Programming Project 03 - Overloading

CS200 - Fall 2024-2025

Due Date: 11:55 pm, Nov 10, 2024
Lead TA: Faheem Shehzad

Contents

1 Overview

1.1 Plagiarism policy

- Students must not share or show program code to other students.
- Copying code from the internet is strictly prohibited.
- Any external assistance, such as discussions, must be indicated at the time of submission.
- Course staff reserves the right to hold vivas at any point during or after the semester.
- Any act of plagiarism will be reported to the Disciplinary Committee.
- In this assignment, LLM's may be used to debug.

1.2 Submission instructions

- Zip the entire folder and use the following naming convention: `PA3_<rollnumber>.zip`
  For example, if your roll number is 26100181, your zip file should be: `PA3_26100181.zip`
- All submissions must be uploaded on the respective LMS assignment tab before the deadline.
  Any other forms of submission (email, dropbox, etc) will not be accepted.

1.3 Aims and learning outcomes

This programming assignment consists of one part.

Aim:
The aim of this programming project is to provide students with a thorough understanding of operator overloading in C++. Through the implementation of a custom `BigNum` class, you will gain practical experience in overloading various operators to handle large integers and perform arbitrary-precision arithmetic.

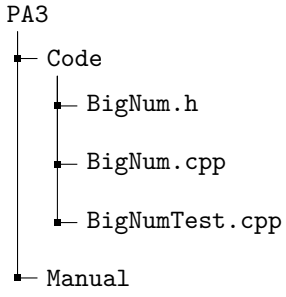
Learning outcomes:

- Learn the syntax and semantics of overloading operators in C++.
- Understand how operator overloading can make classes more intuitive and easier to use.
- Explore techniques for managing large numbers and performing complex arithmetic operations.
- Gain practical experience in implementing and testing overloaded operators to enhance class usability and functionality.

1.4 Grading distribution

Assignment breakdown	
Code Quality	(5 points)
Constructors	(10 points)
Comparison Operators	(10 points)
Arithmetic Operators	(30 points)
Digit Product Operator	(15 points)
Unary Inc/Dec Operator	(5 points)
Compound Assignment Operators	(5 points)
Safe Indexing Operators	(10 points)
I/O Stream Operators	(10 points)
Hidden Test Cases	(10 points)
Total	**110 points**

1.5 Starter code



2 Code Quality

You are expected to follow good coding practices. We will manually look at your code and award points based on the following criteria:

- Readability: Use meaningful variable names as specified in the PEP 8 standards and lectures. Assure consistent indentation. We will be looking at the following:
  - **Variables, Functions, and Methods:** Use lowercase letters and separate words with underscores. For example, my_variable, calculate_sum()
  - **Constants:** Use uppercase letters and separate words with underscores. For example, MAX_SIZE, PI
  - **Classes:** Use CamelCase (capitalize the first letter of each word without underscores). For example, MyClass, CarModel
- **Modularity and Reusability:** Break down your code into functions with single responsibilities.
- **Code Readability:** You should make sure that your code is readable.
- **Documentation** (Optional): Comment your code adequately to explain the logic and flow.
- **Error Handling:** Appropriately handle possible errors and edge cases.
- **Efficiency:** Write code that performs well and avoids unnecessary computations.

3 Assignment Overview

You are the lead software engineer on the Starship Endeavor, humanity's most advanced interstellar vessel, tasked with exploring distant star systems. As the ship ventures further from Earth, the challenges become more complex, requiring sophisticated calculations to ensure the crew's safety and the mission's success.

3.1 The Problem:

During a critical mission to navigate through a newly discovered wormhole, the ship's navigation system has encountered a severe limitation. The calculations required to accurately predict the wormhole's stability and path involve extremely large numbers, far beyond the capacity of the ship's standard computational systems. This poses a significant risk, as inaccurate calculations could result in the ship being lost in space.

Typically, the size of integers is limited by the maximum value that can be stored in the data type provided by the language, such as INTMAX which is usually 2,147,483,647 for a 32-bit integer. This limitation poses significant challenges when dealing with large calculations that require precision beyond this range.

To solve this issue, you propose a BigNum class, capable of handling arbitrary-precision arithmetic. This custom class will allow the ship's computer to perform the necessary calculations with numbers of virtually any size, ensuring accurate navigation through the wormhole.

4 Implementation

You need to write a class BigNum which should be able to hold integers of any size. No matter how big an integer is, your class should be able to handle it. The definition of class will go in BigNum.h and the methods of the class will be defined in the file named BigNum.cpp

4.1 General Overview

While there is no set requirement on how you must define the member functions or store the digits within your BigNum class, **you must choose between the vector library and dynamic arrays to do this project**. However, there are a few recommendations to help guide you:

- **Test cases rely on working constructors and a working comparison operator, ensure these two functionalities work otherwise you will obtain a zero for this project**
- You are free to define as many helper functions as you wish. This includes how you choose to handle arithmetic operations, comparisons, and input/output.
- It is advisable to use the vector library to store the digits. Using vector provides dynamic resizing and ease of use, which simplifies the implementation of the BigNum class.
- Alternatively, you can use dynamic arrays to store the digits. If you choose this method, ensure that you manage memory correctly, which includes creating your own destructors to prevent memory leaks.
- Our final grading will be based on the provided test cases alongside some hidden test cases. Ensure that your implementation passes all the test cases to meet the requirements of the assignment especially corner cases.

NOTE: While you may store data in any variable type, some test cases assume digits are stored as characters. In such cases you may have to convert your chosen data type accordingly.

4.2 Constructors (10 Marks)

4.2.1 C++ String Constructor

Write a constructor that takes in a C++ string, and stores the corresponding number in your BigNum class. The following commands are all valid and should store the numbers:

```
string numString = "3276473648369753";
BigNum num1(numString);
BigNum num2 = BigNum(numString);

```

4.2.2 Integer Constructor

Similarly write a constructor that takes an integer and stores it as a BigNum so that you may do the following:

```
BigNum num4 = 5478;
Bignum num5(7347);

```

4.2.3 Default Constructor

The default constructor with no parameter passed should create a BigNum object with zero stored within it

4.3 Comparison Operator (10 Marks)

With constructors for the BigNum class written, design comparison operators (== and !=) for the class. Specifically, make sure that the following comparisons are possible

- A BigNum with a BigNum
- A BigNum with an int
- An int with a BigNum

The following are valid sample comparisons:

```
BigNum num4 = 2213;
Bignum num5(9237);
num4 == num5
num5 == 90
3232 != num4

```

4.4 Arithmetic Operators (30 Marks)

4.4.1 Addition Operator (10 Marks)

The addition operator must be capable of managing integers that exceed the typical storage limits of standard data types, thus ensuring the correct computation of sums involving such large values. This involves designing an algorithm that accurately adds each digit of the big numbers, accounting for carry-over values between digit positions. A sample addition that utilizes the extended storage capabilities may be:

```
BigNum num4("393249828942448932")
BigNum num5("79489389283982294892");
BigNum num6 = num4 + num5
//Should return a BigNum containing 79882639112924743824

```

4.4.2 Subtraction Operator (10 Marks)

The subtraction operator must be capable of subtracting two BigNum objects, thus ensuring the correct computation of differences involving such large values. This involves designing an algorithm that accurately subtracts each digit of the big numbers, accounting for borrowing between digit positions. A sample subtraction may be:

```
BigNum num1("98765432109876543210");
BigNum num2("12345678901234567890");
BigNum num3 = num1 - num2;
//Should return a BigNum containing 86419753208641975320

```

4.4.3 Multiplication Operator (10 Marks)

Similarly, the multiplication operator must be capable of managing large integers, ensuring the correct computation of products involving such large values. This involves designing an algorithm that accurately multiplies each digit of the big numbers, taking into account the carry-over values between digit positions. An example is provided below:

```
BigNum num1("123456789");
BigNum num2("987654321");
BigNum num3 = num1 * num2;
//Should return a BigNum containing 121932631112635269

```

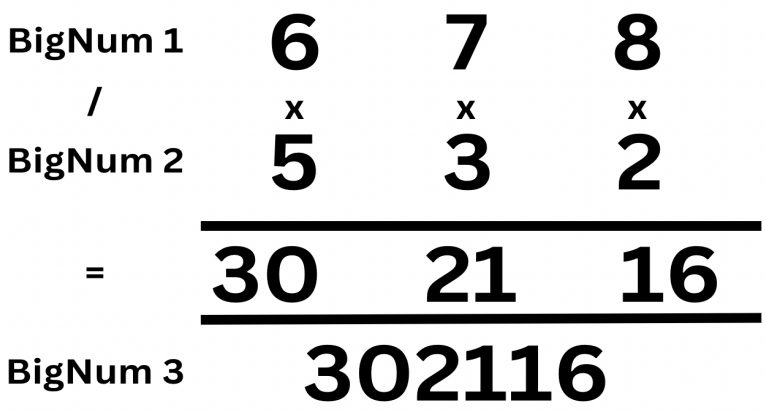
NOTE: Do not try to hard code, there will be hidden test cases. Also keep in mind corner cases and make sure to always remove leading zeros

4.5 Digit Product Operator (15 Marks)

As the Starship Endeavour journeys deeper into the uncharted reaches of space, it encounters increasingly complex challenges that demand advanced computational capabilities. One such challenge involves performing intricate operations on the digits of large numbers—specifically, calculating the products of corresponding digits from different numerical sequences.

To tackle this problem, you propose enhancing the BigNum class with a specialized **"Digit Product Operator."** This operator will compute the product of corresponding digits from two BigNum objects and seamlessly concatenate the results to form a new sequence. By doing so, it enables the ship's computer to handle the sophisticated calculations required for navigating unexplored cosmic territories.

For example suppose we have two numbers 678 and 532 and we wish to apply the **Digit Product Operator** (/) on them, the resultant calculation would be done as follows:



The Digit Product Operator will be tested by overloading the "/" symbol for the BigNum class. An example is provided below:

```
BigNum num1("7820543");
BigNum num2("9731223");
BigNum num3 = num1 / num2;
//Should return a BigNum containing 6356601089

```

NOTE: For the purposes of testing this operator, the Digit Operator will only be tested on numbers of the same length, you do not need to cater for differing lengths.

4.6 Unary Increment and Decrement (5 marks)

Expand your BigNum class by implementing the powerful unary increment (++) and decrement (–) operators. You are required to provide both prefix and postfix variations. This means your class will seamlessly handle operations like ++i and i++, allowing more flexibility in your operations.

```
BigNum num1("4400");
BigNum num2("2000");
num1++;
++num1;
num2--;
--num2;
//The following operations are all valid

```

4.7 Compound Assignment Operators (5 marks)

Implement the compound assignment operators to enhance the arithmetic capabilities of your BigNum class. These operators should be designed to handle large integers efficiently and maintain the integrity of the BigNum class. You are required to implement the following:

```
+=
-=
*=

```

4.8 I/O Stream Operators (10 Marks)

Overload the input (‘»‘) and output (‘«‘) stream operators for the BigNum class. Overloading these operators will allow integration with standard C++ stream operations, allowing BigNum objects to be easily read from and written to input/output streams. This functionality is crucial for testing and debugging.

4.8.1 Input Stream Operator (5 Marks)

The ‘»‘ operator should read a big number from an input stream and store it in a BigNum object.

```
BigNum num;
cin >> num;  // Should read a BigNum from standard input
```

4.8.2 Output Stream Operator (5 Marks)

The '«' operator should output the big number to the terminal in a human-readable format.

```
BigNum num;
cout << num; // Should write the BigNum to standard output
```

4.9 Indexing Operator (10 Marks)

To enable viewing or modifying a specific digit in your BigNum, you need to implement the indexing '[]' operator. This operator should take an 'int' as the index and provide access to individual digits of the BigNum, allowing both retrieval and modification at the specified position.

- The indexing operator should return the digit at the given position when used for retrieval.
- It should allow modification of the digit at the specified position when used for assignment.
- If the specified index does not exist, the operator should return a static char variable holding '\0'

```
BigNum num("123456789");
char digit = num[3];  // Should return the 4th digit
num[3] = '9';         // Should modify the 4th digit to '9'
```

TIP: Reading and writing using the [] operator are two separate overloads

5 Testing

To ensure the correctness and robustness of your BigNum class implementation, testing will be conducted using the provided test file `BigNumTest.cpp`. This file contains a series of predefined test cases that will evaluate various functionalities of your class.

You may compile the executable file using:

```
g++ -o a BigNum.cpp BigNumTest.cpp
```

In addition to the visible test cases, 10 marks will be allocated for hidden test cases. These hidden cases are designed to:

- Prevent hard coding solutions.
- Ensure your implementation can handle a variety of scenarios correctly.

All testing will be done on Docker. Please note that we do NOT require you to use Docker. This is for your convenience only. There are some instances in which students have reported that the code runs on their system but not on the grading server. This is because of the difference in the environment. If you are confident that your code will run on the grading server, you do not need to use Docker. However, if you are not sure, we recommend that you use Docker to test your code.

.

Good luck and Start Early!