OOP Assignment (inf_int Class)

Team #8

유용민
설지환
안지완
이의제
이주형
채승운
최동욱

Index.

List of team members

Name	ID	Role
유용민		
설지환		
안지완		
이의제		
이주형		
채승운		
최동욱		

Introduction

Our team's project is mainly about "making a class that can maintain an infinite length of number, having basic operators of addition, subtraction, and multiplication." As a basic 'Integer class' given by C++ can only maintain at most approximately 2 billion, we should create a customized integer class, named as 'inf_int.' In this report, we will explain how we made a brief concept, specification, and implementation of this customized 'inf_int' class.

Concept of the 'inf_int'

As we explained above, a basic integer class (a.k.a 'int') can only maintain at most 2 billion, which means that assigning above 2 billion to the 'int' will make an overflow of the program, leading to the fatal error to the program. To prevent this, we decided to use 'string' instead. A string type can maintain an infinite length of the string (actual length is 4.2billion, which is still close to infinite, 10^(4.2billion)). A brief abstraction of the class 'inf_int' looks like this:

```
class inf_int
{
private :
    char* digits;
    unsigned int length;
    bool thesign;

public :
    inf_int();
    inf_int(int);
    inf_int(const char* );
    inf_int(const inf_int&);
    ~inf_int();

    inf_int& operator=(const inf_int&);

    friend bool operator==(const inf_int& , const inf_int&);
    friend bool operator!=(const inf_int& , const inf_int&);
    friend bool operator>(const inf_int& , const inf_int&);
    friend bool operator<(const inf_int& , const inf_int&);

    friend inf_int operator+(const inf_int& , const inf_int&);
    friend inf_int operator-(const inf_int& , const inf_int&);
    friend inf_int operator*(const inf_int& , const inf_int&);
    friend ostream& operator<<(ostream& , const inf_int&);
};
```

Functionality

A customized class ‘inf_int’ has a responsibility not only to operate basic arithmetic functions properly, but also to perform comparison operator and I/O functions. Thus, we overloaded the basic operators to get our ‘inf_int” class as the parameters, declaring them as ‘friend’ functions to give them permission to access the internal data (digits, length, thesign). By doing this, operators will access the char-type digits and perform the functions by reading each digit. Now we need to specify the implementations.

Clarifying the functions with a chart, it will look like this:

Classification	Symbol	Description
Constructor	(*construct or symbol*)	Initializing internal members when creating object
Compare	>, < , ==, !=	Comparing inf_int classes’s values
Arithmetic operator	+, -, *, /, %	Performing arithmetic operator (We will perform this by implementing ‘Add’, ‘Sub’ method)
Assign	=	Assigning values to the inf_int object
I/O Function	<<	Output function of the inf_int class

Specifying each implementations

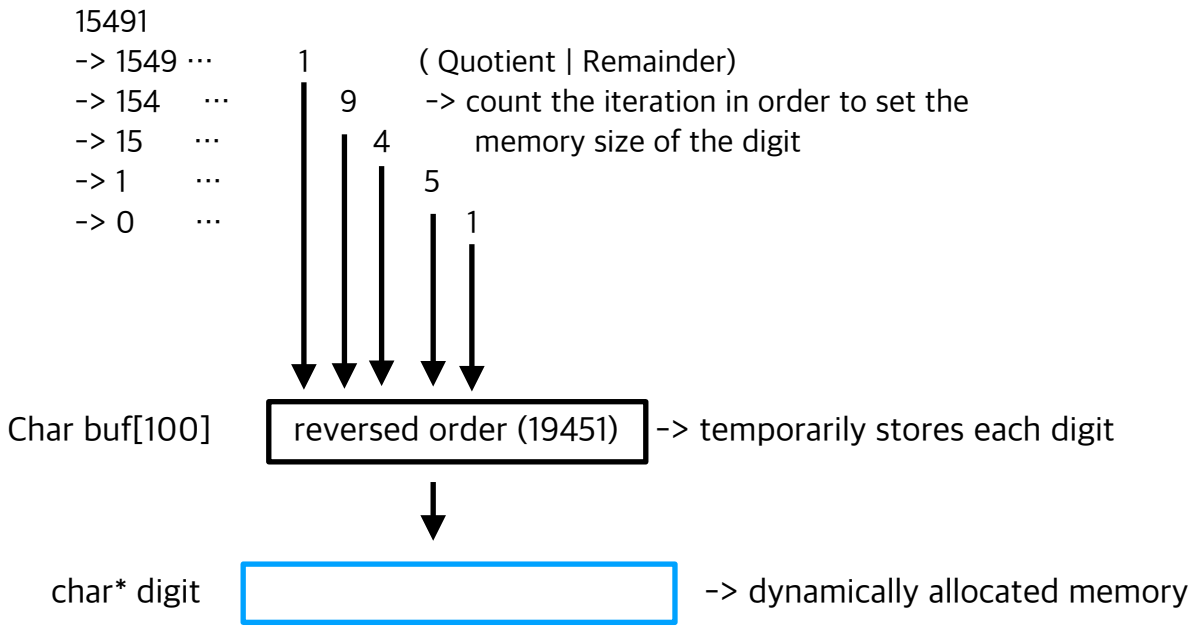
Constructors

Before explaining the implementation of the function, we first need to specify the structure of the internal data (digits, length, thesign). As explained above, we should store digits in char type. However, since we cannot figure out the exact length of the number before declaring it, we should not use the static array. Instead, we should use

‘dynamic allocation.’ This concept will be used in the constructor of the ‘inf_int’ class, as the internal members are initialized when the constructor is called.

Four types of constructors are declared in the ‘inf_int’ class : One that gets an integer as a parameter, one that gets a string, another as a copy constructor, and the other with none parameter.

Starting with the first constructor (integer), we should beware that the structure of a string type, and the integer type is different. As we assumed that the length of the digits is infinite, we cannot simply cast the integer to the string type. If doing so, it won’t properly perform the arithmetic functions with another ‘inf_int’ object having infinite length of digits. Considering this case, we should divide the integer with 10 to get each digit, which will be saved in reversed order for the reason that is explained by the diagram below.



Dividing 10 repeatedly and extracting each remainder will produce a single digit of the target integer, which will be stored in the temporary char array (buf) but in reversed order. After that, the data of the temporary char array will be moved to the digit, which is initialized with dynamic memory allocation method. What makes such process possible is the loop of the division. Counting the loop iteration that performs division until the quotient becomes zero, will allow the program to get the length of the target integer, and by using the length, we can dynamically allocate the memory with such size and initialize the internal member, “length.” The sign will be equal to the target integer.

The code will be like this:
Once the constructor with integer parameter is created, the other types of constructor is not that complicated, as the data type is same as the internal member, ‘digit.’ The one with string parameter just needs some process to read each character and copy into the ‘char* digit,’ getting the length of the target string which will be same as the value of the member ‘unsigned int length,’ and deciding the sign by reading the first character.

```
inf_int::inf_int(int n)

{
    char buf[100];
    if (n < 0)
    {
        this->thesign = false;
        n = -n;
    }
    else
    {
        this->thesign = true;
    }
    int i = 0;
    while (n > 0)
    {
        buf[i] = n % 10 + '0';
        n /= 10;
        i++;
    }

    if (i == 0)
    {
        new (this) inf_int();
    }
    else
    {
        buf[i] = '\0';
        this->digits = new char[i + 1];
        this->length = i;
        strcpy(this->digits, buf);
    }
}
```

The only thing we have to beware is that, the digits should be stored in reverse value in order to perform other functions. Thus, we also have to copy each letter of the target string and store reversely.

Copy constructor only needs a process to read the target ‘inf_int’ object’s internal member and literally copy each data into the internal members. Since the digits are already stored in reversed order in all ‘inf_int’ objects, we could just simply copy its string and paste it to the digits.

A default constructor (that doesn’t need any parameter) will have a simple process to initialize the internal members. In this project, we will initialize the digit as zero, additionally putting the null at the end of the digit array to indicate the end of the string, and set the length as 1 and bool sign as true.

Code will look like this :
Additionally, since we allocated memory dynamically for storing digits, we should free it if the object is to be destructed. Thus, we should declare the destructor so as the digits to be freed by the destructor.

```
inf_int::inf_int()
{
   this->digits = new char[2];
   this->digits[0] = '0';
   this->digits[1] = '\0';
   this->length = 1;
   this->thesign = true;
}

inf_int::inf_int(const char *str)
{
   unsigned int i;
   if (str[0] == '-')
   {
      this->thesign = false;
      this->length = strlen(str) - 1;
      this->digits = new char[length + 1];
      for (i = length - 1; i >= 0; i--)
      {
         digits[i] = str[length - i];
      }
   }
   else
   {
      this->thesign = true;
      this->length = strlen(str);
      this->digits = new char[length + 1];
      for (i = length - 1; i >= 0; i--)
      {
         digits[i] = str[length - i - 1];
      }
   }
}
inf_int::inf_int(const inf_int &a)
{
   this->digits = new char[a.length + 1]
   strcpy(this->digits, a.digits);
   this->length = a.length;
   this->thesign = a.thesign;
}
inf_int::~inf_int()
{
   delete digits;
}
```

Comparator

The next step is specifying the comparators. According to the abstraction class we made before, our class should handle four comparators (>, <, ==, !=). Simply using the basic operators without overloading will make an unexpected result, as our 'inf_int' class has an infinite length of digits. To make it possible, we should compare in a different way.

Assume that there are two numbers of different length, or different sign, and we are using the relational operators(>, <). If the length of the two numbers we are to compare is different, simply just picking up the longer one will get the proper result. Or, if the sign of the two numbers are different, picking up the positive one will be right.

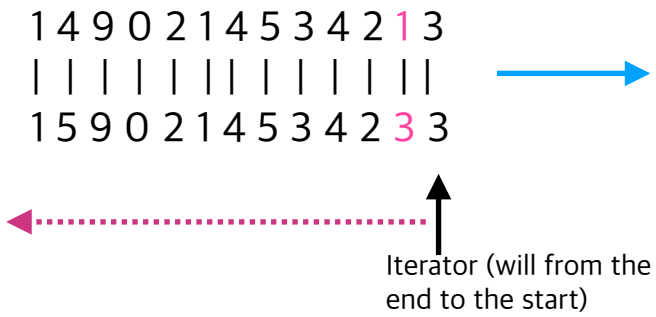
Moreover, assume that we are using the equality operators. We could just compare if the length, and the sign of two numbers are same. The problem occurs when the two numbers **have the same length, and the same sign.** In this case, we should compare each digit one by one.

```
1 4 9 0 2 1 4 5 3 4 2 1 3
| | | | | | | | | | | | |
1 5 9 0 2 1 4 5 3 4 2 3 3
<————————————————————>
Least Significant Digit                Most Significant Digit
```

Reminding the principle of storing digits, we stored digits in reversed way in order to perform functions with infinite length of number, which means that for example, if we intended to save '15032' in inf_int object, the digits will be saved in order of '23051.'

This is the point we should make it clear, that the most significant digit (or the biggest value) is stored at the end of the digit array, whereas the least significant digit is stored at the start point of the array. Thus, we should prioritize the most significant digit while performing the comparator.

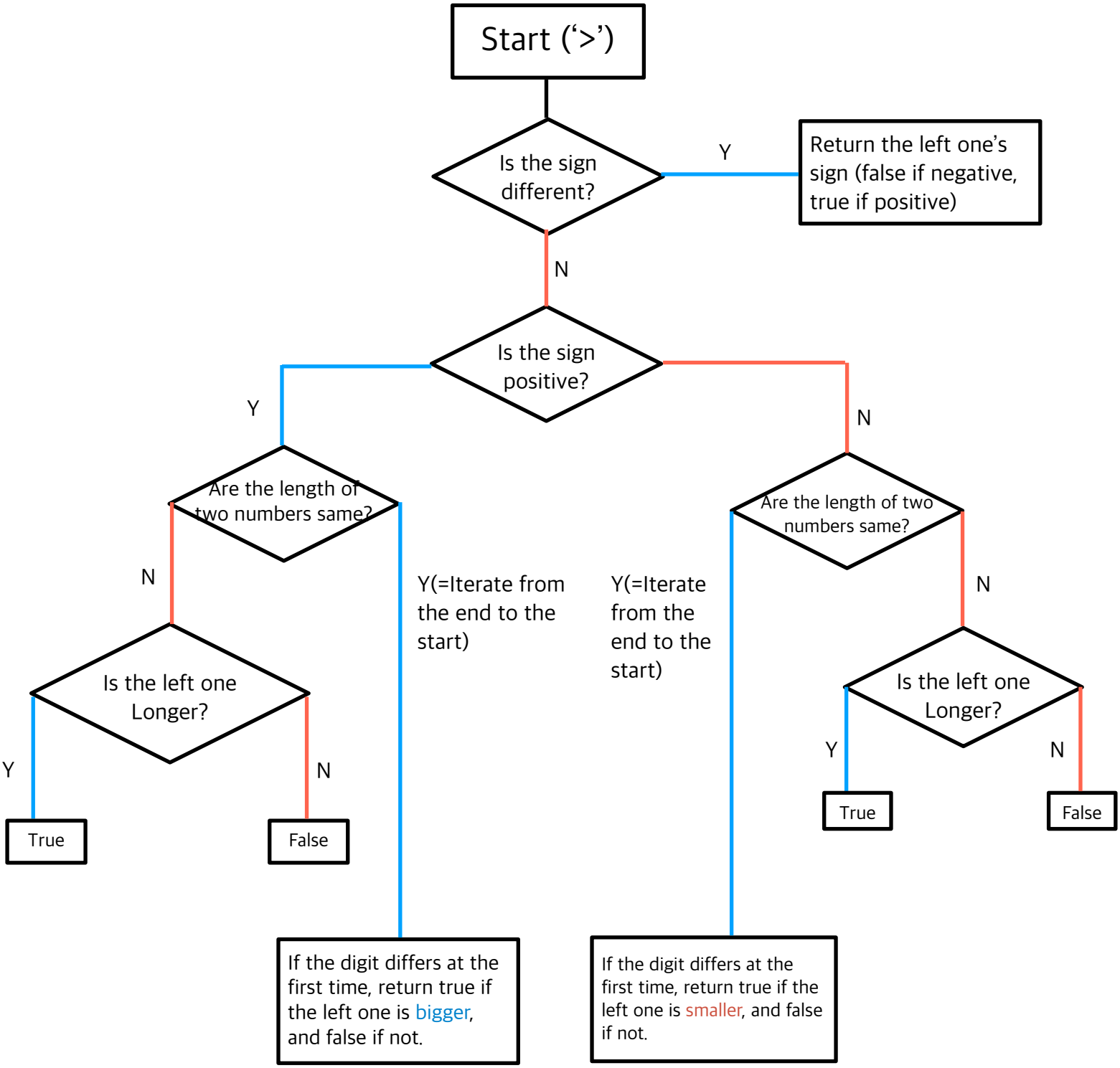
According to the example above, the digit of the number at the right side is bigger than of the number above. Therefore, the comparator will make a result by looking up the pink-highlighted digit.



If the numbers are positive :
-> 1 < 3 => the below is bigger

If the numbers are negative :
-> -1 > -3 => the above is bigger

Representing these cases by diagram will be like this (in this case, '>') :
(Shown in next page)



The other relational operator (<), and the equality operators(==, !=) will follow this principle similarly. The relational operator will perform the principle above in reversed manner. The equality operators will only have to consider if the digits of two numbers are different when the two number's length and sign are same.

Looking up some glimpse of this comparator's code, it will look like this :

```
bool operator>(const inf_int &a, const inf_int &b)
{
    if (a.thesign != b.thesign)
        return a.thesign;

    if (a.thesign) // a가 양수일 때
    {
        if (a.length > b.length) // 길이가 길면 무조건 큼
            return true;
        else if (a.length < b.length) // 길이가 작으면 무조건 작음
            return false;
        else
        {
            for (int i = a.length - 1; i >= 0; i--) // 길이가 같다면 하나하나 비교 시작 (12345면 배열엔 54321이 담겨있는 상태)
            {
                if (a.digits[i] > b.digits[i])
                    return true;
                else if (a.digits[i] < b.digits[i])
                    return false;
                else
                    continue; // 같은 자릿수에 숫자까지 동일하다면 패스
            }
        }
    }
    else // a가 음수일 때
    {
            /…/
    }
}
```

We can know that the structure of the comparator's code is just same as the structure of the diagram above, comparing the sign first, then the length second, and iterating if the sign and the length is same. Closely looking up the iterating part, the 'for loop' starts from the end of the digit array. If the different digit is found for the first time, the program breaks the loop and returns the result, depending on the sign of the target numbers.

Before we move onto the next part (arithmetic operators), we should make one thing clear. Why did we use references in the parameter of the operator? Reminding what we learned in the past lecture, we learned that there are two types of copying. One is a 'shallow copy,' and the other is 'deep copy.' Shallow copy is a type of copy that only copies its reference, not the whole object, which won't make another object in the program, while deep copy is a type of copy that copies the object itself and creates another identical object, which will consume additional memory. What we are copying is a 'inf_int' class, which has an infinite length of digit. Hence, creating another object would create enormous amount of memory which is not efficient at all.

To avoid this, we use 'pass by reference' to perform shallow copy, just to copy its references without creating another 'big-size' object. Not only we can save memory, but also we can ensure that the digits are not changed, because the function literally points the target object so the values will be exactly passed into the function.

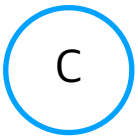
Arithmetic operator

The main feature of our project is arithmetic operators. These are harder than the functions explained above, since there are more things we should consider when performing arithmetic operators. For example, there exists a 'carry' when doing addition, and there needs another process to convert the 'char' into to the 'integer' to perform calculation.

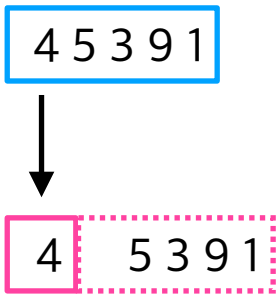
Simply just overloading operators won't make a proper result. First, if a carry occurs, the length of the result becomes longer than those of the operands, which means that there should be an additional memory space to maintain the lengthened digits, and what makes even worser is we should classify the cases that which cases makes the length of the result become longer. This should be considered when doing subtraction, multiplication, and division, which will make the length of the result more variably. Hence, we need to use dynamic memory allocation.

Start with add(operator '+'). An additional thing we have to consider is 'carry.' Whether the carry occurs or not will determine the needs of additional dynamic memory allocation.

Brief Idea of Addition (e.g. 19354+82912):
- Create an empty 'inf_int' object, c.

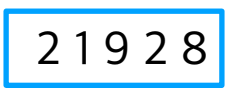


- Add a to c, but by a single digit. Memory spaces will be allocated to c in a single unit with loop.

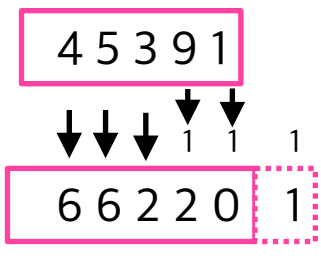


Assuming that the inf_int has a infinite length, we cannot allocate memory at once, so we should allocate memory one by one. Digits will be stored in reversed order.

- Add b to c. 'a' is stored in c in advance, so the actual addition occurs at this process.



Digits of the operands will be converted into the integers, and will be added one by one (with loop).



If carry occurs while performing addition, it will be stored in the next index of the array in advance, and in next iteration, addition will be performed with the carry

If there's another carry at the end of the digit, additional memory allocation will be performed to store carry.