**Lab 05**

**Object Oriented Programming Lab**

**Common Solution**

**16 Marks**

**Challenge-1:** *Big Number*

**BigNumber.h**

#ifndef BIG\_NUMBER\_H

#define BIG\_NUMBER\_H

#include<iostream>

using namespace std;

enum Comparison

{

EQUAL, SMALL, LARGE

};

class BigNumber

{

char\* number;

int numberLength;

int getStrLength(const char\*);

void copyStr(const char\*, char\*);

public:

BigNumber(const char\*);

BigNumber(const BigNumber&);

~BigNumber();

void setNumber(const char\* num);

BigNumber add(BigNumber&)const;

BigNumber subtract(BigNumber&)const;

BigNumber multiply(BigNumber&)const;

void print() const;

Comparison compare(BigNumber&)const;

};

#endif // !BIG\_NUMBER\_H

**BigNumber.cpp**

#include "BigNumber.h"

//Private Functions:

int BigNumber::getStrLength(const char\* str)

{

int length = 0;

while (str[length] != '\0')

{

length++;

}

return length;

}

void BigNumber::copyStr(const char\* source, char\* destination)

{

if (source == nullptr)

{

return;

}

int i = 0;

while (source[i] != '\0')

{

destination[i] = source[i];

i++;

}

destination[i] = '\0';

}

//Public Functions:

BigNumber::BigNumber(const char\* input)

{

if (input == nullptr)

{

number = new char[1];

number[0] = '\0';

numberLength = 0;

return;

}

numberLength = getStrLength(input);

number = new char[numberLength + 1];

copyStr(input, number);

}

BigNumber::BigNumber(const BigNumber& ref)

{

numberLength = ref.numberLength;

number = new char[numberLength + 1];

copyStr(ref.number, number);

}

BigNumber::~BigNumber()

{

delete[]number;

number = nullptr;

numberLength = 0;

}

void BigNumber::setNumber(const char\* num)

{

delete[] number;

numberLength = getStrLength(num);

number = new char[numberLength + 1]; // +1 for null terminator

copyStr(num, number);

}

void BigNumber::print() const

{

cout << number;

}

BigNumber BigNumber::add(BigNumber& other) const

{

// Find the maximum length among the two numbers

int maxLength = (numberLength > other.numberLength) ? numberLength : other.numberLength;

// Allocate memory for the result, add 1 for potential carry

char\* result = new char[maxLength + 1];

result[maxLength] = '\0'; // Null-terminate the result

// Initialize carry

int carry = 0;

// Traverse both numbers from right to left and add digits

int i = numberLength - 1;

int j = other.numberLength - 1;

int k = maxLength - 1;

while (i >= 0 || j >= 0)

{

// Get digits from both numbers or use 0 if index is out of range

int digit1 = (i >= 0) ? (number[i] - '0') : 0;

int digit2 = (j >= 0) ? (other.number[j] - '0') : 0;

// Add digits and carry

int sum = digit1 + digit2 + carry;

// Update carry for the next iteration

carry = sum / 10;

// Store the result digit

result[k--] = (sum % 10) + '0';

// Move to the next digit in both numbers

i--;

j--;

}

// If there is a remaining carry, prepend it to the result

if (carry > 0)

result[k] = carry + '0';

// Create a new BigNumber object from the result

BigNumber sumNumber(result);

// Delete the dynamically allocated memory

delete[] result;

return sumNumber;

}

Comparison BigNumber::compare(BigNumber& ref)const

{

if (numberLength < ref.numberLength)

return SMALL;

else if (numberLength > ref.numberLength)

return LARGE;

else

{

for (int i = 0; i < numberLength; ++i)

{

if (number[i] < ref.number[i])

return SMALL;

else if (number[i] > ref.number[i])

return LARGE;

}

return EQUAL;

}

}

BigNumber BigNumber::subtract(BigNumber& other) const {

int maxLen = (numberLength > other.numberLength) ? numberLength : other.numberLength;

char\* result = new char[maxLen + 1];

result[maxLen] = '\0';

int carry = 0;

int i = numberLength - 1;

int j = other.numberLength - 1;

int k = maxLen - 1;

while (i >= 0 || j >= 0)

{

int digit1 = (i >= 0) ? number[i] - '0' : 0;

int digit2 = (j >= 0) ? other.number[j] - '0' : 0;

int diff = digit1 - digit2 - carry;

if (diff < 0)

{

diff += 10;

carry = 1;

}

else {

carry = 0;

}

result[k--] = diff + '0';

i--;

j--;

}

// Skip leading zeros

int startIdx = 0;

while (result[startIdx] == '0' && startIdx < maxLen - 1)

{

startIdx++;

}

BigNumber resultBN(result + startIdx);

delete[] result;

return resultBN;

}

// Multiplication function

BigNumber BigNumber::multiply(BigNumber& other) const {

// Get the lengths of the two numbers

int len1 = numberLength;

int len2 = other.numberLength;

// Calculate the maximum possible length of the result

int maxLen = len1 + len2;

// Create an array to store the intermediate result

int\* result = new int[maxLen] {0};

// Perform multiplication digit by digit

for (int i = len1 - 1; i >= 0; i--)

{

for (int j = len2 - 1; j >= 0; j--)

{

int mul = (number[i] - '0') \* (other.number[j] - '0');

int sum = mul + result[i + j + 1]; // Add to the current position

result[i + j + 1] = sum % 10; // Store the least significant digit

result[i + j] += sum / 10; // Carry over to the next position

}

}

// Find the first non-zero digit

int idx = 0;

while (idx < maxLen && result[idx] == 0)

{

idx++;

}

// If all digits are zero, return "0"

if (idx == maxLen)

{

return BigNumber("0");

}

// Convert the result array to a char array

char\* resStr = new char[maxLen - idx + 1];

for (int i = idx; i < maxLen; i++) {

resStr[i - idx] = result[i] + '0';

}

resStr[maxLen - idx] = '\0';

// Create a BigNumber object with the result string and return

BigNumber product(resStr);

delete[] result;

delete[] resStr;

return product;

}