**Lab 08**

**Object Oriented Programming Lab**

**Individual Solution**

**Challenge-1:** *Bounded Integral Type*

**BoundedInteger.h**

#ifndef BIG\_NUMBER\_H

#define BIG\_NUMBER\_H

#include<iostream>

using namespace std;

class BigNumber

{

int getStrLength(const char\* str);

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

int compare(const BigNumber& ref)const;

char\* number;

int numberLength;

public:

BigNumber(const char\*);

BigNumber(const BigNumber&);

~BigNumber();

void setNumber(const char\* num);

void print() const;

BigNumber operator = (const BigNumber&);

BigNumber operator + (const BigNumber&) const;

BigNumber operator - (const BigNumber& other) const;

BigNumber operator \* (const BigNumber& other) const;

bool operator == (const BigNumber& other) const;

bool operator > (const BigNumber& other) const;

bool operator < (const BigNumber& other) const;

bool operator >= (const BigNumber& other) const;

bool operator <= (const BigNumber& other) const;

bool operator != (const BigNumber& other) const;

};

#endif // !BIG\_NUMBER\_H

**BoundedInteger.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';

}

int BigNumber::compare(const BigNumber& other) const

{

if (numberLength < other.numberLength)

return 3;

else if (numberLength > other.numberLength)

return 2;

else

{

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

{

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

return 3;

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

return 2;

}

return 1;

}

}

// 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];

copyStr(num, number);

}

void BigNumber::print() const

{

cout << number;

}

BigNumber BigNumber::operator=(const BigNumber& other)

{

if (this == &other)

{

return \*this;

}

delete[] number;

numberLength = other.numberLength;

number = new char[numberLength + 1];

copyStr(other.number, number);

return \*this;

}

BigNumber BigNumber::operator+(const BigNumber& other) const

{

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

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

result[maxLength] = '\0';

int carry = 0;

int i = numberLength - 1;

int j = other.numberLength - 1;

int k = maxLength - 1;

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

{

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

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

int sum = digit1 + digit2 + carry;

carry = sum / 10;

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

i--;

j--;

}

if (carry > 0)

result[k] = carry + '0';

BigNumber sumNumber(result);

delete[] result;

return sumNumber;

}

BigNumber BigNumber::operator-(const BigNumber& other) const

{

if (compare(other))

return BigNumber("0");

int len1 = numberLength;

int len2 = other.numberLength;

int maxLen = (len1 > len2) ? len1 : len2;

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

result[maxLen] = '\0';

int borrow = 0;

int i = len1 - 1;

int j = len2 - 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 - borrow;

if (diff < 0)

{

diff += 10;

borrow = 1;

}

else

{

borrow = 0;

}

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

i--;

j--;

}

int startIndex = 0;

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

{

startIndex++;

}

if (startIndex == maxLen)

{

delete[] result;

return BigNumber("0");

}

for (int m = 0; m < maxLen - startIndex; m++)

{

result[m] = result[startIndex + m];

}

BigNumber resultNumber(result);

delete[] result;

return resultNumber;

}

BigNumber BigNumber::operator\*(const BigNumber& other) const

{

int len1 = numberLength;

int len2 = other.numberLength;

int maxLen = len1 + len2;

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

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];

result[i + j + 1] = sum % 10;

result[i + j] += sum / 10;

}

}

int idx = 0;

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

{

idx++;

}

if (idx == maxLen)

{

return BigNumber("0");

}

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

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

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

}

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

BigNumber product(resStr);

delete[] result;

delete[] resStr;

return product;

}

bool BigNumber::operator==(const BigNumber& other) const

{

return compare(other) == 1;

}

bool BigNumber::operator>(const BigNumber& other) const

{

return compare(other) == 2;

}

bool BigNumber::operator<(const BigNumber& other) const

{

return compare(other) == 3;

}

bool BigNumber::operator>=(const BigNumber& other) const

{

int result = compare(other);

return result == 1 || result == 2;

}

bool BigNumber::operator<=(const BigNumber& other) const

{

int result = compare(other);

return result == 1 || result == 3;

}

bool BigNumber::operator!=(const BigNumber& other) const

{

return compare(other) != 1;

}