Ministry of Science and Higher Education of the Russian Federation

Peter the Great St.Petersburg Polytechnic University

1. —
2. Institute for Cybersecurity and Information Protection

**LAB №1**

1. **STRING CLASS**
2. course «OOP»
3. Student
4. Gr. 4851003/80801 Lyakhova S.

<*signature*>

1. Instructor Chernov A.

< *signature* >

Saint-Petersburg

2020

1. **Task**
2. Implement a class to represent a character string in C++. Should not use the STL library containers and algorithms. The basic class must follow next requirements:

* **Constructor()** - default constructor, copy constructor, char array constructor, initializer list constructor, std::string constructor, init class with count characters of “char string”, init class with count of characters;
* **Destructor()**;
* **Operator +()** - concatenate with Mystring, char array, std::string;
* **Operator +=()** - assignment concatenate with char array, std::string;
* **Operator=()** - char string assignment, std::string assignment, char assignment;
* **Opearator[]** – index operator;
* **Operator>(), Operator<(), Operator>=(), Operator<=(), Operator!=(), Operator==()** - lexicographically comparing;
* **c\_str()** - return a pointer to null-terminated character array;
* **data()** - return a pointer to array data that not required to be null-terminated;
* **length()** - same as size;
* **size()** - return the number of char elements in string;
* **empty()** - true if string is empty;
* **capacity()** - return the current amount of allocated memory for array;
* **shrink\_to\_fit()** - reduce the capacity to size;
* **clear()** - remove all char element in string;
* **Operator<<(std::basic\_ostream), Operator>>(std::basic\_istream);**
* **insert(); erase(); append(); replace(); substr(); find().**

1. Extend python interpreter functionality with implemented class
2. **Additional task**: Implement unit tests for created class. Each implemented method should be covered by 2-3 test cases.
3. **Theory**
   1. **Strings in C and their problems**

A string in C is an array of characters, which in a good way should always end with a '\ 0' - the end-of-line character. Strings on the stack (static) are declared like this:

**char str [n] = {0},** n is the size of the character array, the same as the length of the string.

In addition, declare a pointer and transfer memory on the heap (heap) to it:

**char \* str = malloc (size),** size - the number of bytes that we allocate for the string.

A char array is just an array of characters, if allocated on the stack, it will always occupy, for example, 256 bytes regardless of the length of the text it contains.

If allocated on the heap (using malloc () or new char []), the programmer is responsible for later freeing the memory, and there will always be a heap allocation overhead. If copy text longer than 256 characters into an array, it may crash or cause unexplained (incorrect) behavior somewhere in the program. To determine the length of the text, it is necessary to scan the array character by character for the presence of the \ 0 character.

A string is a class that contains an array of characters, but automatically manages it for the programmer. Most string implementations have a built-in 16 character array (so short strings don't fragment the heap) and use the heap for longer strings.

Based on the definition of a string, there is a problem - a memory leak - which is exactly what happens most often when working with strings in C.

Therefore, for more convenient work with strings, so that the programmer does not have to monitor memory, so that the simplest methods were used for this type of variable, a string class was implemented.

1. **Results**

The presence of all methods inside the body of the class can make it difficult to manage and work with it, therefore methods are defined outside the body of the class itself. Methods of the class are defined as if they were regular functions, but prefix the function name with the class name with the scope resolution operator (::).

So, there are two files: an associated .h file **- string.h**, that consists of class definitions for easy reuse across multiple files or projects, string.cpp - contains class method implementations.

The created class **MyString** contains three private variables: **char\* \_myString** - dynamic character array, **int \_size** - the number of characters in a dynamic array, **int \_capacity** - the size of the storage space currently allocated for the string (this variable is recalculated when the length of the string changes by more than the current capacity).

To begin with, the class constructors and destructor were defined. The constructor as arguments can take an element of the class type, an array of characters, a list of initializers, a string defined by the library, a character string, and the number of characters to be passed.

Also, to work with arithmetic and logical operations, the operators +, + =, =,>, <,> =, <=,! =, ==, this will help us work with class objects. Using the implemented methods, we can get the characteristics of a string - length, capacity, return a pointer to a null-terminated character array, reduce the capacity to size

Functions implemented in string libraries, such as insert, erase, append, replace, substring, find, these functions are easy to implement and are basic.

1. **Wrapping C++ for Python**

For this part of the task was used as a Pybind11, this is a lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code.

The project consists of four files: **string.h** and **string.cpp** - these are the files that will directly contain the source code of string class; **main.cpp** is the file that will include the string.h and also contain the pybind11 code (which will contain the meta code for the bindings); **setup.py** - a file that will create a redistributable library and also simplify the library building process.

To build a solution, use the command: **python setup.py build\_ext -i**

File structure of **main.cpp**:

#include

PYBIND11\_MODULE(library, m) { };

* #include <pybind11/pybind11.h> - connected pybind.
* PYBIND11\_MODULE(library, m) – a macro that allows us to define a python module. Important: the name of the module (the first argument) must match the name specified in setup.py, otherwise an error will occur when importing the module.
* The second argument m is a kind of "factory" with which you can extend the functionality of our library, add functions, classes, etc.

Also methods that aren’t implemented in Python (like operator=, operator+ …) were imported.

1. **Unit test**

Automated tests show that the program works as intended. Such tests can be run multiple times. Successful execution of tests will show the developer that his changes did not break anything that was not planned to break.

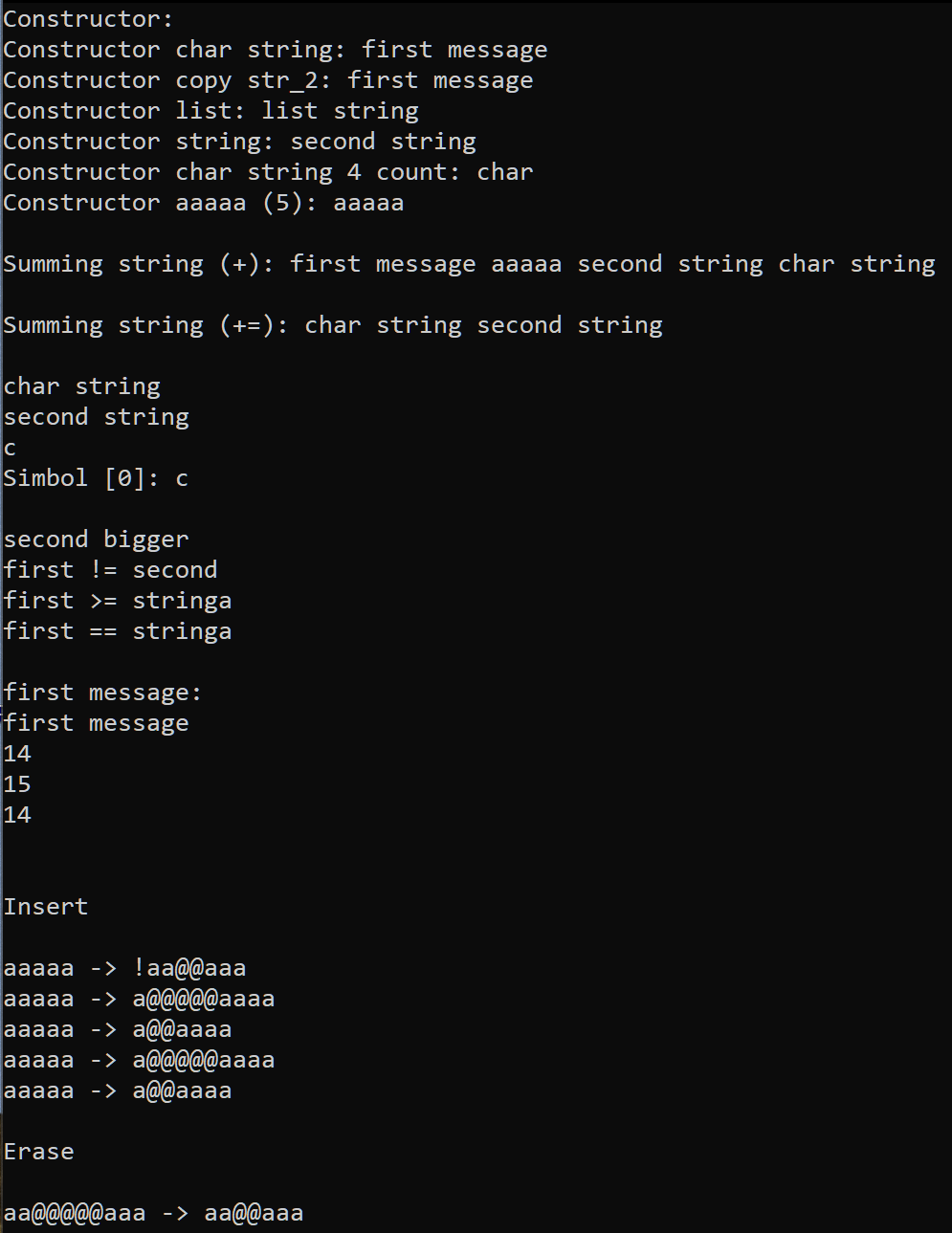
A failed test will reveal that changes have been made to the code that change or break its behavior. Examining the error generated by the failed test and comparing the expected result with the received one will give you an idea of where the error occurred.

In total, I have written 13 tests that completely cover all the functionality of the developed class.

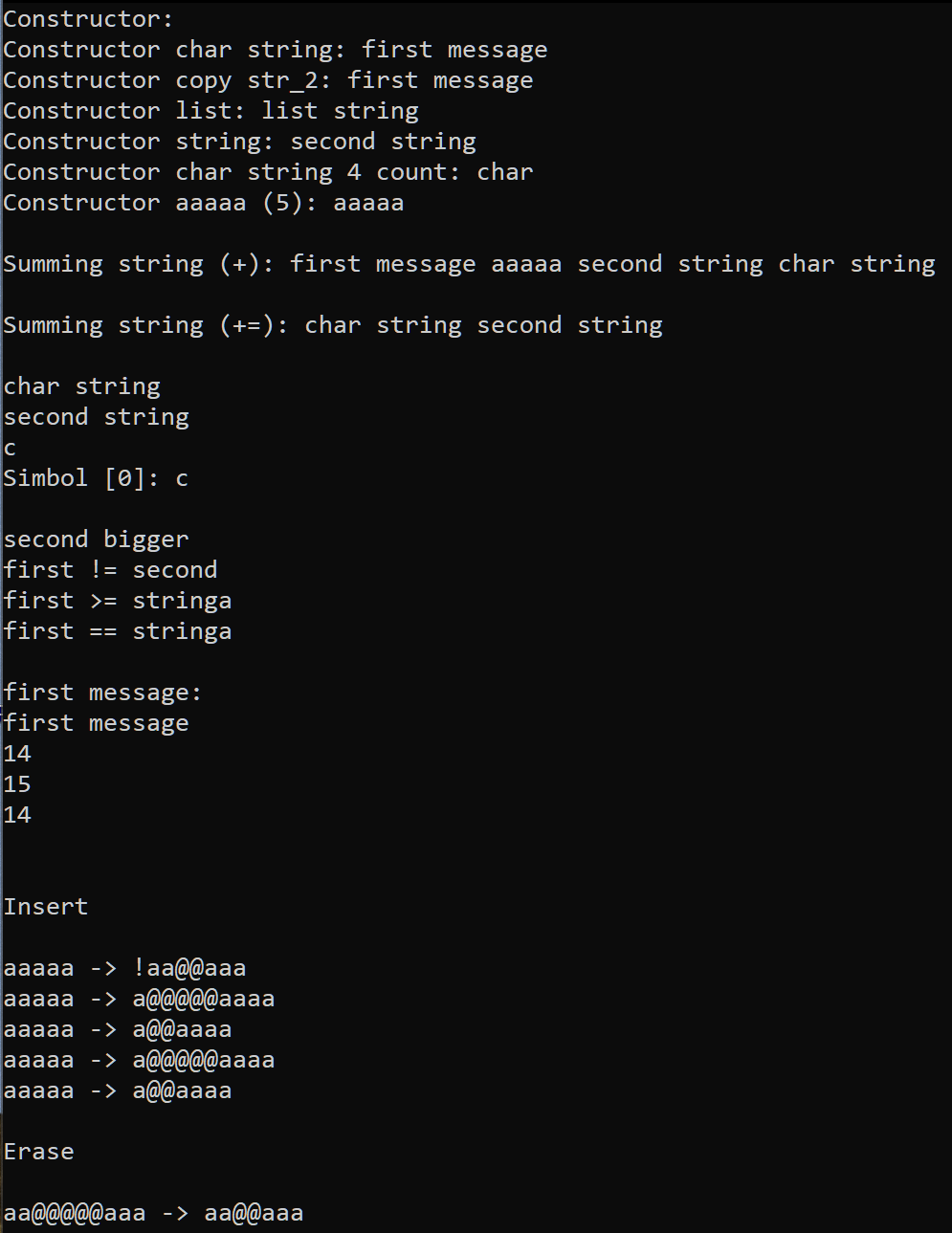
Each test uses the method **Assert::IsTrue(),** if the expression in parentheses is true, then the test is correct, otherwise you can see an error message, which is also an argument to the method. Also, was used **Assert::AreEqual()**, a method that compares two of its arguments.

1. **Conclusion**

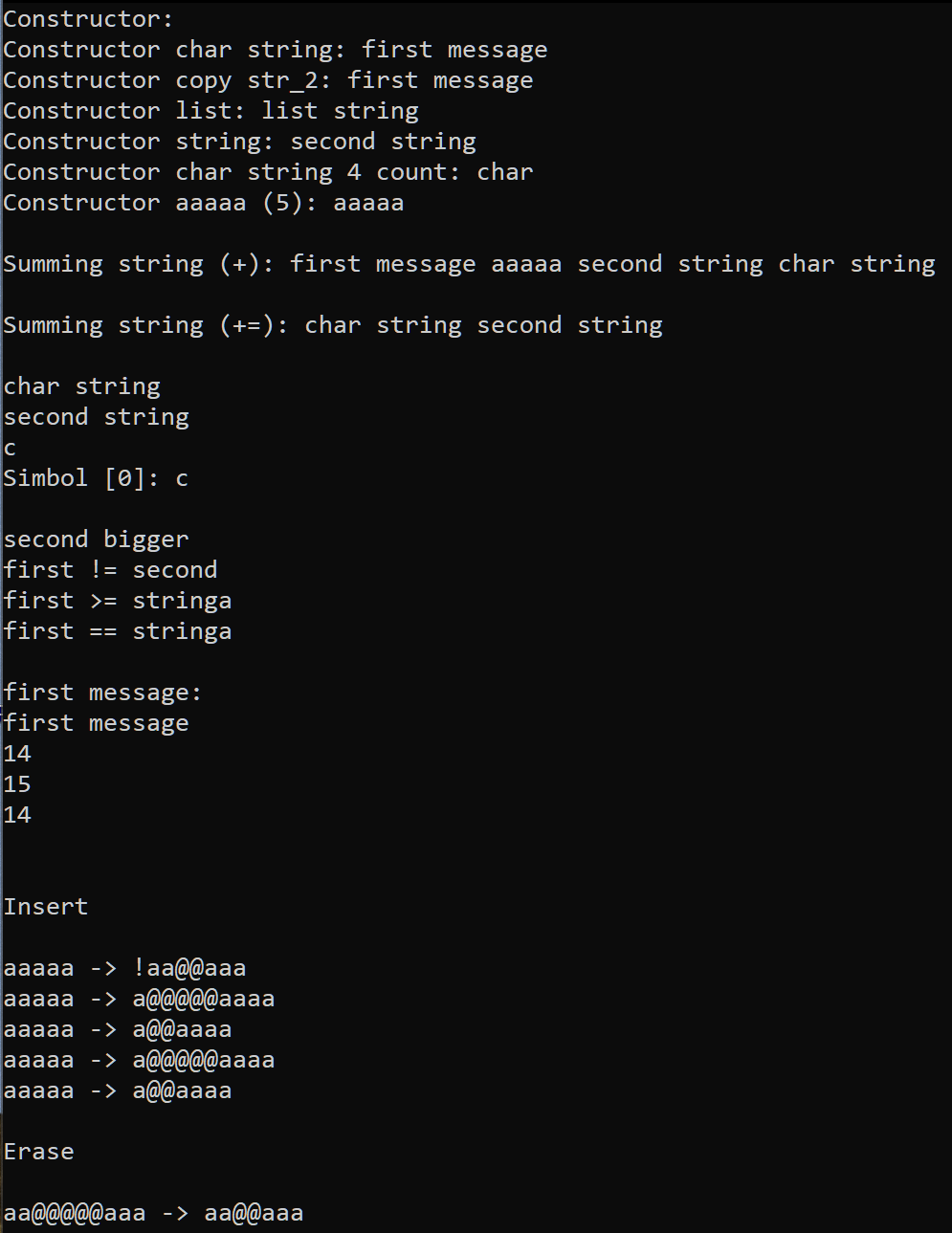
In conclusion, in the course of this lab, I implemented a string class that performs the basic functions for working with strings. Also, the functions were checked for correct operation, the file was implemented test.cpp with an example of work all method of the class.



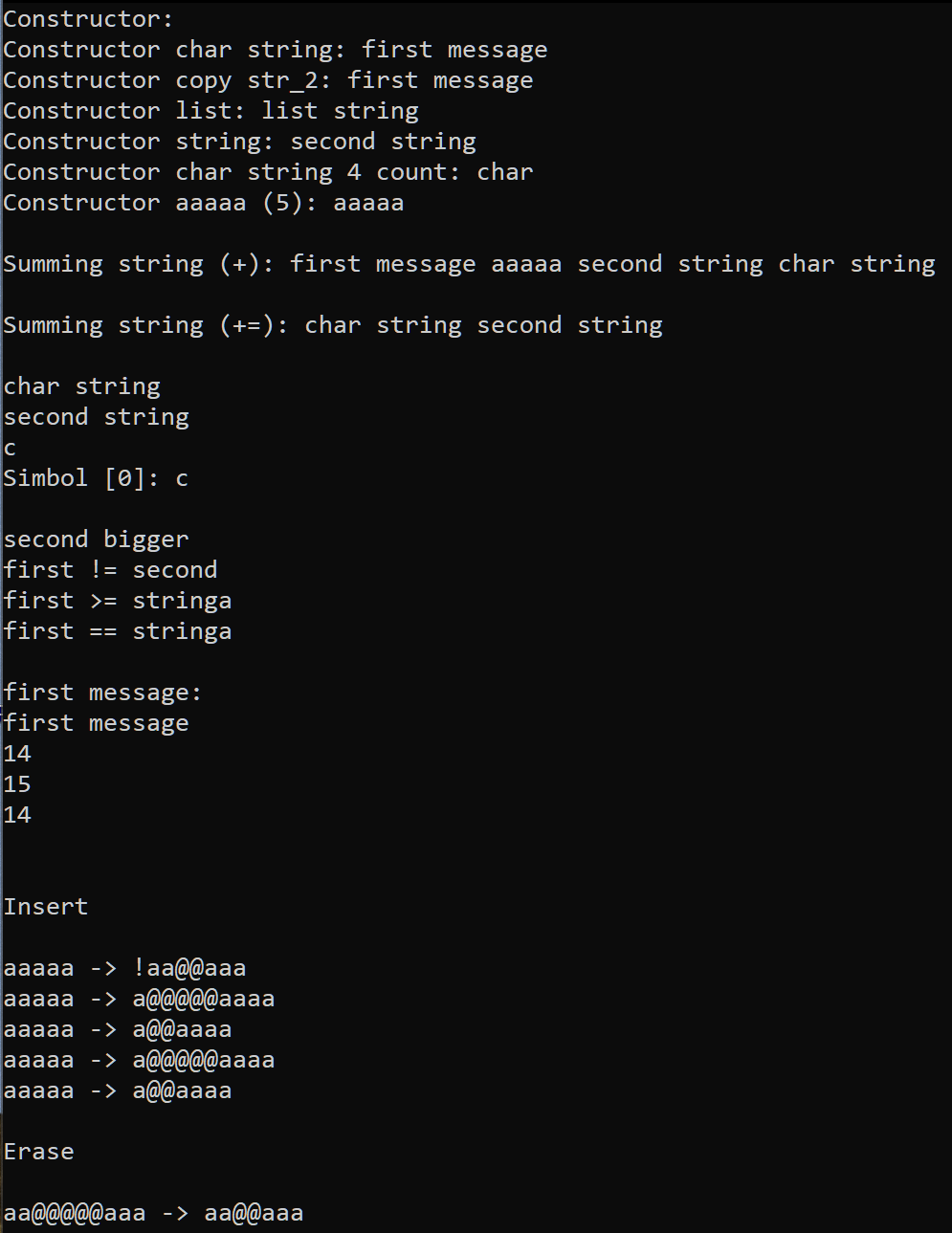
Img. 1. An example of work Constructor()



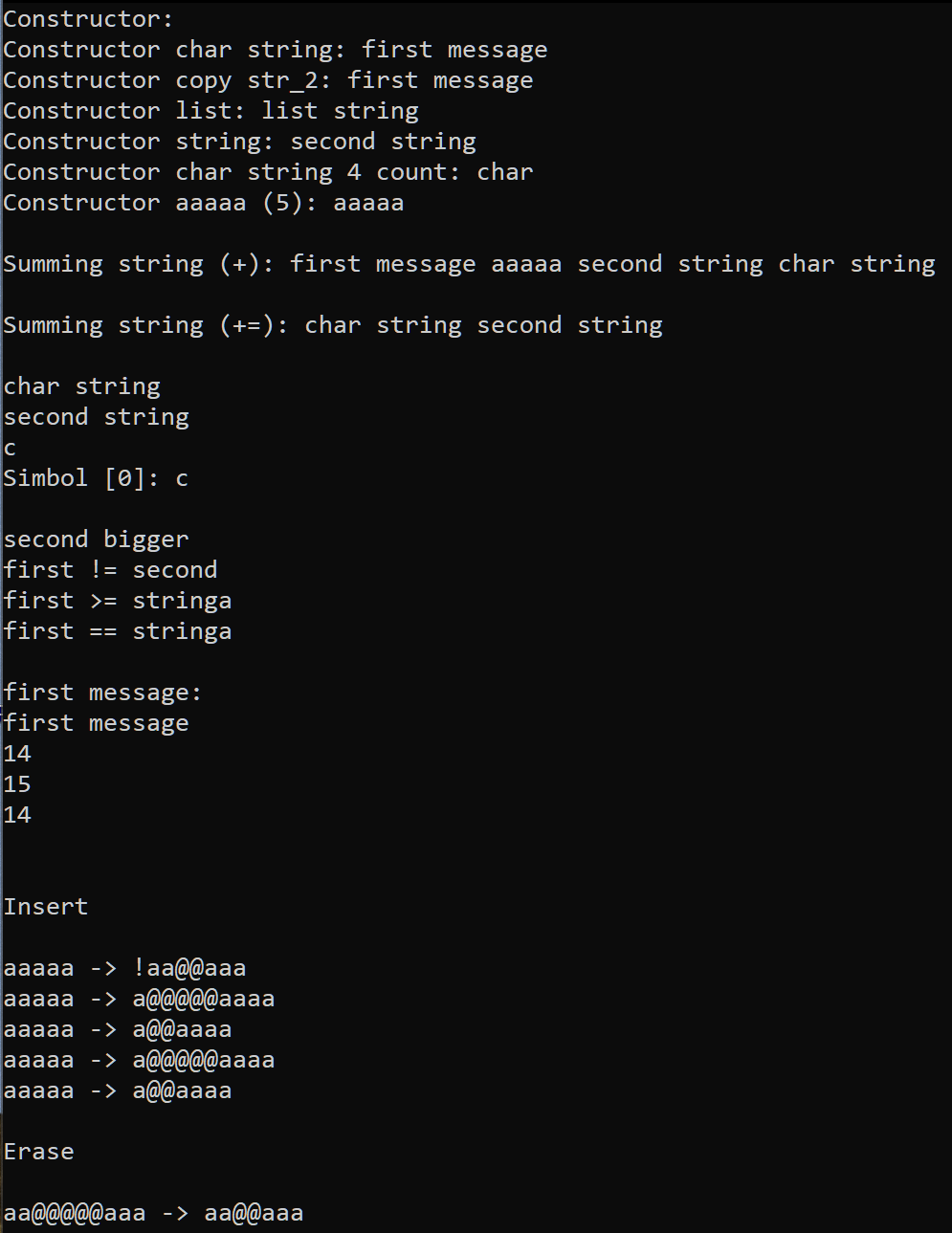
Img. 2. An example of string summation



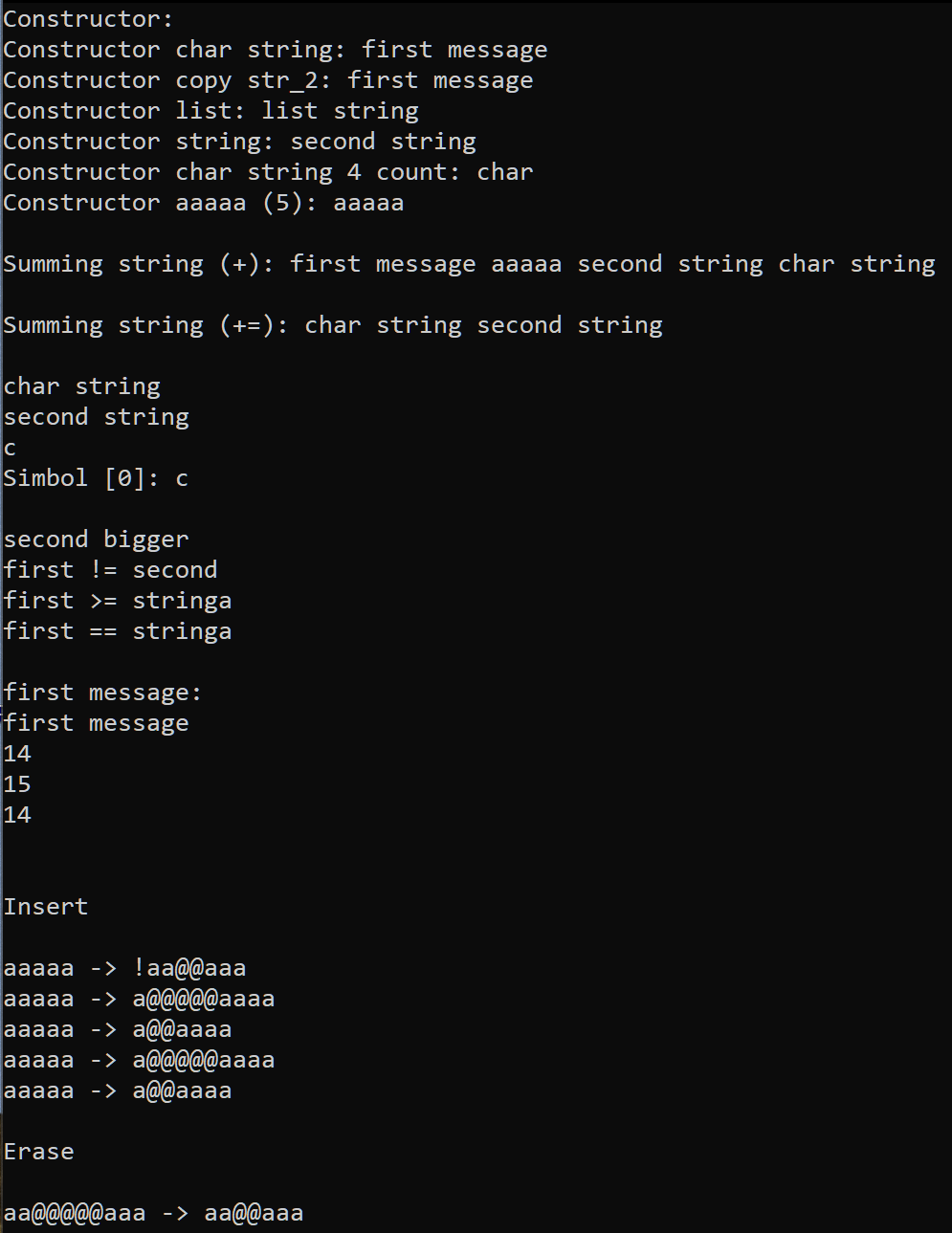
Img. 3. An example of Operator=()



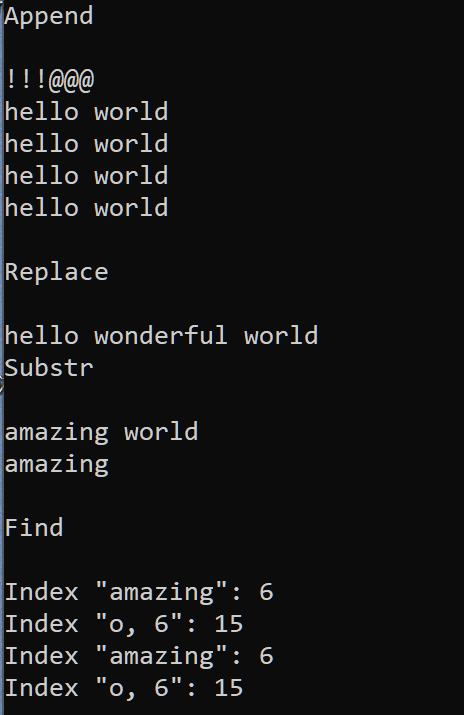
Img. 4. An example of Operator<(), Operator>(), Operator!=(), Operator>=(), Operator==()



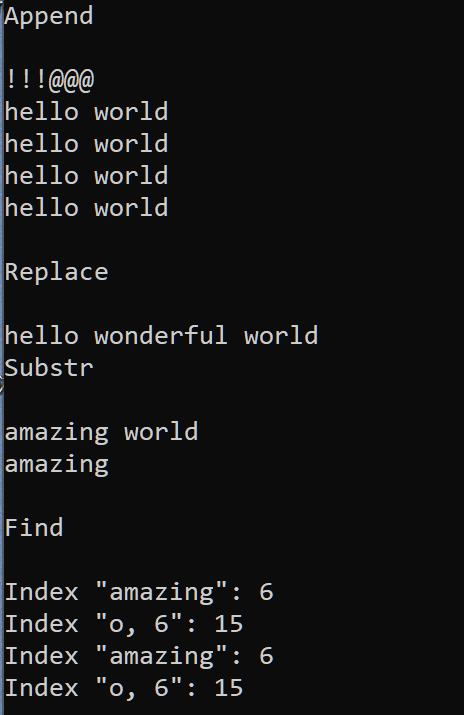
Img. 5. An example of insert



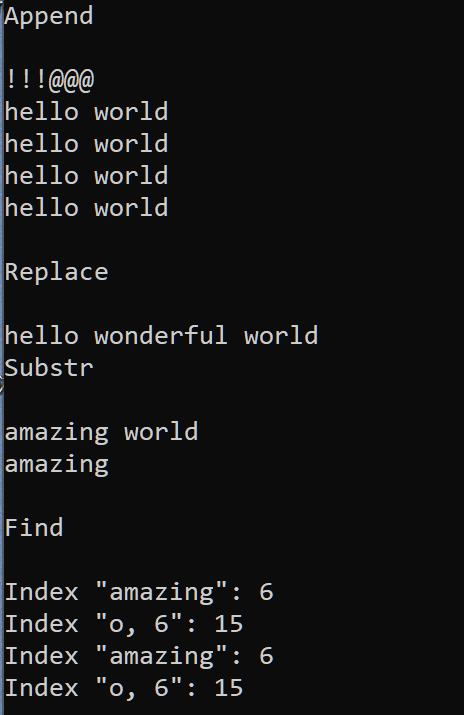
Img. 6. An example of erase



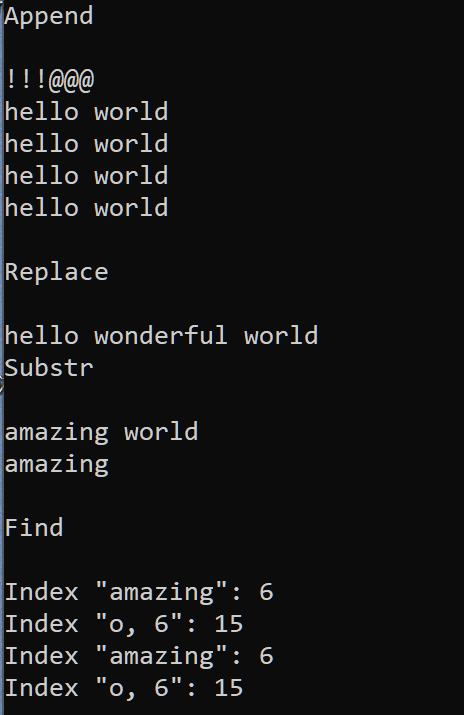
Img. 7. An example of append



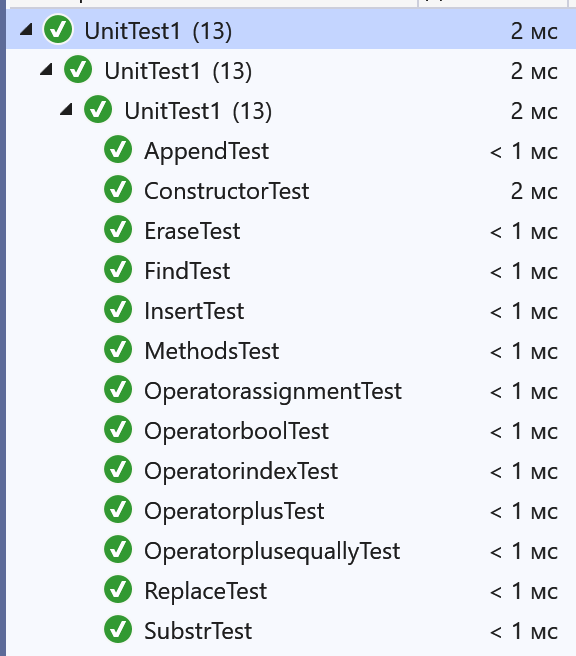
Img. 8. An example of replace



Img. 9. An example of substr



Img. 10. An example of find



Img. 11. Correct work unite test

Appendix A

string.h

#ifndef \_\_STRING\_H\_\_

#define \_\_STRING\_H\_\_

#include<stdio.h>

#include<conio.h>

#include<iostream>

#include<new>

using namespace std;

class MyString

{

public:

MyString(); // default constructor

MyString(const MyString& str); //copy constructor

MyString(const char\* str); // char array constructor

MyString(const initializer\_list<char> &list); // initializer list constructor

MyString(string str); // std::string constructor

MyString(const char\* s, size\_t count);// init class with count characters of “char string”,Constructor(“hello”, 4) -> “hell”

MyString(size\_t count, char c); // init class with count of characters Constructor(4, ‘a’)->“aaa”

MyString(const MyString& str, size\_t index, size\_t count); // use in function find()

~MyString(); // destructor

//operator

MyString operator+ (const MyString& str); // concatenate with Mystring

MyString operator+ (const char\* str); // concatenate with char array

MyString operator+ (string str); // concatenate with std::string

MyString operator+ (char c); // concatenate with char

MyString operator+= (const MyString& str);// assignment concatenate constructor

MyString operator+= (const char\* s); // assignment concatenate with char array

MyString operator+= (char c);// char assignment

MyString operator+= (string str); // concatenate with std::string

MyString& operator= (const MyString& str); // constructor string assignment

MyString& operator= (const char\* str); // char string assignment

MyString& operator= (string str); // std::string assignment

MyString& operator= (const char c); // char assignment

char operator[] (int i); // index operator, Mystring(“012345”)[5] ->‘5’

// lexicographically comparing

bool operator>(const MyString& str);

bool operator> (const char\* str);

bool operator> (string str);

bool operator<(const MyString& str);

bool operator< (const char\* str);

bool operator< (string str);

bool operator>=(const MyString& str);

bool operator>= (const char\* str);

bool operator>= (string str);

bool operator<=(const MyString& str);

bool operator<= (const char\* str);

bool operator<= (string str);

bool operator!=(const MyString& str);

bool operator!= (const char\* str);

bool operator!= (string str);

bool operator==(const MyString& str);

bool operator== (const char\* str);

bool operator== (string str);

////////////////////////////////////////////////////////

char\* c\_str(); // return a pointer to null-terminated character array

char\* data(); // return a pointer to array data that not required to be null-terminated

int length(); // same as size

int size(); // return the number of char elements in string

bool empty(); // true if st getlinering is empty

int capacity(); // return the current amount of allocated memory for array

void shrink\_to\_fit(); //reduce the capacity to size

void clear(); // remove all char element in string

friend ostream& operator<<(ostream&, MyString str);

friend istream& operator>>(istream&, MyString str);

////////////////////////////////////////////////////////

MyString& insert(size\_t index, size\_t count, char c); // insert count of char in index position

MyString& insert(size\_t index, const char\* str); // insert null-terminated char string at index position

MyString& insert(size\_t index, const char\* str, size\_t count); // insert count of null-terminated char string at index position

MyString& insert(size\_t index, string str); // insert std::string at index position

MyString& insert(size\_t index, string str, size\_t count);// insert count of std::string at index position

MyString& insert(size\_t index, const MyString& str);

////////////////////////////////////////////////////////

MyString& erase(size\_t index, size\_t count);

////////////////////////////////////////////////////////

MyString& append(size\_t n, char c); // append count of char

MyString& append(const char\* str); // append null-terminated char string

MyString& append(const char\* str, size\_t index, size\_t count);

MyString& append(string str); // append std:: string

MyString& append(string str, size\_t index, size\_t count); // append a count of std:: string by index position

MyString& append(const MyString& str);

MyString& append(const MyString& str, size\_t index, size\_t count);

////////////////////////////////////////////////////////

MyString& replace(size\_t index, size\_t count, const MyString& str);

MyString& replace(size\_t index, size\_t count, const char\* s); // replace a count of char at index by “string”

MyString& replace(size\_t index, size\_t count, string str);

////////////////////////////////////////////////////////

MyString substr(size\_t index) const; //return a substring starts with index position

MyString substr(size\_t index, size\_t count) const; // return a count of substring’s char starts with index position

////////////////////////////////////////////////////////

int find(const MyString& str, size\_t index) const;

int find(const char\* str) const; // if founded return the index of substring

int find(const char\* str, size\_t index) const; // if founded return the index of substring

int find(string str) const;

int find(string str, size\_t index) const; // if founded return the index of substring

private:

char\* \_myString; //string

int \_size; //size of string

int \_capacity; //max size of string

};

#endif // \_STRING\_H\_

Appendix B

string.cpp

#include"string.h"

MyString::MyString()

{

\_size = 0;

\_capacity = 15;

this->\_myString = new char[\_capacity];

}

MyString::MyString(const MyString& str)

{

this->\_size = str.\_size;

this->\_capacity = ((this->\_size / 16) + 1) \* 16 - 1;

this->\_myString = new char[this->\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = str.\_myString[i];

}

}

MyString::MyString(const char\* str)

{

int size\_str = 0;

while (str[size\_str] != '\0')

{

size\_str++;

}

this->\_size = size\_str;

this->\_capacity = ((this->\_size / 16) + 1) \* 16 - 1;

this->\_myString = new char[this->\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = str[i];

}

}

MyString::MyString(const initializer\_list<char>& list)

{

this->\_size = list.size();

this->\_capacity = ((this->\_size / 16) + 1) \* 16 - 1;

this->\_myString = new char[this->\_capacity];

int count = 0;

for (auto& item : list)

{

this->\_myString[count] = item;

count++;

}

}

MyString::MyString(string str)

{

this->\_size = str.size();

this->\_capacity = ((this->\_size / 16) + 1) \* 16 - 1;

this->\_myString = new char[this->\_capacity];

for (int i =0; i < this->\_size; i++)

{

this->\_myString[i] = str[i];

}

}

MyString::MyString(const char\* str, size\_t count)

{

this->\_size = count;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[this->\_size];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = str[i];

}

}

MyString::MyString(size\_t count, char c)

{

this->\_size = count;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[this->\_size];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = c;

}

}

MyString::~MyString()

{

delete \_myString;

}

////////////////////////////////////////////////////////////////////////////////////

MyString MyString::operator+ (const MyString& str)

{

if (this->\_size + str.\_size <= this->\_capacity)

{

for (int i = 0; i < str.\_size; i++)

{

this->\_myString[i + this->\_size] = str.\_myString[i];

}

this->\_size += str.\_size;

}

else

{

MyString p(\*this);

delete this->\_myString;

this->\_size += str.\_size;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[this->\_size];

for (int i = 0; i < p.\_size; i++)

{

this->\_myString[i] = p.\_myString[i];

}

for (int i = 0; i < str.\_size; i++)

{

this->\_myString[i + p.\_size] = str.\_myString[i];

}

}

return \*this;

}

MyString MyString::operator+ (const char\* str)

{

MyString new\_str(str);

return \*this + new\_str;

}

MyString MyString::operator+ (string str)

{

MyString new\_str(str);

return \*this + new\_str;

}

MyString MyString::operator+ (char c)

{

MyString new\_str(1, c);

return \*this + new\_str;

}

// +=

MyString MyString::operator+= (const MyString& str)

{

return \*this + str;

}

MyString MyString::operator+= (const char\* s)

{

MyString new\_str(s);

return \*this + new\_str;

}

MyString MyString::operator+= (char c)

{

MyString new\_str(1, c);

return \*this + new\_str;

}

MyString MyString::operator+= (string str)

{

MyString new\_str(str);

return \*this + new\_str;

}

// =

MyString& MyString::operator= (const MyString& str)

{

this->\_size = str.\_size;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[this->\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = str.\_myString[i];

}

return \*this;

}

MyString& MyString::operator= (const char\* str)

{

int sizeOfStr = 0;

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

{

sizeOfStr++;

}

this->\_size = sizeOfStr;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[this->\_capacity];

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

{

this->\_myString[i] = str[i];

}

return \*this;

}

MyString& MyString::operator= (string str)

{

this->\_size = str.size();

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[this->\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = str[i];

}

return \*this;

}

MyString& MyString::operator= (const char c)

{

this->\_size = 1;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[this->\_capacity];

this->\_myString[0] = c;

return \*this;

}

char MyString::operator[](int i)

{

return this->\_myString[i];

}

//////////////////////////////////////////////////

bool MyString::operator> (const MyString& str)

{

if (this->\_size < str.\_size)

{

return 0;

}

else if (this->\_size == str.\_size)

{

for (int i = 0; i < this->\_size; i++)

{

if (this->\_myString[i] < str.\_myString[i])

{

return 0;

}

else if (this->\_myString[i] > str.\_myString[i])

{

return 1;

}

}

return 0;

}

}

bool MyString::operator> (const char\* str)

{

MyString new\_str(str);

return \*this > new\_str;

}

bool MyString::operator> (string str)

{

MyString new\_str(str);

return \*this > new\_str;

}

bool MyString::operator<(const MyString& str)

{

if (this->\_size > str.\_size)

{

return 0;

}

else if (this->\_size == str.\_size)

{

for (int i = 0; i < this->\_size; i++)

{

if (this->\_myString[i] > str.\_myString[i])

{

return 0;

}

else if (this->\_myString[i] < str.\_myString[i])

{

return 1;

}

}

return 0;

}

}

bool MyString::operator< (const char\* str)

{

MyString new\_str(str);

return \*this < new\_str;

}

bool MyString::operator< (string str)

{

MyString new\_str(str);

return \*this < new\_str;

}

bool MyString::operator>=(const MyString& str)

{

if (this->\_size < str.\_size)

{

return 0;

}

else if (this->\_size == str.\_size)

{

for (int i = 0; i < this->\_size; i++)

{

if (this->\_myString[i] < str.\_myString[i])

{

return 0;

}

}

}

return 1;

}

bool MyString::operator>= (const char\* str)

{

MyString new\_str(str);

return \*this >= new\_str;

}

bool MyString::operator>= (string str)

{

MyString new\_str(str);

return \*this >= new\_str;

}

bool MyString::operator<=(const MyString& str)

{

if (this->\_size > str.\_size)

{

return 0;

}

else if (this->\_size == str.\_size)

{

for (int i = 0; i < this->\_size; i++)

{

if (this->\_myString[i] > str.\_myString[i])

{

return 0;

}

}

}

return 1;

}

bool MyString::operator<= (const char\* str)

{

MyString new\_str(str);

return \*this <= new\_str;

}

bool MyString::operator<= (string str)

{

MyString new\_str(str);

return \*this <= new\_str;

}

bool MyString::operator!= (const MyString& str)

{

if (this->\_size == str.\_size)

{

int count = 0;

for (int i = 0; i < this->\_size; i++)

{

if (this->\_myString[i] == str.\_myString[i])

{

count++;

}

}

if (count == str.\_size)

{

return 0;

}

}

return 1;

}

bool MyString::operator!= (const char\* str)

{

MyString new\_str(str);

return \*this != new\_str;

}

bool MyString::operator!= (string str)

{

MyString new\_str(str);

return \*this != new\_str;

}

bool MyString::operator==(const MyString& str)

{

if (this->\_size == str.\_size)

{

for (int i = 0; i < this->\_size; i++)

{

if (this->\_myString[i] != str.\_myString[i])

{

return 0;

}

}

}

else

{

return 0;

}

return 1;

}

bool MyString::operator== (const char\* str)

{

MyString new\_str(str);

return \*this == new\_str;

}

bool MyString::operator== (string str)

{

MyString new\_str(str);

return \*this == new\_str;

}

////////////////////////////////////////////////////////////////////////////

char\* MyString::c\_str()

{

this->\_myString[this->\_size] = '\0';

return this->\_myString;

}

char\* MyString::data()

{

this->\_myString[this->\_size] = '\0';

return this->\_myString;

}

int MyString::length()

{

return this->\_size;

}

int MyString::size()

{

return this->\_size;

}

bool MyString::empty()

{

if (this->\_size != 0)

return 0;

return 1;

}

int MyString::capacity()

{

return \_capacity;

}

void MyString::shrink\_to\_fit()

{

char\* new\_str;

new\_str = new char[this->\_size];

for (int i = 0; i < this->\_size; i++)

{

new\_str[i] = this->\_myString[i];

}

delete this->\_myString;

this->\_myString = new char[this->\_size];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

this->\_capacity = this->\_size;

delete new\_str;

}

void MyString::clear()

{

this->\_size = 0;

}

////////////////////////////////////////////////////////////////////////////

ostream& operator<<(ostream& out, MyString str)

{

for (int i = 0; i < str.\_size; i++)

{

out << str.\_myString[i];

}

return out;

}

istream& operator>>(istream& in, MyString str)

{

for (int i = 0; i < str.\_size; i++)

{

in >> str.\_myString[i];

}

return in;

}

////////////////////////////////////////////////////////////////////////////

MyString& MyString::insert(size\_t index, size\_t count, char c)

{

if (this->\_size + count <= this->\_capacity)

{

for (int i = this->\_size + count - 1; i >= index + count; i--)

{

this->\_myString[i] = this->\_myString[i - count];

}

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

{

this->\_myString[i + index] = c;

}

this->\_size += count;

}

else

{

this->\_capacity = ((this->\_size + count) / 16 + 1) \* 16 - 1;

char\* new\_str;

new\_str = new char[this->\_capacity];

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

{

new\_str[i] = this->\_myString[i];

}

for (int i = index; i < index + count; i++)

{

new\_str[i] = c;

}

for (int i = index + count; i < this->\_size + count; i++)

{

new\_str[i] = this->\_myString[i - count];

}

this->\_size += count;

delete this->\_myString;

this->\_myString = new char[this->\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

MyString& MyString::insert(size\_t index, const char\* str)

{

int sizeOfStr = 0;

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

{

sizeOfStr++;

}

if (this->\_size + sizeOfStr <= this->\_capacity)

{

for (int i = this->\_size + sizeOfStr - 1; i >= index + sizeOfStr; i--)

{

this->\_myString[i] = this->\_myString[i - sizeOfStr];

}

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

{

this->\_myString[i + index] = str[i];

}

this->\_size += sizeOfStr;

}

else

{

this->\_capacity = ((this->\_size + sizeOfStr) / 16 + 1) \* 16 - 1;

char\* new\_str;

new\_str = new char[this->\_capacity];

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

{

new\_str[i] = this->\_myString[i];

}

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

{

new\_str[i + index] = str[i];

}

for (int i = index + sizeOfStr; i < this->\_size + sizeOfStr; i++)

{

new\_str[i] = this->\_myString[i - sizeOfStr];

}

this->\_size += sizeOfStr;

delete this->\_myString;

this->\_myString = new char[this->\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

MyString& MyString::insert(size\_t index, const char\* str, size\_t count)

{

if (this->\_size + count <= this->\_capacity)

{

for (int i = this->\_size + count - 1; i >= index + count; i--)

{

this->\_myString[i] = this->\_myString[i - count];

}

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

{

this->\_myString[i + index] = str[i];

}

this->\_size += count;

}

else

{

this->\_capacity = ((this->\_size + count) / 16 + 1) \* 16 - 1;

char\* new\_str;

new\_str = new char[this->\_capacity];

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

{

new\_str[i] = this->\_myString[i];

}

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

{

new\_str[i + index] = str[i];

}

for (int i = index + count; i < this->\_size + count; i++)

{

new\_str[i] = this->\_myString[i - count];

}

this->\_size += count;

delete this->\_myString;

this->\_myString = new char[this->\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

MyString& MyString::insert(size\_t index, string str)

{

int sizeOfStr = str.size();

if (this->\_size + sizeOfStr <= this->\_capacity)

{

for (int i = this->\_size + sizeOfStr - 1; i >= index + sizeOfStr; i--)

{

this->\_myString[i] = this->\_myString[i - sizeOfStr];

}

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

{

this->\_myString[i + index] = str[i];

}

this->\_size += sizeOfStr;

}

else

{

this->\_capacity = ((this->\_size + sizeOfStr) / 16 + 1) \* 16 - 1;

char\* new\_str;

new\_str = new char[this->\_capacity];

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

{

new\_str[i] = this->\_myString[i];

}

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

{

new\_str[i + index] = str[i];

}

for (int i = index + sizeOfStr; i < this->\_size + sizeOfStr; i++)

{

new\_str[i] = this->\_myString[i - sizeOfStr];

}

this->\_size += sizeOfStr;

delete this->\_myString;

this->\_myString = new char[this->\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

MyString& MyString::insert(size\_t index, string str, size\_t count)

{

if (this->\_size + count <= this->\_capacity)

{

for (int i = this->\_size + count - 1; i >= index + count; i--)

{

this->\_myString[i] = this->\_myString[i - count];

}

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

{

this->\_myString[i + index] = str[i];

}

this->\_size += count;

}

else

{

this->\_capacity = ((this->\_size + count) / 16 + 1) \* 16 - 1;

char\* new\_str;

new\_str = new char[this->\_capacity];

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

{

new\_str[i] = this->\_myString[i];

}

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

{

new\_str[i + index] = str[i];

}

for (int i = index + count; i < this->\_size + count; i++)

{

new\_str[i] = this->\_myString[i - count];

}

this->\_size += count;

delete this->\_myString;

this->\_myString = new char[this->\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

MyString& MyString::insert(size\_t index, const MyString& str)

{

if (this->\_size + str.\_size <= this->\_capacity)

{

for (int i = this->\_size + str.\_size - 1; i >= index + str.\_size; i--)

{

this->\_myString[i] = this->\_myString[i - str.\_size];

}

for (int i = 0; i < str.\_size; i++)

{

this->\_myString[i + index] = str.\_myString[i];

}

this->\_size += str.\_size;

}

else

{

this->\_capacity = ((this->\_size + str.\_size) / 16 + 1) \* 16 - 1;

char\* new\_str;

new\_str = new char[this->\_capacity];

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

{

new\_str[i] = this->\_myString[i];

}

for (int i = index; i < index + str.\_size; i++)

{

new\_str[i] = str.\_myString[i - index];

}

for (int i = index + str.\_size; i < this->\_size + str.\_size; i++)

{

new\_str[i] = this->\_myString[i - str.\_size];

}

this->\_size += str.\_size;

delete this->\_myString;

this->\_myString = new char[this->\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

////////////////////////////////////////////////////////////////////////////

MyString& MyString::erase(size\_t index, size\_t count)

{

for (int i = 0; i < this->\_size - index - count; i++)

{

this->\_myString[i + index] = this->\_myString[i + index + count];

}

this->\_size = this->\_size - count;

return \*this;

}

////////////////////////////////////////////////////////////////////////////

MyString& MyString::append(size\_t count, char c)

{

if (this->\_size + count <= this->\_capacity)

{

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

{

this->\_myString[i + this->\_size] = c;

}

this->\_size += count;

}

else

{

char\* new\_str;

new\_str = new char[this->\_size + count];

for (int i = 0; i < this->\_size; i++)

{

new\_str[i] = this->\_myString[i];

}

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

{

new\_str[i + this->\_size] = c;

}

delete this->\_myString;

this->\_size += count;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

MyString& MyString::append(const char\* str)

{

int sizeOfStr = 0;

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

{

sizeOfStr++;

}

if (this->\_size + sizeOfStr <= this->\_capacity)

{

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

{

this->\_myString[i + this->\_size] = str[i];

}

this->\_size += sizeOfStr;

}

else

{

char\* new\_str;

new\_str = new char[this->\_size + sizeOfStr];

for (int i = 0; i < this->\_size; i++)

{

new\_str[i] = this->\_myString[i];

}

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

{

new\_str[i + this->\_size] = str[i];

}

delete this->\_myString;

this->\_size += sizeOfStr;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

MyString& MyString::append(const char\* str, size\_t index, size\_t count)

{

if (this->\_size + count <= this->\_capacity)

{

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

{

this->\_myString[i + this->\_size] = str[i + index];

}

this->\_size += count;

}

else

{

char\* new\_str;

new\_str = new char[this->\_size + count];

for (int i = 0; i < this->\_size; i++)

{

new\_str[i] = this->\_myString[i];

}

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

{

new\_str[i + this->\_size] = str[i + index];

}

delete this->\_myString;

this->\_size += count;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

MyString& MyString::append(string str)

{

MyString new\_str(str);

return this->append(new\_str);

}

MyString& MyString::append(string str, size\_t index, size\_t count)

{

if (this->\_size + count <= this->\_capacity)

{

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

{

this->\_myString[i + this->\_size] = str[i + index];

}

this->\_size += count;

}

else

{

char\* new\_str;

new\_str = new char[this->\_size + count];

for (int i = 0; i < this->\_size; i++)

{

new\_str[i] = this->\_myString[i];

}

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

{

new\_str[i + this->\_size] = str[i + index];

}

delete this->\_myString;

this->\_size += count;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

MyString& MyString::append(const MyString& str)

{

if (this->\_size + str.\_size <= this->\_capacity)

{

for (int i = 0; i < str.\_size; i++)

{

this->\_myString[i + this->\_size] = str.\_myString[i];

}

this->\_size += str.\_size;

}

else

{

char\* new\_str;

new\_str = new char[this->\_size + str.\_size];

for (int i = 0; i < this->\_size; i++)

{

new\_str[i] = this->\_myString[i];

}

for (int i = 0; i < str.\_size; i++)

{

new\_str[i + this->\_size] = str.\_myString[i];

}

delete this->\_myString;

this->\_size += str.\_size;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

MyString& MyString::append(const MyString& str, size\_t index, size\_t count)

{

if (this->\_size + count <= this->\_capacity)

{

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

{

this->\_myString[i + this->\_size] = str.\_myString[index + i];

}

this->\_size += count;

}

else

{

char\* new\_str;

new\_str = new char[this->\_size + count];

for (int i = 0; i < this->\_size; i++)

{

new\_str[i] = this->\_myString[i];

}

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

{

new\_str[i + this->\_size] = str.\_myString[index + i];

}

delete this->\_myString;

this->\_size = this->\_size + count;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[\_capacity];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = new\_str[i];

}

delete new\_str;

}

return \*this;

}

////////////////////////////////////////////////////////////////////////////

MyString& MyString::replace(size\_t index, size\_t count, const MyString& str)

{

if (index + count <= this->\_size)

{

if (this->\_size + (str.\_size - count) < this->\_capacity)

{

char\* p; //after the change word

p = new char[this->\_size - index - count];

for (int i = 0; i < this->\_size - index - count; i++)

{

p[i] = this->\_myString[index + count + i];

}

for (int i = 0; i < str.\_size; i++)

{

this->\_myString[i + index] = str.\_myString[i];

}

for (int i = 0; i < this->\_size - index - count; i++)

{

this->\_myString[i + index + str.\_size] = p[i];

}

this->\_size = this->\_size + str.\_size - count;

delete p;

}

else

{

char\* p1, \* p2;

p1 = new char[index]; //befor word

p2 = new char[this->\_size - index - count]; //after word

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

{

p1[i] = this->\_myString[i];

}

for (int i = 0; i < this->\_size - index - count; i++)

{

p2[i] = this->\_myString[index + count + i];

}

delete this->\_myString;

this->\_capacity = ((this->\_size + str.\_size - count) / 16 + 1) \* 16 - 1;

this->\_myString = new char[this->\_capacity];

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

{

this->\_myString[i] = p1[i];

}

for (int i = 0; i < str.\_size; i++)

{

this->\_myString[i + index] = str.\_myString[i];

}

for (int i = 0; i < this->\_size - index - count; i++)

{

this->\_myString[i + index + str.\_size] = p2[i];

}

this->\_size = this->\_size + str.\_size - count;

delete p1, p2;

}

}

else

{

if (this->\_size + (str.\_size - count) < this->\_capacity)

{

this->\_size = index + str.\_size;

for (int i = 0; i < str.\_size; i++)

{

this->\_myString[i + index] = str.\_myString[i];

}

}

else

{

char\* p1;

p1 = new char[index];

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

{

p1[i] = this->\_myString[i];

}

delete this->\_myString;

this->\_size = index + str.\_size;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[this->\_capacity];

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

{

this->\_myString[i] = p1[i];

}

for (int i = 0; i < str.\_size; i++)

{

this->\_myString[i + index] = str.\_myString[i];

}

delete p1;

}

}

return \*this;

}

MyString& MyString::replace(size\_t index, size\_t count, const char\* str)

{

MyString new\_str(str);

return this->replace(index, count, new\_str);

}

MyString& MyString::replace(size\_t index, size\_t count, string str)

{

MyString new\_str(str);

return this->replace(index, count, new\_str);

}

////////////////////////////////////////////////////////////////////////////////

MyString MyString::substr(size\_t index) const

{

return this->substr(index, this->\_size - index);

}

MyString MyString::substr(size\_t index, size\_t count) const

{

MyString str;

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

{

str += this->\_myString[i + index];

}

return str;

}

////////////////////////////////////////////////////////////////////////////////

MyString::MyString(const MyString& str, size\_t index, size\_t count)

{

this->\_size = count;

this->\_capacity = (this->\_size / 16 + 1) \* 16 - 1;

this->\_myString = new char[this->\_size];

for (int i = 0; i < this->\_size; i++)

{

this->\_myString[i] = str.\_myString[i + index];

}

}

int MyString::find(const MyString& str, size\_t index) const

{

for (int i = index; i < this->\_size - str.\_size; i++)

{

MyString p(\*this, i, str.\_size);

if (p == str)

{

return i;

}

}

return -1;

}

int MyString::find(const char\* str) const

{

MyString new\_str(str);

return this->find(new\_str, 0);

}

int MyString::find(const char\* str, size\_t index) const

{

MyString new\_str(str);

return this->find(new\_str, index);

}

int MyString::find(string str) const

{

MyString new\_str(str);

return this->find(new\_str, 0);

}

int MyString::find(string str, size\_t index) const

{

MyString new\_str(str);

return this->find(new\_str, index);

}

Appendix C

UnitTest1.cpp

#include "CppUnitTest.h"

#include "..\class\_string\string.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1

{

TEST\_CLASS(UnitTest1)

{

public:

TEST\_METHOD(ConstructorTest)

{

initializer\_list<char> list = { 'l', 'i', 's', 't', ' ', 's', 't', 'r', 'i', 'n', 'g' };

string string\_1 = "second string";

const char\* char\_string = "char string";

MyString str\_1,

str\_2("first message"),

str\_3(list),

str\_4(string\_1),

str\_5(char\_string, 4),

str\_6(5, 'a'),

str\_copy(str\_2);

Assert::AreEqual(str\_1.c\_str(), "", "Constructor () - uncorrect");

Assert::IsTrue(str\_1.size() == 0, L"Constructor () - uncorrect");

Assert::AreEqual(str\_2.c\_str(), "first message", "Constructor (char) - uncorrect");

Assert::IsTrue(str\_2.size() == 13, L"Constructor (char) - uncorrect");

Assert::AreEqual(str\_3.c\_str(), "list string", "Constructor (list) - uncorrect");

Assert::IsTrue(str\_3.size() == 11, L"Constructor (list) - uncorrect");

Assert::AreEqual(str\_4.c\_str(), "second string", "Constructor (string) - uncorrect");

Assert::IsTrue(str\_4.size() == 13, L"Constructor (string) - uncorrect");

Assert::AreEqual(str\_4.c\_str(), "second string", "Constructor (string) - uncorrect");

Assert::IsTrue(str\_4.size() == 13, L"Constructor (string) - uncorrect");

Assert::AreEqual(str\_5.c\_str(), "char", "Constructor (“char string”, count) - uncorrect");

Assert::IsTrue(str\_5.size() == 4, L"Constructor (“char string”, count) - uncorrect");

Assert::AreEqual(str\_6.c\_str(), "aaaaa", "Constructor (count, char) - uncorrect");

Assert::IsTrue(str\_6.size() == 5, L"Constructor (count, char) - uncorrect");

Assert::IsTrue((str\_copy == str\_2), L"Constructor (copy) - uncorrect");

}

TEST\_METHOD(OperatorplusTest)

{

string string\_1 = "second string";

const char\* char\_string = "char string";

MyString str\_1("first message");

MyString str\_2("third message");

MyString sum\_string;

sum\_string = str\_1 + " ";

Assert::AreEqual(sum\_string.c\_str(), "first message ", "Operator +(“char string”) - uncorrect");

Assert::IsTrue(sum\_string.size() == 14, L"Operator +(“char string”) - uncorrect");

sum\_string = sum\_string + string\_1 + " ";

Assert::AreEqual(sum\_string.c\_str(), "first message second string ", "Operator +(std::string) - uncorrect");

Assert::IsTrue(sum\_string.size() == 28, L"Operator +(std::string) - uncorrect");

sum\_string = sum\_string + str\_2;

Assert::AreEqual(sum\_string.c\_str(), "first message second string third message", "Operator +(MyString) - uncorrect");

Assert::IsTrue(sum\_string.size() == 41, L"Operator +(MyString) - uncorrect");

Assert::IsTrue(string\_1 == "second string");

Assert::IsTrue(char\_string == "char string");

Assert::IsTrue(str\_1 == "first message");

Assert::IsTrue(str\_2 == "third message");

Assert::IsTrue(sum\_string == "first message second string third message");

}

TEST\_METHOD(OperatorplusequallyTest)

{

const char\* char\_string = "char string";

string string\_1 = "second string";

initializer\_list<char> list = { 'l', 'i', 's', 't', ' ', 's', 't', 'r', 'i', 'n', 'g' };

MyString sum\_string;

sum\_string += char\_string;

sum\_string += " ";

Assert::IsTrue(char\_string == "char string");

Assert::IsTrue(sum\_string == "char string ");

sum\_string += string\_1;

sum\_string += " ";

Assert::IsTrue(string\_1 == "second string");

Assert::IsTrue(sum\_string == "char string second string ");

sum\_string += list;

Assert::AreEqual(sum\_string.c\_str(), "char string second string list string", "Operator +=() - uncorrect");

Assert::IsTrue(sum\_string.size() == 37, L"Operator +=() - uncorrect");

}

TEST\_METHOD(OperatorassignmentTest)

{

MyString equally;

MyString str\_1("char string");

MyString str\_2("second string");

MyString str\_c("c");

string string\_1 = "second string";

equally = "char string";

Assert::IsTrue(equally == str\_1, L"Operator =(“char string”) - uncorrect");

equally = string\_1;

Assert::IsTrue(equally == str\_2, L"Operator =(std::string) - uncorrect");

equally = 'c';

Assert::IsTrue(equally == str\_c, L"Operator =(‘char’) - uncorrect");

}

TEST\_METHOD(OperatorindexTest)

{

MyString str\_1("char string");

Assert::IsTrue(str\_1[0] == 'c', L"Operator[] - uncorrect");

Assert::IsTrue(str\_1[12] == '\0', L"Operator[] - uncorrect");

}

TEST\_METHOD(OperatorboolTest)

{

MyString first("stringa"), second("stringb");

Assert::IsTrue(first < second, L"Operator <() - uncorrect");

Assert::IsTrue(!(first > second), L"Operator >() - uncorrect");

Assert::IsTrue(first != second, L"Operator !=() - uncorrect");

Assert::IsTrue(!(first == second), L"Operator ==() - uncorrect");

Assert::IsTrue(!(first >= second), L"Operator >=() - uncorrect");

Assert::IsTrue(first <= second, L"Operator <=() - uncorrect");

}

TEST\_METHOD(MethodsTest)

{

MyString str("first message");

MyString str\_empty;

Assert::AreEqual(str.c\_str(), "first message", "c\_str() - uncorrect");

Assert::IsTrue(str.size() == 13, L"size() - uncorrect");

Assert::IsTrue(str.size() == str.length(), L"length() - uncorrect");

Assert::IsTrue(str.c\_str() == str.data(), L"data() - uncorrect");

Assert::IsTrue(!(str.empty()), L"empty() - uncorrect");

str.change\_capacity();

Assert::IsTrue(str.capacity() == 15, L"capacity() - uncorrect");

str.shrink\_to\_fit();

Assert::IsTrue(str.capacity() == 13, L"shrink\_to\_fit() - uncorrect");

str.clear();

Assert::IsTrue(str == str\_empty, L"clear() - uncorrect");

}

TEST\_METHOD(InsertTest)

{

MyString s("aaaaa");

s.insert(0, 1, '!');

s.insert(3, 2, '@');

Assert::IsTrue(s == "!aa@@aaa", L"insert() - uncorrect");

s = "aaaaa";

s.insert(1, "@@@@@");

Assert::IsTrue(s == "a@@@@@aaaa", L"insert() - uncorrect");

s = "aaaaa";

s.insert(1, "@@@@@", 2);

Assert::IsTrue(s == "a@@aaaa", L"insert() - uncorrect");

s = "aaaaa";

s.insert(1, std::string("@@@@@"));

Assert::IsTrue(s == "a@@@@@aaaa", L"insert() - uncorrect");

s = "aaaaa";

s.insert(1, std::string("@@@@@"), 2);

Assert::IsTrue(s == "a@@aaaa", L"insert() - uncorrect");

}

TEST\_METHOD(EraseTest)

{

MyString s("aa@@@@@aaa");

s.erase(2, 3);

Assert::IsTrue(s == "aa@@aaa", L"erase() - uncorrect");

}

TEST\_METHOD(AppendTest)

{

MyString s;

s.clear();

s.append(3, '!');

s.append(3, '@');

Assert::IsTrue(s == "!!!@@@", L"append() - uncorrect");

s.clear();

s.append("hello ");

s.append("world");

Assert::IsTrue(s == "hello world", L"append() - uncorrect");

s.clear();

s.append("hello world", 0, 6);

s.append("hello world", 6, 5);

Assert::IsTrue(s == "hello world", L"append() - uncorrect");

string str = "hello ";

s.clear();

s.append(str);

str = "world";

s.append(str);

Assert::IsTrue(s == "hello world", L"append() - uncorrect");

s.clear();

str = "hello world";

s.append(str, 0, 6);

s.append(str, 6, 5);

Assert::IsTrue(s == "hello world", L"append() - uncorrect");

}

TEST\_METHOD(ReplaceTest)

{

MyString s;

s = "hello amazing world";

s.replace(6, 7, "wonderful");

Assert::IsTrue(s == "hello wonderful world", L"replace() - uncorrect");

s = "hello amazing world";

s.replace(6, 7, std::string("wonderful"));

Assert::IsTrue(s == "hello wonderful world", L"replace() - uncorrect");

}

TEST\_METHOD(SubstrTest)

{

MyString s;

s = "hello amazing world";

s = s.substr(6);

Assert::IsTrue(s == "amazing world", L"substr() - uncorrect");

s = "hello amazing world";

s = s.substr(6, 7);

Assert::IsTrue(s == "amazing", L"substr() - uncorrect");

}

TEST\_METHOD(FindTest)

{

MyString s;

s = "hello amazing world";

int find = s.find("amazing");

Assert::IsTrue(find == 6, L"find() - uncorrect");

find = s.find("o", 6);

Assert::IsTrue(find == 15, L"find() - uncorrect");

MyString str;

str = "amazing";

find = s.find(str);

Assert::IsTrue(find == 6, L"find() - uncorrect");

str = "o";

find = s.find(str, 6);

Assert::IsTrue(find == 15, L"find() - uncorrect");

}

};

}