just subtract **'0'**.) Look up "ASCII table" online, or on page 494 of the textbook, to see more information about how different characters are represented.

It is recommended that you get the constructor and **to_string()** function working first. Then this program (**test0.cpp**) should print out "103":

```
BigInt myInt("103");
cout << myInt.to_string() << endl;
```

4 Procedure

4.1 Provided Code

Download the files for this lab: the **bigint.h** header and some test programs.

```
mkdir ~/lab-bigint
cd ~/lab-bigint
wget http://bytes.usc.edu/files/cs103/lab-bigint.tar
tar xvf lab-bigint.tar
```

4.2 [3 pts.] constructor, to_string()

Implement the constructor and print function for the BigInt class.

- We have provided **bigint.h** that is mostly complete:
  - add the necessary private data member to the definition.

- Create **bigint.cpp** and fill in the actual definitions for the constructor and the to_string() function.

- Run **make test0** and **./test0**, which should print out **103**.

4.3 [5 pts.] add()

Implement the add() function.

- **Before** starting to code, write up pseudocode for how the addition should work. How many times should it loop? How will you represent the "carried" number? Does it use **push_back(),** which appends to the end of a vector?

- Note that inside of **void BigInt::add(BigInt b)**, you are allowed to access the private variables of **b**. This is because your code is part of the definition of BigInt. Privacy prevents other classes from seeing inside, but not other objects of the same class.

- In order to get your numbers to line up, you might like to add some extra zeroes to one vector or the other. Note that inside of **void BigInt::add(BigInt b)**, you are allowed to change the data members of **b**. This is safe because C++ uses **pass-by-value**, even with objects: you are only changing a *copy* of **b**.

- Run **./test1**, which should print 55 and 68. Create and run a test of your own design that checks whether your code is correct in other cases (carrying, different lengths, etc). Look at the other sample programs provided.

4.4 [2 pts.] Discussion

Write a program **badaccess.cpp** that, when compiled (you can use **make badaccess**), generates this error message:

error: '???' is a private member of 'BigInt'

(The **???** will depend on your implementation.) Then, change the data member visibility to public and make this error message go away, to confirm that was the source of the problems. Finally, change the visibility back to private (which is best practice). Show your TA that you are able to generate this error message.

Demonstrate all of your work to your TA/CP.

4.5 [1 pts.] Extra Credit: All Your Base

There is nothing special about the number 10, except that the median human has that many fingers. Add a new constructor

```
BigInt(string s, int base)
```

that creates a BigInt operating in the given base. Add a new data member if needed. Your newly extended class should work with any base between 2 and 36, using the digits 0, 1, …, 9, A, B, C, …, Z. For example:

```
BigInt first("DADCAFE", 16); // hexadecimal
BigInt second("AAABEEF", 16);
first.add(second);
cout << first.to_string(); // prints 185889ED
```

To help the user, **throw a runtime error** when a user tries to add two different-base numbers.

```
#include <stdexcept>
using namespace std;

void BigInt::add(…)
{
 if( /* bases don't match */ )
   throw runtime_error("Error - different bases");
 else { /* bases do match so perform the operation */ }
}
```