1. Ministry of Education and Science of the Russian Federation
2. St. Petersburg State Polytechnic University
3. —
4. **Cybersecurity and Information Security department**

**REPORT**

**FOR Laboratory work № 1**

1. **«Class String»**
2. subject «OOP»
3. Completed by
4. student from gr. 4851003/90801 A.G. Kuleeva

<*signature*>

1. Checked by
2. teacher’s assistant A.D. Dahnovich

<*signature*>

1. Saint – Petersburg

2021

The **goal** of this laboratory work is to familiarize with OOP and implement string class in OOP style.

1. Task

The main tasks of this laboratory work are:

* implement a class for representing a character string in C++, not using the STL library containers and algorithms;
* extend python interpreter functionality with the implemented class.

2. Theory

The OOP paradigm has three main principles.

1. Encapsulation is needed to hide implementation from user and give him interface for interaction. Also encapsulation unite objects and methods,
2. Inheritance means that every daughter class has the same properties that the parent class,
3. Polymorphism means that we can work with different data types.

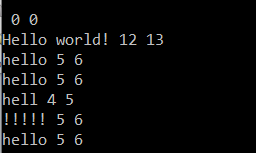
Also there are some other important definitions. Abstraction is a highlight of most important parts of object which help to solve task in optimum way. Class is an analog of structure in C language, there is a number of fields of different data types. But the main difference with struct is that class also has methods ― analogs of functions, which is already associated with objects of class.

Every class should has object constructor and destructor. Constructor creates a new object of class, allocates memory for it. Destructor is opposite to constructor, it deletes object and frees memory. If programmer doesn’t implement these both methods compiler creates default constructor and destructor, there behavior could be unpredictable.

Fields in class could be public or private. Any other user has access to public fields, for example object.field. If they are private another coder can interact with them, but not change them directly. Usually there are classic methods called setter (to set value) and getter (to get value), for example object.SetX(0) or x = object.GetX().

3. Implementation

The task was implemented with C++ language. Full code is placed in Appendix 1. Class MyString has private fields char \*string, size and capacity. In code there are c\_str(), size() and capacity() methods to get value of these fields. There is a number of constructors: default, char array, std::string, std::list, char array and count of elements, char symbol and count of elements, number of chars (see Pic 1).



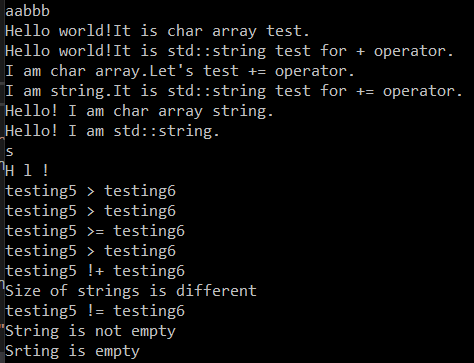
Picture 1 ― Testing constructors

Special attention was paid to copy constructor. On Pic 2 there are addresses in memory so process of copying is quite clear.



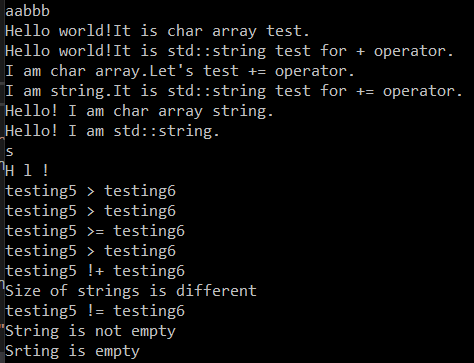
Picture 2 ― Testing copy constructor

On Pic 3 there are test results of operators +, +=, = and []. This called overload in C++. Also I needed to overload input and output operators.



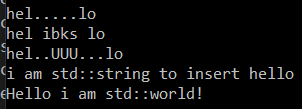
Picture 3 ― Testing +, +=, =, [ ] operators

On Pic 4 there is lexicographic comparing of strings. To implement this part I needed to search std::string lib in C++. Also on this Pic there is demonstration of empty() method.



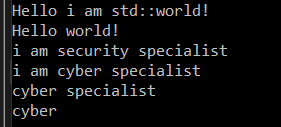
Picture 4 ― Testing lexicographic compare and empty()

On Pic 5 there is testing of insert() by index methods. User can insert std::string, char array, number of chars and number of elements in array or std::string.



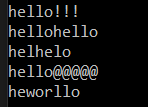
Picture 5 ― Testing insert()

Methods erase(), replace(), substr() was also implemented in MyString class, see Pic 6 for demonstration. Note, that replace() is based on erase() and insert(). Substr() returns string by index.



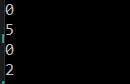
Picture 6 ― Testing erase(), replace(), substr()

Method append() implemented on the base of insert(), but index is also last element in string (Pic 7).



Picture 7 ― Testing append()

Method find() returns index of sought string (Pic 8).



Picture 8 ― Testing find()

4. Extra task

The goal of extra task was to create Python wrapping around implemented class. I decided to use SWIG. First of all setup and unpack tool. Then create file string.i and write next lines:

/\* File : string.i \*/

%module string

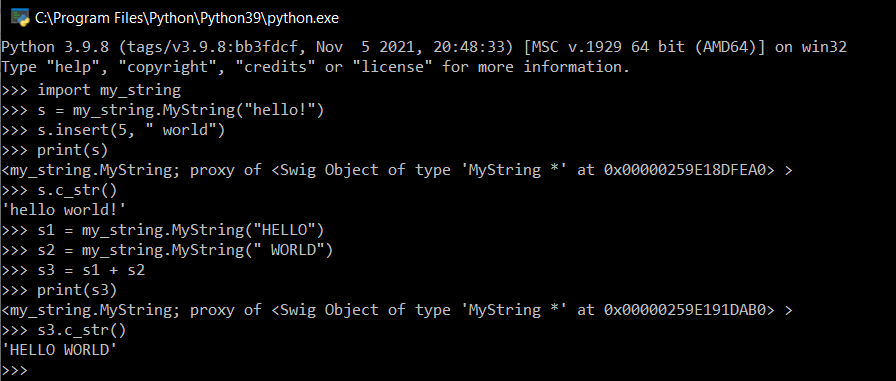
%{

#include "string.h"

%}

%include "string.h"

Then in cmd line we should to write swig -c++ -python string.i. This creates two different files; a C++ source file string\_wrap.cxx and a Python source file string.py. The generated C++ source file contains the low-level wrappers that need to be compiled and linked with the rest of your C/C++ application to create an extension module. The Python source file contains high-level support code. This is the file that you will import to use the module. Finally you should have string.py and \_string.pyd. Both of them import to C:\..\Python\Lib. Now they are ready to use! It’s important to mention that Python don’t have a couple of methods, for example =, +=, << and >>.



Picture 9 ― Example of usage

Appendix

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include "string.h"

#include <string.h>

#include <string>

MyString::MyString()

{

//std::cout << "Construct: " << this << std::endl;

this->str = NULL;

this->size\_ = 0;

this->capacity\_ = 0;

}

MyString::MyString(const char \*str)

{

//std::cout << "Construct: " << this << std::endl;

this->size\_ = strlen(str);

this->capacity\_ = this->size\_ + 1;

this->str = new char[this->size\_ + 1];

strncpy(this->str, str, this->size\_);

this->str[this->size\_] = '\0';

}

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

{

unsigned get\_size = list.size();

this->str = new char[get\_size + 1];

this->size\_ = get\_size;

this->capacity\_ = get\_size + 1;

unsigned int i = 0;

for (auto &element : list)

{

this->str[i] = element;

++i;

}

this->str[i] = '\0';

}

MyString::MyString(std::string s)

{

//std::cout << "Construct: " << this << std::endl;

this->size\_ = s.size();

this->capacity\_ = this->size\_ + 1;

this->str = new char[this->capacity\_];

unsigned int i = 0;

for (; i < this->size\_; i++)

this->str[i] = s[i];

this->str[i] = '\0';

}

MyString::MyString(const char \*str, unsigned int count)

{

this->str = new char[count + 1];

this->size\_ = count;

this->capacity\_ = count + 1;

strncpy(this->str, str, count);

this->str[count] = '\0';

}

MyString::MyString(unsigned int count, const char c)

{

//std::cout << "Construct: " << this << std::endl;

this->str = new char[count + 1];

unsigned int i = 0;

for (; i < count; i++)

this->str[i] = c;

this->str[i] = '\0';

this->size\_ = count;

this->capacity\_ = count + 1;

}

MyString::MyString(const MyString &other)

{

//std::cout << "Construct: " << this << "; From: " << &other << std::endl;

this->size\_ = other.size\_;

this->capacity\_ = other.capacity\_;

if (this->capacity\_ == 0)

{

this->str = NULL;

}

else {

this->str = new char[other.capacity\_];

strncpy(this->str, other.str, this->size\_);

this->str[this->size\_] = '\0';

}

}

MyString::~MyString()

{

//std::cout << "Destruct: " << this << std::endl;

//if (this->str[0] == '\0') return;

delete[] this->str;

}

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

{

if (other.str == NULL) return \*this;

unsigned int new\_size = this->size\_ + other.size\_;

MyString new\_string;

new\_string.str = new char[new\_size + 1];

new\_string.size\_ = new\_size;

new\_string.capacity\_ = new\_size + 1;

unsigned int i = 0;

for (; i < this->size\_; i++)

{

new\_string.str[i] = this->str[i];

}

for (unsigned int j = 0; j < other.size\_; j++, i++)

{

new\_string.str[i] = other.str[j];

}

new\_string.str[i] = '\0';

return new\_string;

}

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

{

unsigned int new\_size = this->size\_ + strlen(string);

MyString new\_string;

new\_string.str = new char[new\_size + 1];

new\_string.size\_ = new\_size;

new\_string.capacity\_ = new\_size + 1;

unsigned int i = 0;

for (; i < this->size\_; i++)

{

new\_string.str[i] = this->str[i];

}

for (unsigned int j = 0; j < strlen(string); j++, i++)

{

new\_string.str[i] = string[j];

}

new\_string.str[i] = '\0';

return new\_string;

}

MyString MyString::operator + (std::string s)

{

unsigned int new\_size = this->size\_ + s.size();

MyString new\_string;

new\_string.str = new char[new\_size + 1];

new\_string.size\_ = new\_size;

new\_string.capacity\_ = new\_size + 1;

unsigned int i = 0;

for (; i < this->size\_; i++)

{

new\_string.str[i] = this->str[i];

}

for (unsigned int j = 0; j < s.size(); j++, i++)

{

new\_string.str[i] = s[j];

}

new\_string.str[i] = '\0';

return new\_string;

}

//new

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

{

if (this->capacity\_ >= this->size\_ + strlen(string))

{

for (unsigned int i = this->size\_, j = 0; j < strlen(string); j++, i++)

{

this->str[i] = string[j];

}

return \*this;

}

unsigned int new\_size = this->size\_ + strlen(string);

unsigned int first\_size = this->size\_;

char \*buf = new char[this->size\_];

memcpy(buf, this->str, this->size\_);

//delete[] this->str;

this->str = new char[new\_size + 1];

this->capacity\_ = new\_size + 1;

this->size\_ = new\_size;

unsigned int i = first\_size, j = 0;

memcpy(this->str, buf, this->size\_);

for (; i < new\_size; j++, i++)

{

this->str[i] = string[j];

}

this->str[i] = '\0';

delete[] buf;

return \*this;

}

//new

MyString MyString::operator += (std::string s)

{

if (this->capacity\_ >= this->size\_ + s.size())

{

for (unsigned int i = this->size\_, j = 0; j < s.size(); j++, i++)

{

this->str[i] = s[j];

}

return \*this;

}

unsigned int new\_size = this->size\_ + s.size();

unsigned int first\_size = this->size\_;

char \*buf = new char[this->size\_];

memcpy(buf, this->str, this->size\_);

//delete[] this->str;

this->str = new char[new\_size + 1];

this->capacity\_ = new\_size + 1;

this->size\_ = new\_size;

unsigned int i = first\_size, j = 0;

memcpy(this->str, buf, this->size\_);

for (; i < new\_size; j++, i++)

{

this->str[i] = s[j];

}

this->str[i] = '\0';

delete[] buf;

return \*this;

}

MyString MyString::operator = (MyString &&other)

{

this->operator = (other.c\_str());

return \*this;

}

//new

MyString MyString::operator = (const char \*string)

{

unsigned int new\_size = strlen(string);

if (this->capacity\_ < strlen(string))

{

delete[] this->str;

this->str = new char[new\_size + 1];

this->capacity\_ = new\_size + 1;

this->size\_ = new\_size;

}

memcpy(this->str, string, strlen(string));

this->str[new\_size] = '\0';

return \*this;

}

//new

MyString MyString::operator = (std::string s) //std::string assignment

{

return \*this = s.c\_str();

}

//new

MyString MyString::operator = (const char c) //char assignment

{

if (this->capacity\_ < 2)

{

delete[] this->str;

this->str = new char[2];

this->capacity\_ = 2;

}

this->size\_ = 1;

this->str[0] = c;

this->str[1] = '\0';

return \*this;

}

char MyString::operator[] (unsigned int index) //index operator

{

if (this->size\_ >= index)

{

return this->str[index];

}

else {

std::cout << "Error!" << std::endl;

return 1;

}

}

//all lexic. comp are new

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

{

unsigned int i = 0;

for (;(this->str[i] != this->str[this->size\_]) && (other.str[i] != other.str[other.size\_]); i++)

{

if (this->str[i] > other.str[i]) return true;

if (this->str[i] < other.str[i]) return false;

}

return (this->str[i] == this->str[this->size\_]) && (other.str[i] != other.str[other.size\_]);

}

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

{

return !(this->str > other.str);

}

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

{

if (this->str > other.str || this->str == other.str)

return true;

else

return false;

}

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

{

if (this->str < other.str || this->str == other.str)

return true;

else

return false;

}

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

{

if (this->size\_ != other.size\_)

{

std::cout << "Size of strings is different" << std::endl;

return true;

}

else

{

for (unsigned int i = 0; i < this->size\_; i++)

{

if (this->str[i] == other.str[i]) continue;

else

{

std::cout << "Contents of strings is different" << std::endl;

return true;

}

}

return false;

}

}

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

{

return !(this->str != other.str);

}

char\* MyString::c\_str()

{

return this->str;

}

char\* MyString::data()

{

return this->str;

}

unsigned int MyString::size()

{

return this->size\_;

}

unsigned int MyString::length()

{

return size();

}

unsigned int MyString::capacity()

{

return this->capacity\_;

}

bool MyString::empty()

{

return (this->size\_ == 0);

}

//new

void MyString::shrink\_to\_fit()

{

this->capacity\_ = this->size\_ + 1;

char \*buf = new char[this->size\_];

memcpy(buf, this->str, this->size\_);

delete[] this->str;

this->str = new char[this->capacity\_];

memcpy(this->str, buf, this->capacity\_);

}

void MyString::clear()

{

this->size\_ = 0;

for (unsigned int i = 0; i < this->capacity\_; i++)

{

this->str[i] = 0;

}

}

std::ostream& operator << (std::ostream& out, MyString tmp)

{

if (tmp.size\_ == 0) out << "";

else out << tmp.str;

return out;

}

std::istream& operator >> (std::istream& in, MyString& tmp)

{

while (1)

{

unsigned int i = 0;

char \*buf = new char[1024];

for (; i < 1024; i++)

{

buf[i] = in.get();

if (buf[i] == '\n')

{

buf[i] = '\0';

break;

}

}

if (i < 1024)

{

tmp.str = new char[i];

tmp.str = buf;

tmp.size\_ = i - 1;

tmp.capacity\_ = i;

break;

}

else tmp.str += \*buf;

}

return in;

}

//new

MyString MyString::insert(unsigned int index, unsigned int count, char c)

{

char \*buf = new char[count];

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

{

buf[i] = c;

}

buf[count] = '\0';

//delete[] buf;

return insert(index, buf, count);

}

MyString MyString::insert(unsigned int index, const char\* string)

{

return insert(index, string, strlen(string));

}

//new

MyString MyString::insert(unsigned int index, const char\* string, unsigned int count)

{

unsigned int i = index, j = 0;

char \*buf = new char[this->size\_];

memcpy(buf, this->str, this->size\_);

unsigned int new\_size = this->size\_ + count;

unsigned int old\_cap = this->capacity\_;

this->capacity\_ = new\_size + 1;

this->size\_ = new\_size;

if (old\_cap <= this->size\_ + count)

{

unsigned int first\_size = this->size\_;

this->str = new char[new\_size + 1];

memcpy(this->str, buf, index);

}

for (; j < count; i++, j++)

{

this->str[i] = string[j];

}

for (j = index; i < new\_size; j++, i++)

{

this->str[i] = buf[j];

}

this->str[i] = '\0';

delete[] buf;

return \*this;

}

MyString MyString::insert(unsigned int index, std::string s)

{

return insert(index, s, s.size());

}

//new

MyString MyString::insert(unsigned int index, std::string s, unsigned int count) //c\_str() ->

{

const char \*buf = s.c\_str();

//delete[] buf;

return insert(index, buf, count);

}

MyString MyString::erase(unsigned int index, unsigned int count)

{

unsigned int next\_index = index + count;

for (unsigned int i = index, j = next\_index; j < this->capacity\_; i++, j++)

{

this->str[i] = this->str[j];

if (this->str[j] == '/0') break;

}

return \*this;

}

MyString MyString::replace(unsigned int index, unsigned int count, const char \*string)

{

this->erase(index, count);

this->insert(index, string);

return \*this;

}

MyString MyString::replace(unsigned int index, unsigned int count, std::string s)

{

this->erase(index, count);

this->insert(index, s);

return \*this;

}

//new

MyString MyString::substr(unsigned int index)

{

MyString new\_str(&str[index]);

return new\_str;

// new\_str;

}

//new

MyString MyString::substr(unsigned int index, unsigned int count)

{

MyString new\_str(&str[index], count);

return new\_str;

}

MyString MyString::append(unsigned int count, char c)

{

return this->insert(this->size\_, count, c);

}

MyString MyString::append(const char \*string)

{

return this->insert(this->size\_, string);

}

MyString MyString::append(const char \*string, unsigned int index, unsigned int count)

{

return this->insert(index, string, count);

}

MyString MyString::append(std::string s)

{

return this->insert(this->size\_, s);

}

MyString MyString::append(std::string s, unsigned int index, unsigned int count)

{

return this->insert(index, s, count);

}

unsigned int MyString::find(const char \*string)

{

unsigned int r = this->find(string, 0);

return r;

}

unsigned int MyString::find(const char \*string, unsigned int index)

{

if (strlen(string) > this->size\_)

{

std::cout << "No such substr" << std::endl;

return 0;

}

unsigned int t;

bool flag = false;

for (unsigned int i = index, j = 0; i < this->size\_; i++)

{

if (this->str[i] == string[j])

{

flag = true;

j++;

t = i;

}

else {

flag = false;

j = 0;

}

if (j == strlen(string)) break;

}

if (flag) return t + 1 - strlen(string);

else {

std::cout << "No such substr" << std::endl;

return 0;

}

}

unsigned int MyString::find(std::string s)

{

unsigned int r = this->find(s, 0);

return r;

}

unsigned int MyString::find(std::string s, unsigned int index)

{

if (s.size() > this->size\_)

{

std::cout << "No such substr" << std::endl;

return 0;

}

unsigned int t;

bool flag = false;

for (unsigned int i = index, j = 0; i < this->size\_; i++)

{

if (this->str[i] == s[j])

{

flag = true;

j++;

t = i;

}

else {

flag = false;

j = 0;

}

if (j == s.size()) break;

}

if (flag) return t + 1 - s.size();

else {

std::cout << "No such substr" << std::endl;

return 0;

}

}