# Task

Implement a class for representation a character string in C++. You should not use the STL library containers and algorithms.

Extend python interpreter functionality with implemented class (you can use Boost, SWIG, SIP, Pybind, CXX or whatever you want).

# Results

Part I. Class MyString

I studied the C++11 standard and the logic of string class. String could contain any data type, not only letters. It could contain integer or float list, symbol array, etc. So, my class MyString use not null-terminated data attribute.

Class MyString contains three attributes: dynamic array “*elems*” (not null-terminated string), integer “*count*” (the size of the string) and dynamic array “*c\_str\_elems*” (the null-terminated string from “*elems*”, if string is char, or null, if string is not char).

Class MyString contains several constructors:

* *MyString()*;
* *MyString(MyString &str)*;
* *MyString(char \*str)*;
* *MyString(std::initializer\_list<T> list)*;
* *MyString(std::string str)*;
* *MyString(char \*str, int count)*; this constructor initializes the string with *count* symbols of *str*;
* *MyString(T \*data, int count, int flag)*; this constructor exists for my own service;
* *MyString(int count, char simbol)*; this constructor initializes the string with *count* amount of *symbols*.

Class MyString contains overloading for all arithmetic, comparison, assignment operators and member access *operator [ ]*.

Class MyString contains overloading for following methods:

* *const char \*c\_str()*; it returns a pointer to null-terminated character array;
* *const T\* data()*; it returns a pointer to array data that is not required to be null-terminated;
* *int size()*; it returns the number of char elements in string
* *int length()*; it is same as size;
* *bool empty()*; it returns true if string is empty;
* *int capacity()*; it returns the current amount of allocated memory for array;
* *void clear()*; it removes all char elements in a string.

I did not implement method *shrink\_to\_fit()*, because I reallocate memory every time the string size changes. Also, the amount of ‘\0’ symbols at the end of the string could not be deleted even in this method, because I mean that class MyString could contain not only char symbols, but also numbers, so ‘\0’ symbols at the end of the string could not be just unnecessary extra symbols which could be deleted without consequences.

Class MyString contains several implementations for *insert()*, *append()*, *erase()*, *replace()*, *substr()*, *find()* functions with different parameters.

C++ code of class MyString is given in the Appendix 1.

Part II. Python interpreter

I use Boost.Python to extend Python interpreter functionality with implemented class. I studied Python data model.

To wrap class MyString into Python firstly I implemented template function *TemplateFunc<>()*. It contains the following types of wrapping definitions:

* *.def(init<>())*; these definitions describe all containers;
* *.def(self + std::string())*; this template could be used for wrapping the operators;
* *.def("c\_str", &MyString<T>::c\_str)*; this template is used for wrapping void methods without parameters;
* *.def("substr", static\_cast< const char \* (MyString<T>::\*)(int) > (&MyString<T>::substr), args("index"), return\_value\_policy<return\_by\_value>())*; this template is used for wrapping methods with parameters and return value.

The only operator that could not be wrapped as others is *operator [ ]*. There is no operation for wrapping this operator in Boost.Python. So, I wrap it like method, but using key-world *\_\_getitem\_\_*:

*.def("\_\_getitem\_\_", static\_cast< T& (MyString<T>::\*)(int) > (&MyString<T>::operator[]), args("i"), return\_value\_policy<return\_by\_value>()).*

Some of MyString methods return the value of type char \*. It is impossible to use this type in Python. So I added the type converter. It convert char array to bytes for Python specifically by using function *PyBytes\_FromStringAndSize().*

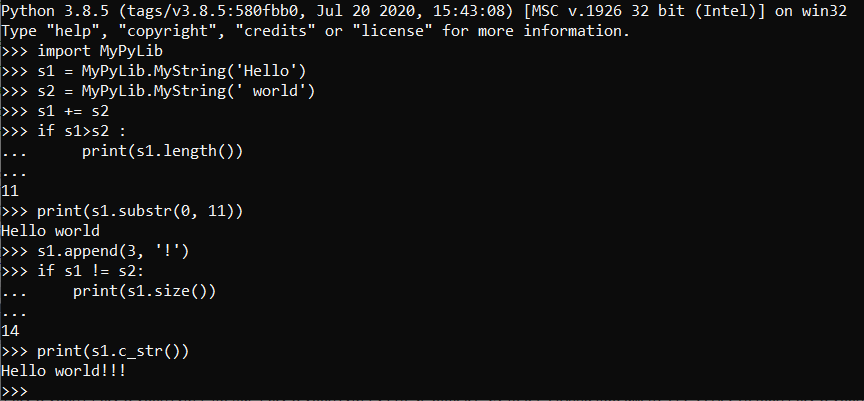
The demonstration of some methods of the class you can see in the Figure 1. Code of wrapping class MyString into Python is given in the Appendix 2.

Figure 1. The demonstration.

# Summary

I implemented class MyString for representation a character string in C++. I did not use the STL library containers and algorithms, data is stored in the dynamic array *elems*. I also studied the C++11 standard for string.

I extended Python interpreter functionality with implemented class. Using Boost I created a library MyPyLib so I can use my class MyString in Python.

# Appendix 1

// MyString.h

#pragma once

#include <iostream>

#include <string>

template <typename T>

class MyString;

template <typename T>

class MyString

{

private:

int count = 0;

T\* elems;

char\* c\_str\_elems = NULL;

public:

MyString() : count(0) { elems = new T[0]; }

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

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

MyString(std::initializer\_list<T> list); // initializer list constructor

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

MyString(char \*str, int count); // init class with count characters of char \*str,Constructor(“hello”, 4) -> “hell”

MyString(T \*data, int count, int flag); // for own service

MyString(int count, char simbol); // init class with count of characters MyString(4, ‘a’)->“aaa”

~MyString();

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

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

T\* operator + (std::string str); // concatenate with std::string

T\* operator += (const MyString &str); // assignment with Mystring

T\* operator += (char \*str); // assignment concatenate with char array

T\* operator += (std::string str); // assignment concatenate with std::string

T\* operator = (const MyString &str); // Mystring assignment

T\* operator = (char \*str); // char string assignment

T\* operator = (std::string str); // std::string assignment

T\* operator = (char simbol); // char assignment

T& operator[](int i); // index operator, Mystring(“012345”);[5] ->‘5’

bool operator>(char\* str);

bool operator>(std::string str);

bool operator>(const MyString &str);

bool operator<(char\* str);

bool operator<(std::string str);

bool operator<(const MyString &str);

bool operator>=(char\* str);

bool operator>=(std::string str);

bool operator>=(const MyString &str);

bool operator<=(char\* str);

bool operator<=(std::string str);

bool operator<=(const MyString &str);

bool operator==(char\* str);

bool operator==(std::string str);

bool operator==(const MyString &str);

bool operator!=(char\* str);

bool operator!=(std::string str);

bool operator!=(const MyString &str);

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

const T\* 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 string is empty

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

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

void insert(int index, int count, char simbol); // insert count of char in index position s = “aaaaa” \

s.insert(0, 1, ’!’)->“!aaaaa” \

s.insert(3, 2, ’@’)->“!aa@@aaa”

void insert(int index, char \*str); // insert null-terminated char string at index position\

s = “aaaaa” \

s.insert(1, ”@@@@@”)->“a@@@@@aaaa”

void insert(int index, char \*str, int l\_count); // insert count of null-terminated char string at index position \

s = “aaaaa” \

s.insert(1, ”@@@@@”, 2)->“a@@aaaa”

void insert(int index, std::string str); // insert std::string at index position \

s = “aaaaa” \

string = “@@@@@” \

s.insert(1, std::string(“@@@@@”))->“a@@@@@aaaa”

void insert(int index, std::string str, int l\_count); // insert count of std::string at index position \

s = “aaaaa” \

s.insert(1, std::string(“@@@@@”)), 2)->“a@@aaaa”

void erase(int index, int l\_count); // erase count of char at index position \

s = “aa@@@@@aaa” \

s.erase(2, 3)->“a@@aaaa”

void append(int count, char simbol); // append count of char \

str.clear()->“” \

str.append(3, ’!’)->“!!!” \

str.append(3, ’@’)->“!!!@@@”

void append(char \*str); // append null-terminated char string \

str.clear()->“” \

str.append(“hello ”)->“hello ” \

str.append(“world”)->“hello world”

void append(char \*str, int index, int count); // append a count of null-terminated char string by index position \

str.clear()->“” \

str.append(“hello world”, 0, 6)->“hello ” \

str.append(“hello world”, 6, 5)->“hello world”

void append(std::string str); // append std:: string \

str.clear()->“” \

str.append(std::string(“hello ”))->“hello ” \

str.append(std::string(“world”))->“hello world”

void append(std::string str, int index, int count); // append a count of std:: string by index position \

str.clear()->“” \

str.append(std::string(“hello world”), 0, 6)->“hello ” \

str.append(std::string(“hello world”), 6, 5)->“hello world”

void replace(int index, int count, char \*str); // replace a count of char at index by char \*str \

s = “hello amazing world” \

s.replace(6, 7, ”wonderful”)->“hello wonderful world”

void replace(int index, int count, std::string str); // replace a count of char at index by std::string \

s = “hello amazing world” \

s.replace(6, 7, std::string(”wonderful”))->“hello wonderful world”

const char \* substr(int index); //return a substring starts with index position \

s = “hello amazing world” \

s.substr(6)->“amazing world”

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

s = “hello amazing world” \

s.substr(6, 7)->“amazing”

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

int find(char \*str, int index); // same as find(char \*str) but search starts from index position

int find(std::string str); // if founded return the index of substring

int find(std::string str, int index); // same as find(char \*str) but search starts from index position

};

// copy constructor

template<typename T>

MyString<T>::MyString(const MyString & str)

{

count = str.count;

elems = new T[count];

memmove(elems, str.elems, count);

}

// char array constructor

template <typename T>

MyString<T>::MyString(char \*str) // char array constructor

{

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

{

if (str[i] != 0) count++;

else break;

}

elems = new T[count];

memmove(elems, str, count);

}

// initializer list constructor

template<typename T>

MyString<T>::MyString(std::initializer\_list<T> list)

{

count = list.size();

elems = new T[count];

T\* point = elems;

for (auto it = list.begin(); it != list.end(); ++it, ++point)

{

\*point = \*(const\_cast<T \*>(it));

}

}

// std::string constructor

template<typename T>

MyString<T>::MyString(std::string str)

{

count = str.length();

elems = new T[count];

memcpy(elems, str.c\_str(), count);

}

// init class with count characters of char \*str,Constructor(“hello”, 4) -> “hell”

template<typename T>

MyString<T>::MyString(char \* str, int count)

{

this->count = count;

elems = new T[count];

memmove(elems, str, count);

}

// any-type array constructor, for own service

template <typename T>

MyString<T>::MyString(T \*data, int count, int flag)

{

this->count = count;

elems = new T[count];

memcpy(elems, data, count \* sizeof(T));

}

// init class with count of characters MyString(4, ‘a’)->“aaa”

template<typename T>

MyString<T>::MyString(int count, char simbol)

{

this->count = count;

elems = new T[count];

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

elems[i] = simbol;

}

template <typename T>

MyString<T>::~MyString()

{

if (count != 0)

{

delete[] elems;

if (c\_str\_elems != NULL)

delete[] c\_str\_elems;

}

}

// concatenate with MyString

template <typename T>

T\* MyString<T>::operator + (const MyString &str)

{

T\* new\_elems = new T[count + str.count];

memcpy(new\_elems, elems, count \* sizeof(T));

memcpy(new\_elems + count, str.elems, str.count \* sizeof(T));

return new\_elems;

}

template<typename T>

T\* MyString<T>::operator+(char \* str)

{

// determine str size

int l\_count = 0;

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

{

if (str[i] != 0) l\_count++;

else break;

}

// create new char\* new\_str

T\* new\_str = new T[count + l\_count];

memcpy(new\_str, elems, count \* sizeof(T));

memcpy(new\_str + count, str, l\_count \* sizeof(T));

return new\_str;

}

template<typename T>

T\* MyString<T>::operator+(std::string str)

{

// create new char\* new\_str

T\* new\_str = new T[count + str.length()];

memcpy(new\_str, elems, count \* sizeof(T));

memcpy(new\_str + count \* sizeof(T), str.c\_str(), str.length() \* sizeof(T));

return new\_str;

}

template<typename T>

T \* MyString<T>::operator+=(const MyString & str)

{

T\* new\_elems = new T[count + str.count];

memcpy(new\_elems, elems, count \* sizeof(T));

memcpy(new\_elems + count \* sizeof(T), str.elems, str.count \* sizeof(T));

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

count += str.count;

return new\_elems;

}

template<typename T>

T \* MyString<T>::operator+=(char \* str)

{

// determine str size

int l\_count = 0;

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

{

if (str[i] != 0) l\_count++;

else break;

}

// expand the elems

T\* new\_elems = new T[count + l\_count];

memcpy(new\_elems, elems, count \* sizeof(T));

memcpy(new\_elems + count \* sizeof(T), str, l\_count \* sizeof(T));

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

count += l\_count;

return new\_elems;

}

template<typename T>

T \* MyString<T>::operator+=(std::string str)

{

// expand the elems

T\* new\_elems = new T[count + str.length()];

memcpy(new\_elems, elems, count \* sizeof(T));

memcpy(new\_elems + count \* sizeof(T), str.c\_str(), str.length() \* sizeof(T));

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

count += str.length();

return new\_elems;

}

template<typename T>

T \* MyString<T>::operator=(const MyString & str)

{

T\* new\_elems = new T[str.count];

memcpy(new\_elems, str.elems, str.count \* sizeof(T));

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

count = str.count;

return new\_elems;

}

template<typename T>

T\* MyString<T>::operator=(char \* str)

{

// determine str size

int l\_count = 0;

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

{

if (str[i] != 0) l\_count++;

else break;

}

// expand or narrow the elems

T\* new\_elems = new T[l\_count];

memcpy(new\_elems, str, l\_count \* sizeof(T));

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

count = l\_count;

return new\_elems;

}

template<typename T>

T \* MyString<T>::operator=(std::string str)

{

// expand or narrow the elems

T\* new\_elems = new T[str.length()];

memcpy(new\_elems, str.c\_str(), str.length() \* sizeof(T));

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

count = str.length();

return new\_elems;

}

template<typename T>

T \* MyString<T>::operator=(char simbol)

{

// expand or narrow the elems

T\* new\_elems = new T[1];

new\_elems[0] = simbol;

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

count = 1;

return new\_elems;

}

template<typename T>

T & MyString<T>::operator[](int i)

{

return elems[i];

}

template<typename T>

bool MyString<T>::operator>(char \* str)

{

// determine str size

int l\_count = 0;

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

{

if (str[i] != 0) l\_count++;

else break;

}

// compare

if (count > l\_count) return true;

return false;

}

template<typename T>

bool MyString<T>::operator>(std::string str)

{

if (count > str.length()) return true;

return false;

}

template<typename T>

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

{

if (count > str.count) return true;

return false;

}

template<typename T>

bool MyString<T>::operator<(char \* str)

{

return !operator>=(str);

}

template<typename T>

bool MyString<T>::operator<(std::string str)

{

return !operator>=(str);

}

template<typename T>

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

{

return !operator>=(str);

}

template<typename T>

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

{

// determine str size

int l\_count = 0;

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

{

if (str[i] != 0) l\_count++;

else break;

}

// compare

if (count >= l\_count) return true;

return false;

}

template<typename T>

bool MyString<T>::operator>=(std::string str)

{

if (count >= str.length()) return true;

return false;

}

template<typename T>

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

{

if (count >= str.count) return true;

return false;

}

template<typename T>

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

{

return !operator>(str);

}

template<typename T>

bool MyString<T>::operator<=(std::string str)

{

return !operator>(str);

}

template<typename T>

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

{

return !operator>(str);

}

template<typename T>

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

{

// determine str size

int l\_count = 0;

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

{

if (str[i] != 0) l\_count++;

else break;

}

// compare

if (count == l\_count && !strcmp(elems, str)) return true;

return false;

}

template<typename T>

bool MyString<T>::operator==(std::string str)

{

if (count == str.length() && !strcmp(elems, str.c\_str())) return true;

return false;

}

template<typename T>

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

{

if (count == str.count && !strcmp(elems, str.elems)) return true;

return false;

}

template<typename T>

bool MyString<T>::operator!=(char \* str)

{

return !operator==(str);

}

template<typename T>

bool MyString<T>::operator!=(std::string str)

{

return !operator=(str);

}

template<typename T>

bool MyString<T>::operator!=(const MyString & str)

{

return !operator==(str);

}

// return a pointer to null-terminated character array

template<typename T>

const char\* MyString<T>::c\_str()

{

if (!c\_str\_elems)

{

c\_str\_elems = new char[count + 1];

memcpy(c\_str\_elems, elems, count);

c\_str\_elems[count] = '\0';

}

else

{

if (strncmp(elems, c\_str\_elems, count) || count != strlen(c\_str\_elems))

{

delete[] c\_str\_elems;

c\_str\_elems = NULL;

return c\_str();

}

}

return static\_cast<const char \*> (c\_str\_elems);

}

// return a pointer to array data that not required to be null-terminated

template<typename T>

const T \* MyString<T>::data()

{

return elems;

}

template<typename T>

int MyString<T>::length()

{

return size();

}

template<typename T>

int MyString<T>::size()

{

return count;

}

template<typename T>

bool MyString<T>::empty()

{

if (!count) return true;

return false;

}

// return the current amount of allocated memory for array (in bytes)

template<typename T>

int MyString<T>::capacity()

{

return count \* sizeof(T);

}

template<typename T>

void MyString<T>::clear()

{

count = 0;

delete[] elems;

if (!c\_str\_elems) delete[] c\_str\_elems;

elems = new T[0];

}

// insert count of char in index position s = “aaaaa” \

s.insert(0, 1, ’!’)->“!aaaaa” \

s.insert(3, 2, ’@’)->“!aa@@aaa”

template<typename T>

void MyString<T>::insert(int index, int count, char simbol)

{

T\* new\_elems = new T[this->count + count];

memcpy(new\_elems, elems, index);

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

new\_elems[index + i] = simbol;

memcpy(new\_elems + (index + count) \* sizeof(T), elems + index \* sizeof(T), this->count - index);

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

this->count += count;

}

// insert null-terminated char string at index position\

s = “aaaaa” \

s.insert(1, ”@@@@@”)->“a@@@@@aaaa”

template<typename T>

void MyString<T>::insert(int index, char \* str)

{

// determine str size

int l\_count = 0;

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

{

if (str[i] != 0) l\_count++;

else break;

}

// insert

T\* new\_elems = new T[count + l\_count];

memcpy(new\_elems, elems, index);

memcpy(new\_elems + index \* sizeof(T), str, l\_count);

memcpy(new\_elems + (index + l\_count) \* sizeof(T), elems + index \* sizeof(T), count - index);

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

this->count += l\_count;

}

// insert count of null-terminated char string at index position \

s = “aaaaa” \

s.insert(1, ”@@@@@”, 2)->“a@@aaaa”

template<typename T>

void MyString<T>::insert(int index, char \* str, int l\_count)

{

T\* new\_elems = new T[count + l\_count];

memcpy(new\_elems, elems, index);

memcpy(new\_elems + index \* sizeof(T), str, l\_count);

memcpy(new\_elems + (index + l\_count) \* sizeof(T), elems + index \* sizeof(T), count - index);

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

this->count += l\_count;

}

// insert std::string at index position \

s = “aaaaa” \

string = “@@@@@” \

s.insert(1, std::string(“@@@@@”))->“a@@@@@aaaa”

template<typename T>

void MyString<T>::insert(int index, std::string str)

{

T\* new\_elems = new T[count + str.length()];

memcpy(new\_elems, elems, index);

memcpy(new\_elems + index \* sizeof(T), str.c\_str(), str.length());

memcpy(new\_elems + (index + str.length()) \* sizeof(T), elems + index \* sizeof(T), count - index);

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

this->count += str.length();

}

// insert count of std::string at index position \

s = “aaaaa” \

s.insert(1, std::string(“@@@@@”)), 2)->“a@@aaaa”

template<typename T>

void MyString<T>::insert(int index, std::string str, int l\_count)

{

T\* new\_elems = new T[count + l\_count];

memcpy(new\_elems, elems, index);

memcpy(new\_elems + index \* sizeof(T), str.c\_str(), l\_count);

memcpy(new\_elems + (index + l\_count) \* sizeof(T), elems + index \* sizeof(T), count - index);

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

this->count += l\_count;

}

// erase count of char at index position \

s = “aa@@@@@aaa” \

s.erase(2, 3)->“a@@aaaa”

template<typename T>

void MyString<T>::erase(int index, int l\_count)

{

T\* new\_elems = new T[count - l\_count];

memcpy(new\_elems, elems, index);

memcpy(new\_elems + index \* sizeof(T), elems + (index + l\_count) \* sizeof(T), count - index);

T\* old\_elems = elems;

elems = new\_elems;

delete[] old\_elems;

this->count -= l\_count;

}

// append count of char \

str.clear()->“” \

str.append(3, ’!’)->“!!!” \

str.append(3, ’@’)->“!!!@@@”

template<typename T>

void MyString<T>::append(int count, char simbol)

{

insert(this->count, count, simbol);

}

// append null-terminated char string \

str.clear()->“” \

str.append(“hello ”)->“hello ” \

str.append(“world”)->“hello world”

template<typename T>

void MyString<T>::append(char \* str)

{

insert(count, str);

}

// append a count of null-terminated char string by index position \

str.clear()->“” \

str.append(“hello world”, 0, 6)->“hello ” \

str.append(“hello world”, 6, 5)->“hello world”

template<typename T>

void MyString<T>::append(char \* str, int index, int count)

{

insert(this->count, str + index, count);

}

// append std:: string \

str.clear()->“” \

str.append(std::string(“hello ”))->“hello ” \

str.append(std::string(“world”))->“hello world”

template<typename T>

void MyString<T>::append(std::string str)

{

insert(count, str);

}

// append a count of std:: string by index position \

str.clear()->“” \

str.append(std::string(“hello world”), 0, 6)->“hello ” \

str.append(std::string(“hello world”), 6, 5)->“hello world”

template<typename T>

void MyString<T>::append(std::string str, int index, int count)

{

insert(this->count, str.c\_str() + index, count);

}

// replace a count of char at index by char \*str \

s = “hello amazing world” \

s.replace(6, 7, ”wonderful”)->“hello wonderful world”

template<typename T>

void MyString<T>::replace(int index, int count, char \* str)

{

erase(index, count);

insert(index, str);

}

// replace a count of char at index by std::string \

s = “hello amazing world” \

s.replace(6, 7, std::string(”wonderful”))->“hello wonderful world”

template<typename T>

void MyString<T>::replace(int index, int count, std::string str)

{

erase(index, count);

insert(index, str);

}

//return a substring starts with index position \

s = “hello amazing world” \

s.substr(6)->“amazing world”

template<typename T>

const char \* MyString<T>::substr(int index)

{

return substr(index, this->count - index);

}

// return a count of substring’s char starts with index position \

s = “hello amazing world” \

s.substr(6, 7)->“amazing”

template<typename T>

const char \* MyString<T>::substr(int index, int count)

{

if (!c\_str\_elems)

{

c\_str\_elems = new char[count + 1];

memcpy(c\_str\_elems, elems + index, count);

c\_str\_elems[count] = '\0';

}

else

{

delete[] c\_str\_elems;

c\_str\_elems = NULL;

return substr(index, count);

}

return c\_str\_elems;

}

// if founded return the index of substring

template<typename T>

int MyString<T>::find(char \* str)

{

return find(str, 0);

}

// if founded return the index of substring; search starts from index position

template<typename T>

int MyString<T>::find(char \* str, int index)

{

// determine str size

int l\_count = 0;

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

{

if (str[i] != 0) l\_count++;

else break;

}

// find

for (int i = index; i <= count - l\_count; i++)

{

if (!strncmp(elems + i, str, l\_count))

return i;

}

return -1;

}

// if founded return the index of substring

template<typename T>

int MyString<T>::find(std::string str)

{

return find(const\_cast<char \*>(str.c\_str()));

}

// if founded return the index of substring; search starts from index position

template<typename T>

int MyString<T>::find(std::string str, int index)

{

return find(const\_cast<char \*>(str.c\_str()), index);

}

template<typename T>

std::ostream & operator << (std::ostream & s, MyString<T>& str)

{

s << str.c\_str();

return s;

}

# Appendix 2

// Source.cpp

#define BOOST\_PYTHON\_STATIC\_LIB

#include "Sourсe.h"

#include <string>

#include <iostream>

#include <boost/python.hpp>

#include "MyString.h"

using namespace boost::python;

// type conversion

template< typename T >

struct type\_into\_python

{

static PyObject\* convert(T const&);

};

PyObject\* type\_into\_python<char \*>::convert(char \* const& t)

{

const char\* src = (strlen(t) == 0) ? "" : reinterpret\_cast<const char\*>(&t);

return PyBytes\_FromStringAndSize(src, strlen(t));

}

template <typename T>

void TemplateFunc()

{

class\_<MyString<T>>("MyString")

.def(init<>())

.def(init<const MyString<T> &>())

.def(init<std::initializer\_list<T> >())

.def(init<std::string >())

.def(init<char \*, int >())

.def(init<int, char >())

.def(self + self)

.def(self + std::string())

.def(self += self)

.def(self += std::string())

.def("\_\_getitem\_\_", static\_cast< T& (MyString<T>::\*)(int) > (&MyString<T>::operator[]), args("i"), return\_value\_policy<return\_by\_value>())

.def(self > std::string())

.def(self > self)

.def(self < std::string())

.def(self < self)

.def(self >= std::string())

.def(self >= self)

.def(self <= std::string())

.def(self <= self)

.def(self == std::string())

.def(self == self)

.def(self != std::string())

.def(self != self)

.def("c\_str", &MyString<T>::c\_str)

.def("data", &MyString<T>::data, return\_value\_policy<return\_by\_value>())

.def("length", &MyString<T>::length)

.def("size", &MyString<T>::size)

.def("empty", &MyString<T>::empty)

.def("capacity", &MyString<T>::capacity)

.def("clear", &MyString<T>::clear)

.def("insert", static\_cast< void (MyString<T>::\*)(int, int, char) > (&MyString<T>::insert), args("index", "count", "symbol"))

.def("insert", static\_cast< void (MyString<T>::\*)(int, std::string) > (&MyString<T>::insert), args("index", "str"))

.def("insert", static\_cast< void (MyString<T>::\*)(int, std::string, int) > (&MyString<T>::insert), args("index", "str", "l\_count"))

.def("erase", &MyString<T>::erase, args("index", "l\_count"))

.def("append", static\_cast< void (MyString<T>::\*)(int, char) > (&MyString<T>::append), args("count", "symbol"))

.def("append", static\_cast< void (MyString<T>::\*)(std::string) > (&MyString<T>::append), args("str"))

.def("append", static\_cast< void (MyString<T>::\*)(std::string, int, int) > (&MyString<T>::append), args("str", "index", "count"))

.def("replace", static\_cast< void (MyString<T>::\*)(int, int, std::string) > (&MyString<T>::replace), args("index", "count", "str"))

.def("substr", static\_cast< const char \* (MyString<T>::\*)(int) > (&MyString<T>::substr), args("index"), return\_value\_policy<return\_by\_value>())

.def("substr", static\_cast< const char \* (MyString<T>::\*)(int, int) > (&MyString<T>::substr), args("index", "xount"), return\_value\_policy<return\_by\_value>())

.def("find", static\_cast< int (MyString<T>::\*)(std::string) > (&MyString<T>::find), args("str"))

.def("find", static\_cast< int (MyString<T>::\*)(std::string, int) > (&MyString<T>::find), args("str", "index"))

;

}

BOOST\_PYTHON\_MODULE(MyPyLib)

{

to\_python\_converter< char \*, type\_into\_python<char \*> >();

TemplateFunc<char>();

}