Peter the Great St. Petersburg Polytechnic University

—

Graduate school of cybersecurity

**GUIDELINES FOR LABORATORY WORK № 1**

**«String class»**

course «Object-Oriented Programming»

1. Performed by

student of gr. 5151004/10101 Veremeichuk Y.Y.

<signature>

1. Checked by

teacher Kubrin G.S.

<signature>

1. Saint-Petersburg
2. 2023

1 INTRODUCTION

* 1. PURPOSE

The main tasks of this laboratory work is to implement a class for representing a character string in C++, not using the STL library containers and algorithms.

* 1. THEORY

Object-Oriented Programming (OOP) is a programming paradigm that focuses on organizing code into objects, which are instances of classes. It promotes concepts like encapsulation, inheritance, and polymorphism to create modular and reusable code.

Char arrays were used in this work, let's look at their pros and cons:

Pros: Low-level control, memory efficiency, and compatibility with C libraries.

Cons: Lack of safety checks, difficulty in handling strings, and potential buffer overflows.

Python uses a dynamic, high-level, and garbage-collected data model. It differs from C/C++ in being dynamically typed, memory-managed, and having built-in data structures like lists and dictionaries.

The Standard Template Library (STL) in C++ provides containers (like vectors and maps), algorithms (sorting and searching), and iterators to simplify common programming tasks.

"const," "friend," "&" in C++:

"const": Used to declare that a variable is immutable or a method does not modify an object.

"friend": Allows external functions or classes to access private members of a class.

"&": Used to declare a reference variable, allowing manipulation of the referenced object directly.

Operator overloading and inheritance:

Operator overloading: Allows defining custom behaviors for operators like +, -, \*, etc., for user-defined classes.

Inheritance: Enables a class to inherit properties and methods from a base class, promoting code reuse and hierarchy.

How the wrapping method works:

In the provided code, the C-style character array (char\*) was chosen as the underlying data structure for MyString class. This means that MyString objects internally store character data in dynamically allocated char arrays, and various member functions manipulate this array to perform string operations.

The wrapping process and the results (wrapping code):

The wrapping process involves encapsulating the C-style character array within the MyString class. Here's how it works:

There are some constructors that initialize MyString objects using various input sources, such as char arrays, std::strings, or initializer lists. These constructors allocate memory for the char array and copy the input data into it.

The concatenation, assignment, insertion, erasure, appending, and replacement operations involve managing the dynamic memory allocated for the internal char array. These operations ensure that the char array is resized and updated accordingly to maintain the string data.

Accessor functions like c\_str(), data(), size(), length(), capacity(), and empty() provide controlled access to the internal char array and relevant information about the string.

Memory management functions like clear() and shrink\_to\_fit() ensure that the allocated memory is correctly deallocated when no longer needed or resized as necessary to optimize memory usage.

Various operators, such as operator+, operator+=, operator=, operator[], and comparison operators, allow to work with MyString objects just like regular strings while maintaining control over the underlying char array.

Overall, this wrapping approach allows to use C-style character arrays as the foundation for MyString objects while adding functionality and memory management features to make working with strings safer and more convenient. It encapsulates the complexities of managing character arrays within a user-friendly interface.

2 MAIN BODY

2.1 STARTING

A class is a blueprint or a template for creating objects in object-oriented programming (OOP). It defines a set of attributes (data members) and methods (functions) that characterize an object's behavior. Classes serve as a way to encapsulate data and functionality into a single unit, promoting modularity, code reusability, and organization in software development.

The MyString class is a custom implementation of a string (character array) in the C++ programming language. This class provides the management of variable-length strings and offers various operations for working with them.

Private pointer str\_ to a character array is the actual storage for the characters in the string. Initially set to nullptr, indicating that the string contains no data. During the creation and manipulation of the string, this pointer allocates and deallocates memory based on the current length of the string.

The private field size\_ holds the number of characters (length) of the string. It keeps track of the current length of the string and is updated when operations such as insertion, deletion, and character replacement are performed. The length of the string does not include the null-terminating character.

This private field capacity\_ determines the current capacity (allocated memory) of the string. It indicates the number of characters that the current allocated memory for the string can hold. When necessary, the capacity is increased to meet the memory needs of the string during expansion operations.

2.2 CONSTRUCTIONS AND DESTRUCTOR

Constructors are special member functions in a class that are responsible for initializing objects of that class. They are called automatically when an object is created and allocate memory, set initial values, and perform any necessary setup. Constructors enable the creation of objects with proper initial states and help ensure that objects are ready to be used when they are instantiated.

Here are the constructors that were implemented:

MyString();

MyString(*const* char\* arr);

MyString(std::initializer\_list<char> il);

MyString(*const* std::string& arr);

MyString(*const* char\* arr, size\_t count);

MyString(size\_t count, char ch);

MyString(*const* MyString& arr);

**Default Constructor**

The default constructor initializes an empty MyString object. It sets the size and capacity to 0 and allocates no memory for the string.

MyString::**MyString**() *//default* *constructor*

{

std::cout << *\_\_PRETTY\_FUNCTION\_\_* << std::endl;

}

**Character Array Constructor**

The constructor takes a C-style character array as input. It calculates the length of the array, allocates memory for the string, copies the content, and sets the size and capacity accordingly.

MyString::**MyString**(*const* char\* arr) *//* *char* *array* *constructor*

{

std::cout << *\_\_PRETTY\_FUNCTION\_\_* << std::endl;

*if* (arr)

{

capacity\_ = strlen(arr) + 1;

str\_ = *new* char[capacity\_];

strcpy\_s(*str\_*, capacity\_, arr);

size\_ = capacity\_ - 1;

}

}

**Initializer List Constructor**

The constructor initializes the MyString object using an initializer list of characters. It determines the size and capacity based on the number of characters in the list, allocates memory, and copies the characters.

MyString::**MyString**(std::initializer\_list<char> il) *//* *initializer* *list* *constructor*

: size\_(il.size()), capacity\_(size\_ + 1)

{

std::cout << *\_\_PRETTY\_FUNCTION\_\_* << std::endl;

*if* (il.size())

{

str\_ = *new* char[capacity\_];

size\_t i = 0;

*for* (char ch : il)

{

str\_[i++] = ch;

}

str\_[i] = '\0';

}

*//* *no* *else.* *str\_* *==* *nullptr,* *size\_* *==* *capacity\_* *==* *0*

}

**std::string Constructor**

The constructor creates a MyString object from an std::string. It calculates the size and capacity based on the length of the std::string, allocates memory, and copies the content.

MyString::**MyString**(*const* std::string& arr) *//* *std::string* *constructor*

: size\_(arr.length()), capacity\_(size\_ + 1)

{

std::cout << *\_\_PRETTY\_FUNCTION\_\_* << std::endl;

CreateCopyArray(arr.c\_str(), capacity\_);

}

**Character Array Constructor with Count**

The constructor initializes the object with the specified number of characters from a C-style character array. It allocates memory, copies the characters up to the specified count, and sets the size and capacity accordingly.

MyString::**MyString**(*const* char\* arr, size\_t count) *//* *init* *class* *with* *count* *characters* *of* *“char* *string”*

: size\_(count), capacity\_(count + 1)

{

std::cout << *\_\_PRETTY\_FUNCTION\_\_* << std::endl;

str\_ = *new* char[capacity\_];

strncpy(*str\_*, arr, size\_);

str\_[size\_] = '\0';

}

**Character Constructor with Count**

The constructor creates a MyString object with the specified number of characters, all set to the same character. It allocates memory, initializes the characters, and sets the size and capacity accordingly.

MyString::**MyString**(size\_t count, char ch) *//* *init* *class* *with* *count* *of* *characters*

: size\_(count), capacity\_(count + 1)

{

std::cout << *\_\_PRETTY\_FUNCTION\_\_* << std::endl;

str\_ = *new* char[capacity\_];

*for* (size\_t i = 0; i < size\_; ++i)

{

str\_[i] = ch;

}

str\_[count] = '\0';

}

**Copy Constructor**

The copy constructor creates a new MyString object by copying the content of another MyString object. It allocates memory, copies the content, and sets the size and capacity to match the source object.

MyString::**MyString**(*const* MyString& ob) *//* *copy* *constructor*

: size\_(ob.size\_), capacity\_(ob.capacity\_)

{

std::cout << *\_\_PRETTY\_FUNCTION\_\_* << std::endl;

*if* (ob.str\_)

{

CreateCopyArray(ob.str\_, ob.capacity\_);

}

}

The CreateCopyArray() function is a private member-function solving the copy-paste problem.

void MyString::**CreateCopyArray**(*const* char\* arr, size\_t str\_size)

{

str\_ = *new* char[str\_size];

strcpy\_s(*str\_*, capacity\_, arr);

}

**Destructor**

Destructor is responsible for releasing any dynamically allocated memory used by an instance of the class. Its primary purpose is to perform cleanup operations when an object of the class goes out of scope or is explicitly destroyed.

MyString::~**MyString**()

{

std::cout << *\_\_PRETTY\_FUNCTION\_\_* << std::endl;

clear();

}

void MyString::**clear**() *//* *remove* *all* *char* *element* *in* *string*

{

*if* (str\_)

{

*delete*[] str\_;

str\_ = *nullptr*;

size\_ = 0;

capacity\_ = 0;

}

}

2.3 OPERATORS

Several operators were implemented in the class that allow performing various operations with myString objects.

**Concatenation Operators (+)**

The + operator was overloaded to allow concatenation of MyString objects with other MyString objects, C-style character arrays (char\*), and standard C++ strings (std::string). These operators create a new MyString object by merging two strings together. OperatorConcatenate() function is a private member-function solving the copy-paste problem.

MyString MyString::*operator*+(*const* MyString& ob) *//* *concatenate* *with* *Mystring*

{

MyString res;

OperatorConcatenate(*res*, ob.str\_, ob.size\_);

*return* res;

}

MyString MyString::*operator*+(*const* char\* arr) *//* *concatenate* *with* *char* *array*

{

MyString res;

OperatorConcatenate(*res*, arr, strlen(arr));

*return* res;

}

MyString MyString::*operator*+(*const* std::string& arr) *//* *concatenate* *with* *std::string*

{

MyString res;

OperatorConcatenate(*res*, arr.c\_str(), arr.size());

*return* res;

}

void MyString::**OperatorConcatenate**(MyString& ob, *const* char\* arr, size\_t str\_size)

{

ob.size\_ = size\_ + str\_size; *//* *the* *length* *of* *the* *str\_* *consists* *of* *the* *old* *length* *and* *the* *length* *of* *the* *added* *array/string/object*

ob.capacity\_ = ob.size\_ + 1;

ob.str\_ = *new* char[ob.capacity\_];

*if* (str\_)

strncpy(*ob.str\_*, str\_, capacity\_);

*else*

ob.str\_[0] = '\0';

strncat(*ob.str\_*, arr, str\_size);

ob.str\_[ob.size\_] = '\0';

}

**Assignment Operators (=)**

The = operator was overloaded to assign values from other MyString objects, C-style character arrays, and standard C++ strings to MyString objects. These operators manage memory allocation and copying appropriately. OperatorAssignment() is a private member-function solving the copy-paste problem.

MyString & MyString::*operator*=(*const* MyString& ob) *//* *copy* *assignment*

{

*//* *we* *must* *handle* *the* *case* *&ob* *==* *this* *(when* *an* *object* *assigns* *itself),* *otherwise* *in* *the*

*//* *function* *OperatorAssignment()* *memory* *will* *be* *cleared* *in* *str\_.*

*if* (&ob == *this*) *//* *checking* *the* *equality* *of* *addresses*

*return* \**this*;

OperatorAssignment(ob.str\_, ob.size\_);

*return* \**this*;

}

MyString & MyString::*operator*=(*const* char\* arr) *//* *char* *string* *assignment*

{

*if* (arr)

OperatorAssignment(arr, strlen(arr));

*else*

{

str\_[0] = '\0';

size\_ = 0;

}

*return* \**this*;

}

MyString & MyString::*operator*=(*const* std::string& arr) *//* *std::string* *assignment*

{

OperatorAssignment(arr.c\_str(), arr.size());

*return* \**this*;

}

MyString & MyString::*operator*=(*const* char ch) *//* *char* *assignment*

{

*//* *OperatorAssignment(ch,* *1);* *we* *can't* *use* *it* *since* *it's* *a* *symbol*

*if* (capacity\_ <= 1) *//* *there* *is* *no* *memory* *for* *the* *symbol.* *need* *to* *reallocate*

{

clear();

str\_ = *new* char[2]; *//* *we* *allocate* *memory* *only* *for* *the* *symbol*

capacity\_ = 2;

} *//* *otherwise,* *capacity\_* *remains* *the* *same,* *and* *we* *just* *add* *a* *character* *and* *end* *the* *string*

str\_[0] = ch;

str\_[1] = '\0';

size\_ = 1;

*return* \**this*;

}

void MyString::**OperatorAssignment**(*const* char\* arr, size\_t arr\_size)

{

*if* (capacity\_ > arr\_size) *//* *if* *there* *is* *enough* *free* *space,* *we* *do* *not* *reallocate* *the* *memory,* *just* *clear* *the* *array*

str\_[0] = '\0';

*else* *//* *otherwise,* *we* *delete* *everything* *and* *reallocate* *memory*

{

clear();

capacity\_ = arr\_size + 1;

str\_ = *new* char[capacity\_];

}

strcpy\_s(*str\_*, capacity\_, arr);

size\_ = arr\_size;

}

**Addition Assignment Operator (+=)**

The += operator is overloaded to perform in-place concatenation. It allows you to append MyString objects, C-style character arrays, or standard C++ strings to an existing MyString object without creating a new one. This operator handles memory management efficiently, either by extending the current buffer or reallocating memory when necessary. OperatorConcatenateAssignment() is a private member-function solving the copy-paste problem.

MyString & MyString::*operator*+=(*const* char\* arr)

{

*if* (arr)

{

size\_t arr\_size = strlen(arr);

size\_t new\_size = size\_ + arr\_size;

*if* (new\_size >= capacity\_)

{

*//* *Expand* *the* *buffer* *if* *it's* *not* *large* *enough*

size\_t new\_capacity = new\_size + 1;

char\* new\_str = *new* char[new\_capacity];

*//* *Copy* *the* *existing* *string* *to* *the* *new* *buffer*

*if* (str\_ != *nullptr*) {

strncpy(*new\_str*, str\_, size\_);

}

new\_str[size\_] = '\0';

*//* *Concatenate* *the* *new* *string* *to* *the* *end*

strncat(*new\_str*, arr, arr\_size);

new\_str[new\_capacity - 1] = '\0';

*//* *Clear* *the* *old* *data,* *update* *size* *and* *capacity,* *and* *assign* *the* *new* *buffer*

clear();

size\_ = new\_capacity - 1;

capacity\_ = new\_capacity;

str\_ = new\_str;

}

*else*

{

*//* *If* *there's* *enough* *space,* *simply* *concatenate* *the* *new* *string*

strncat(*str\_*, arr, arr\_size);

size\_ = new\_size;

}

}

*return* \**this*;

}

MyString & MyString::*operator*+=(*const* std::string& arr)

{

*return* \**this* += arr.c\_str();

}

**Indexing Operators ([])**

The [] operator allows both const and non-const access to individual characters of a MyString object. It performs bounds checking to ensure safe access.

char & MyString::*operator*[](size\_t index) *const* *//* *const* *index* *operator*

{

*//std::cout* *<<* *"const* *operator[]"* *<<* *std::endl;* *//* *for* *debugging.* *you* *can* *check* *that* *this* *particular* *operator* *is* *being* *called*

*if* (index < size\_)

{

*return* str\_[index];

}

*else*

{

*throw* std::out\_of\_range("Index out of range");

}

}

*const* char & MyString::*operator*[](size\_t index) *//* *index* *operator*

{

*//std::cout* *<<* *"not* *const* *operator[]"* *<<* *std::endl;*

*if* (index < size\_)

{

*return* str\_[index];

}

*else*

{

*throw* std::out\_of\_range("Index out of range");

}

}

**Comparison Operators (==, !=, <, >, <=, >=)**

Were overloaded to compare MyString objects lexicographically, allowing to check if one MyString is equal to, not equal to, less than, greater than, less than or equal to, or greater than or equal to another MyString.

bool MyString::*operator*>(*const* MyString& ob) *const*

{

*return* strcmp(str\_, ob.str\_) > 0;

}

bool MyString::*operator*<(*const* MyString& ob) *const*

{

*return* strcmp(str\_, ob.str\_) < 0;

}

bool MyString::*operator*>=(*const* MyString& ob) *const*

{

*return* !(\**this* < ob);

}

bool MyString::*operator*<=(*const* MyString& ob) *const*

{

*return* !(\**this* > ob);

}

bool MyString::*operator*==(*const* MyString& ob) *const*

{

*return* strcmp(str\_, ob.str\_) == 0;

}

bool MyString::*operator*!=(*const* MyString& ob) *const*

{

*return* !(\**this* == ob);

}

**Stream Operators (<<, >>)**

The << operator was overloaded to enable the printing of MyString objects to an output stream. The >> operator allows reading a line from an input stream into a MyString object.

std::ostream & *operator*<<(std::ostream &s, *const* MyString &ob)

{

*if* (ob.str\_)

s << ob.str\_;

*else*

s << "{null}";

*return* s;

}

std::istream & *operator*>>(std::istream& s, MyString& ob)

{

*const* int bufferSize = 1024;

char\* buffer = *new* char[bufferSize];

*if* (s.getline(buffer, bufferSize)) {

ob = buffer;

}

*return* s;

}

2.4 INSERT OPERATIONS

The implementation of insert-methods involves shifting characters within the string to make room for the inserted data and, if necessary, expanding the string's memory capacity to accommodate the new data.

**void MyString::insert(size\_t index, size\_t count, char ch)**

This method allows you to insert a specified number of characters (given by count) with a specific character ch at the specified index within the string.

void MyString::**insert**(size\_t index, size\_t count, char ch)

{

*if* (index > size\_ || count == 0) *throw* std::out\_of\_range("index out of range");

MyString inserted(count, ch);

insert(index, inserted);

}

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

This method lets you insert a null-terminated character array str at the specified index in the string.

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

{

*if* (index > size\_) *throw* std::out\_of\_range("index out of range");

*else* *if* (!str)*throw* std::out\_of\_range("str == nullptr");

MyString inserted(str);

insert(index, inserted);

}

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

This method allows you to insert a specified number of characters (count) from a null-terminated character array str at the specified index in the string.

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

{

*if* (index > size\_) *throw* std::out\_of\_range("index out of range");

*else* *if* (!str)*throw* std::out\_of\_range("str == nullptr");

MyString inserted(str, count);

insert(index, inserted);

}

**void MyString::insert(size\_t index, const std::string& arr)**

This method enables you to insert the contents of a std::string object arr at the specified index in the string.

void MyString::**insert**(size\_t index, *const* std::string& arr)

{

*if* (index > size\_) *throw* std::out\_of\_range("index out of range");

MyString inserted(arr);

insert(index, inserted);

}

**void MyString::insert(size\_t index, const std::string& arr, size\_t count)**

Similar to the previous method, this one allows you to insert a specified number of characters (count) from a std::string object arr at the specified index in the string.

void MyString::**insert**(size\_t index, *const* std::string& arr, size\_t count)

{

*if* (index > size\_) *throw* std::out\_of\_range("index out of range");

MyString inserted(arr.substr(0, count));

insert(index, inserted);

}

**void MyString::insert(size\_t index, const MyString& inserted)**

Inserts the content of another MyString object (inserted) into the current MyString object at a specified index. It handles memory management by reallocating memory if necessary and shifting characters to accommodate the insertion.

void MyString::**insert**(size\_t index, *const* MyString& inserted)

{

*if* (index > size\_) *throw* std::out\_of\_range("index out of range");

size\_t new\_size = size\_ + inserted.size();

*//* *Need* *to* *increase* *the* *capacity*

*if* (new\_size >= capacity\_)

{

size\_t new\_capacity = new\_size + 1;

char\* new\_str = *new* char[new\_capacity];

*//* *Copy* *the* *data* *to* *the* *insertion* *position*

strncpy(*new\_str*, str\_, index);

*//* *Copy* *the* *inserted* *line*

strncpy(*new\_str* *+* *index*, inserted.c\_str(), inserted.size());

*//* *Copy* *the* *rest* *of* *the* *data* *after* *insertion*

strncpy(*new\_str* *+* *index* *+* *inserted.size()*, str\_ + index, size\_ - index);

new\_str[new\_capacity - 1] = '\0';

*//* *Release* *the* *current* *memory* *and* *update*

clear();

size\_ = new\_size;

capacity\_ = new\_capacity;

str\_ = new\_str;

}

*else*

{

*//* *Shift* *the* *characters* *to* *the* *right* *to* *make* *room* *for* *insertion*

*for* (size\_t i = size\_; i >= index; --i)

{

str\_[i + inserted.size()] = str\_[i];

}

*//* *Inserting* *a* *line* *into* *the* *current* *line*

*for* (size\_t i = 0; i < inserted.size(); ++i)

{

str\_[index + i] = inserted[i];

}

size\_ = new\_size;

}

}

Inside each version of the insert method, a MyString object is created, which is then inserted into the current row. This use of constructors of the myString class makes the code more readable and reduces the complexity of the code inside the insert method.

2.5 APPEND OPERATIONS

The implementation of the append-methods in the MyString class allows for adding characters, C-style strings, or parts of C-style strings to the end of a MyString object.

void MyString::**append**(size\_t count, char ch)

{

\**this* = \**this* + MyString(count, ch);

}

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

{

\**this* = \**this* + MyString(str);

}

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

{

\**this* = \**this* + MyString(str + index, count);

}

void MyString::**append**(*const* std::string& arr)

{

\**this* = \**this* + MyString(arr);

}

void MyString::**append**(*const* std::string& arr, size\_t index, size\_t count)

{

\**this* = \**this* + MyString(arr.substr(index, count));

}

All versions of the append method actually delegate their work to the operator+ for string concatenation. This provides a single and clean way to add data to the current line and reduces code duplication.

2.6 REPLACE OPERATIONS

The replace-methods are designed to replace a portion of the string with a new substring or character sequence.

**void replace(size\_t index, size\_t count, const char\* str)**

This method replaces count characters starting from the index position in the string with the character sequence specified by the str parameter. It ensures that the resulting string maintains its capacity and size.

void MyString::**replace**(size\_t index, size\_t count, *const* char\* str) *//* *replace* *a* *count* *of* *char* *at* *index* *by* *“string”*

{

*//* *MultiReplace(index,* *count,* *str);*

*if* (str && index < size\_)

{

count = std::min(count, size\_ - index); *//* *Limit* *the* *value* *of* *count* *so* *as* *not* *to* *go* *beyond* *the* *line*

size\_t str\_len = strlen(str);

size\_t new\_str\_len = size\_ - count + str\_len;

char\* new\_str = *new* char[new\_str\_len + 1];

*//* *Copy* *the* *part* *of* *the* *string* *to* *the* *index* *that* *needs* *to* *be* *replaced*

strncpy(*new\_str*, str\_, index);

new\_str[index] = '\0';

*//* *Copy* *a* *new* *line*

strncpy(*new\_str* *+* *index*, str, str\_len);

*//* *Copy* *the* *rest* *of* *the* *original* *line*

strncpy(*new\_str* *+* *index* *+* *str\_len*, str\_ + index + count, size\_ - index - count);

new\_str[new\_str\_len] = '\0';

*//* *Free* *the* *old* *memory* *and* *update* *the* *data*

clear();

size\_ = new\_str\_len;

capacity\_ = new\_str\_len + 1;

CreateCopyArray(new\_str, new\_str\_len);

*delete*[] new\_str;

}

}

**void replace(size\_t index, size\_t count, const std::string& arr)**

This overload of the replace method takes a std::string object arr and behaves similarly to the first version, replacing count characters starting from the index position with the characters from the arr string.

void MyString::**replace**(size\_t index, size\_t count, *const* std::string& arr) *//* *replace* *a* *count* *of* *char* *at* *index* *by* *std::string*

{

replace(index, count, arr.c\_str());

}

2.7 FIND OPERATIONS

The find-methods are implemented to search for substrings within the MyString object. These methods provide the ability to locate the position of a specific substring or character sequence within the string.

**size\_t find(const char\* substr) const**

This method searches for the first occurrence of the specified null-terminated character array substr within the MyString. It returns the position of the found substring if it exists, or zero if the substring is not found.

size\_t MyString::**find**(*const* char\* substr) *const*

{

*return* find(substr, 0);

}

**size\_t find(const char\* substr, size\_t index) const**

Similar to the previous method, this variant allows you to specify a starting index from which the search begins. It searches for the substring starting from the given index position.

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

{

*if* (substr == *nullptr* || index >= size\_)

{

*return* 0;

}

*const* char\* result = std::strstr(*str\_* *+* *index*, substr); *//* *Looking* *for* *a* *substring* *starting* *from* *a* *given* *position*

*if* (result == *nullptr*)

{

*return* 0;

}

*return* result - str\_; *//* *Calculate* *the* *index* *of* *the* *first* *character* *of* *the* *found* *substring*

}

**size\_t find(const std::string& arr) const**

This method performs a search for the first occurrence of the specified std::string arr within the MyString object. It returns the position of the found substring if it exists or zero if not.

size\_t MyString::**find**(*const* std::string& arr) *const*

{

*return* find(arr.c\_str(), 0);

}

**size\_t find(const std::string& arr, size\_t index) const**

Similar to the previous method, this variant lets you specify a starting index for the search. It searches for the substring starting from the given index position.

size\_t MyString::**find**(*const* std::string& arr, size\_t index) *const*

{

*return* find(arr.c\_str(), index);

}

2.8 OTHER OPERATIONS

**c\_str() Method**

The c\_str() method returns a pointer to a null-terminated character array representing the content of the MyString object.

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

{

return str\_;

}

**data() Method**

The data() method returns a pointer to the character data stored in the MyString object. Unlike c\_str(), it does not necessarily have to be null-terminated.

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

{

return str\_;

}

**size() Method (length() Method)**

The size() method (length()) returns the number of characters in the MyString object, excluding the null-terminator.

size\_t MyString::**size**() const // return the number of char elements in string

{

return size\_;

}

size\_t MyString:: **length**() const // same as size

{

return size\_;

}

**capacity() Method**

The capacity() method returns the current amount of memory allocated for the MyString object, which can be larger than its actual size. It provides insight into how much memory is reserved for future string operations, including appending characters. The capacity may be greater than or equal to the size.

size\_t MyString::**capacity**() const // return the current amount of allocated memory for array

{

return capacity\_;

}

**empty() Method**

The empty() method returns true if the MyString object is empty (contains no characters), and false otherwise.

bool MyString::**empty**() *const* *//* *true* *if* *string* *is* *empty*

{

*return* (!str\_);

}

**clear() Method**

The clear() method removes all characters from the MyString object, making it empty. It also deallocates any memory previously allocated for the string.

void MyString::**clear**() // remove all char element in string

{

if (str\_)

{

delete[] str\_;

str\_ = nullptr;

size\_ = 0;

capacity\_ = 0;

}

}

**shrink\_to\_fit() Method**

The shrink\_to\_fit() method reduces the capacity of the MyString object to match its size. It can be used to release excess memory that may have been allocated due to previous operations.

void MyString::**shrink\_to\_fit**() *//* *reduce* *the* *capacity* *to* *size*

{

*if* (capacity\_ - size\_ > 1)

{

size\_t tmp\_size\_ = size\_;

char\* tmp\_str\_ = *new* char[tmp\_size\_ + 1];

strcpy\_s(*tmp\_str\_*, capacity\_, str\_);

clear();

capacity\_ = tmp\_size\_ + 1;

CreateCopyArray(tmp\_str\_, capacity\_);

size\_ = tmp\_size\_;

*delete*[] tmp\_str\_;

}

}

**substr() Method**

The substr() method returns a new MyString object that contains a substring of the original string.

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

{

*if* (index >= size\_)

{

*return* MyString();

}

*return* MyString(str\_ + index); *//* *Returning* *a* *new* *MyString* *starting* *from* *the* *given* *position*

}

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

{

*if* (index >= size\_)

{

*return* MyString();

}

*//* *We* *limit* *the* *value* *of* *count* *so* *as* *not* *to* *go* *beyond* *the* *line*

count = std::min(count, size\_ - index);

*return* MyString(str\_ + index, count); *//* *Returning* *a* *new* *MyString* *starting* *from* *a* *given* *position* *of* *a* *given* *count*

}

**erase() Method**

The erase() method removes a specified number of characters from the MyString object, starting from a given index position. It takes two arguments: index, which specifies the starting position for erasing, and count, which specifies the number of characters to erase.

void MyString::**erase**(size\_t index, size\_t count) *//* *erase* *count* *of* *char* *at* *index* *position*

{

*if* (index < size\_)

{

size\_t chars\_to\_erase = std::min(count, size\_ - index); *//* *Determine* *the* *number* *of* *characters* *that* *we* *will* *delete*

*for* (size\_t i = index; i < size\_ - chars\_to\_erase; ++i)

{

str\_[i] = str\_[i + chars\_to\_erase]; *//* *Shift* *the* *characters* *to* *the* *left* *by* *the* *number* *of* *characters* *to* *be* *deleted*

}

size\_ -= chars\_to\_erase;

str\_[size\_] = '\0';

}

}

**PrintString() Method**

The PrintString() method is a custom method that prints the content of the MyString object to the standard output (usually the console). It displays the string along with its size and capacity.

void MyString::**PrintString**()

{

*if* (str\_)

std::cout << str\_ << " (size = " << size() << ", capacity = " << capacity() << ")" << std::endl << std::endl;

*else*

std::cout << "{null}" << " (size = " << size() << ", capacity = " << capacity() << ")" << std::endl << std::endl;

}

2.9 UNIT TESTS

Test functions cover various aspects of the MyString class, including constructors, copy assignment, assignment with different types, concatenation, character and substring manipulation, and comparison operations.

**TestConstructors()**

This function tests various constructors of the MyString class, including default constructor, char array constructor, initializer list constructor, std::string constructor, and constructors with count parameters.

It creates instances of MyString using different constructor variations and prints the results.

**TestCopyAssignment()**

This function tests the copy assignment operator of the MyString class. It demonstrates various scenarios like assigning small strings to larger strings, equal strings, empty strings, and more.

**TestCharStringAssignment()**

This function tests assignments from C-style character arrays to MyString instances. It checks different scenarios, including small to large assignments, empty assignments, and more.

**TestStringAssignment()**

This function tests assignments from std::string objects to MyString instances. It covers various cases, including small to large assignments, empty assignments, and more.

**TestCharAssignment()**

This function tests single-character assignments to MyString instances. It demonstrates assigning characters to both empty and non-empty strings.

**TestConcatenate()**

This function tests the concatenation of MyString instances with other MyString instances, C-style character arrays, and std::string objects. It covers scenarios with empty strings, non-empty strings, and more.

**TestConcatenateWithAssignment()**

This function tests the += operator for concatenating strings with other MyString instances, C-style character arrays, and std::string objects. It explores different cases, including empty strings, full strings, and more.

**TestIndex()**

This function tests the index operator ([]) for accessing characters within MyString instances. It demonstrates accessing characters at various positions and handles both non-const and const MyString objects.

**TestLexicographicallyComparing()**

This function tests lexicographical comparison operations (==, !=, >, >=, <, <=) for MyString instances. It compares two MyString instances for equality and order and prints the results.

**TestAnotherOperators()**

This function tests various other member functions of the MyString class, including c\_str(), data(), size(), length(), capacity(), empty(), clear(), and shrink\_to\_fit(). It demonstrates these functions on different MyString instances.

**TestInputOperator()**

This function tests the input operator (>>) for reading input into a MyString instance from the console.

**TestInsert()**

This function tests the insert function of the MyString class, which allows inserting characters and substrings at specified positions. It covers various cases of inserting characters and substrings.

**TestErase()**

This function tests the erase function of the MyString class, which allows removing characters from a string. It explores different cases of erasing characters.

**TestAppend()**

This function tests the append function of the MyString class, which allows appending characters and substrings to a string. It covers scenarios involving characters, substrings, and std::string objects.

**TestReplace()**

This function tests the replace function of the MyString class, which allows replacing characters or substrings within a string. It demonstrates various cases of replacing characters and substrings.

**TestSubstr()**

This function tests the substr function of the MyString class, which allows extracting substrings from a string. It covers different scenarios of substring extraction.

**TestFind()**

This function tests the find function of the MyString class, which allows searching for substrings within a string. It checks for the presence of substrings and returns their positions.

Also, its own Unit Tests mechanism was prescribed. You can organize testing in such a way that the program checks how many tests have been conducted, how many have been passed, in order to optimize the work of the programmer who will change the code and check it for changes in its component parts.

2.10 EXCEPTIONS

Exceptions in programming are a way of handling unexpected or exceptional situations that can occur during the execution of a program.

**Throwing an Exception**

When an exceptional situation occurs, the program raises an exception. This is typically done using the throw keyword in languages like C++ or Java. The exception contains information about the nature of the problem, which is often represented as an object of an exception class.

**Catching an Exception**

Code that may potentially cause exceptions is placed inside a try block. Immediately following the try block, there can be one or more catch blocks. Each catch block specifies a particular type of exception it can handle. If an exception is thrown within the try block, the program looks for a matching catch block to handle it.

**Handling the Exception**

When an exception is thrown, the program searches for the appropriate catch block based on the type of exception thrown. If a matching catch block is found, the code inside that block is executed, allowing you to handle the exception. The error can be logged, display a message to the user, or take corrective actions. If no matching catch block is found within the current function, the function terminates abruptly, and the exception is propagated up the call stack to higher-level functions. This continues until a suitable catch block is found or until the program exits.

To provide support for basic std::exceptions, the <stdexcept> library was connected. substr(), replace(), erase(), insert() functions have been changed so that an out\_of\_range exception is thrown for incorrect values (Рисунок 1 – Пример исключения).

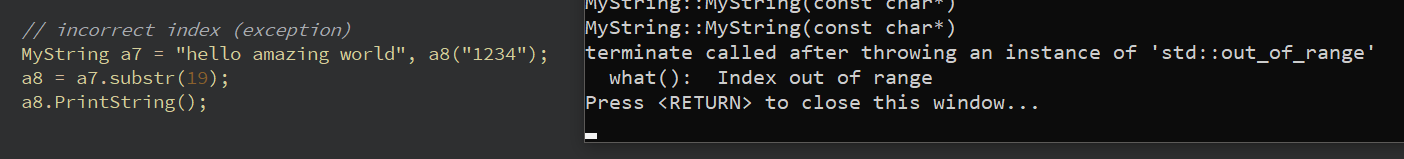


Рисунок 1 – Пример исключения

In order to create its own exception, a new class was created, which will be inherited from the standard std::exception class.

*class* **MyException** : *public* std::exception

{

*private*:

std::string errorMessage;

*public*:

**MyException**() = *delete*;

**MyException**(*const* std::string& message) : errorMessage(message) {}

*const* char\* ***what***() *const* *noexcept* *override* {*return* errorMessage.c\_str();}

};

Next, a situation was modeled in which some calling function throws an exception, and it is catched in the main function.

void TestException()

{

*try* {

[[maybe\_unused]] int c = SomeTestFoo();

}

*catch* (*const* std::exception& e) {

std::cerr << "Exception was catched: " << e.*what*() << std::endl;

}

}

int SomeTestFoo()

{

int a = 10, b = 0;

*if* (b == 0)

{

*throw* MyException("division by zero");

}

*else* *return* a/b;

}

2.11 MOVE CONSTRUCTOR, ASSIGNMENT

A move constructor in C++ is a special constructor used for efficiently transferring resources from one object to another without unnecessary data copying. Move Constructor transfers data from the source object to the target object and "empties" the source object. Essentially, it "steals" resources from the source object, saving resources on copying. After calling the move constructor, the source object remains in a state where its data is incorrect.

MyString str1("Hello");

MyString str2(std::move(str1)); // Here, data is moved from 'source' to 'target'

std::move is a standard library function in C++ used for explicit moving (or "transferring ownership") of an object from one place to another. It is part of the C++ Standard Library (STL) and is commonly used with move constructors and move assignment operators.

MyString::**MyString**(MyString&& ob) *//* *move* *constructor*

: str\_(ob.str\_), size\_(ob.size\_), capacity\_(ob.capacity\_)

{

std::cout << *\_\_PRETTY\_FUNCTION\_\_* << std::endl;

ob.size\_ = 0;

ob.capacity\_ = 0;

ob.str\_ = *nullptr*;

}

Move assignment, similar to move constructor, is an operation that efficiently transfers resources between objects of the same type while avoiding unnecessary data copying.

MyString & MyString::*operator*=(MyString&& ob) *//* *move* *assignment*

{

*//* *we* *must* *handle* *the* *case* *&ob* *==* *this* *(when* *an* *object* *assigns* *itself),* *otherwise* *in* *the*

*//* *function* *OperatorAssignment()* *memory* *will* *be* *cleared* *in* *str\_.*

*if* (&ob != *this*)

{

*//* *Releasing* *current* *resources*

clear();

*//* *Transferring* *resources* *from* *ob* *to* *the* *current* *object*

size\_ = ob.size\_;

capacity\_ = ob.capacity\_;

str\_ = ob.str\_;

*//* *Clearing* *ob*

ob.size\_ = 0;

ob.capacity\_ = 0;

ob.str\_ = *nullptr*;

}

*return* \**this*;

}

2.11 INTEGER, FLOAT CONSTRUCTORS

The MyString::MyString(unsigned int hex\_val) constructor dynamically determines the required size for representing an unsigned integer value in hexadecimal format using snprintf. It allocates memory accordingly, converts the integer to a hexadecimal string using snprintf, and stores it in the MyString object.

MyString::**MyString**(int hex\_val) // conversion constructor for integer types

{

std::cout << \_\_PRETTY\_FUNCTION\_\_ << std::endl;

int requiredSize = snprintf(nullptr, 0, "0x%X", hex\_val); // Determine the size of the string required to represent the number in hexadecimal form

capacity\_ = requiredSize + 1; // Allocate memory for the line

str\_ = new char[capacity\_];

snprintf(str\_, capacity\_, "0x%X", hex\_val); // Convert an integer to a string in hexadecimal form and store it in the myString object

size\_ = requiredSize;

}

The MyString::MyString(float floatValue) constructor dynamically determines the maximum number of characters required to represent a floating-point value using snprintf. It allocates memory accordingly, converts the float to a string using snprintf, and stores it in the MyString object.

MyString::**MyString**(float floatValue) *//* *conversion* *constructor* *for* *float* *types*

{

std::cout << *\_\_PRETTY\_FUNCTION\_\_* << std::endl;

int requiredSize = snprintf(*nullptr*, 0, "%f", floatValue);

capacity\_ = requiredSize + 1;

str\_ = *new* char[capacity\_];

snprintf(*str\_*, capacity\_, "%f", floatValue);

size\_ = requiredSize;

}

2.12 MORE MEMBER-FUNCTIONS

**Operator<< (std::basic\_ofstream)**

This operator overloads the output stream operator (<<) for writing a MyString object to a file using an std::basic\_ofstream (file output stream).

std::basic\_ofstream<char>& *operator*<<(std::basic\_ofstream<char>& file, *const* MyString& str)

{

*if* (str.size\_ > 0 && str.str\_)

{

file.write(str.str\_, str.size\_);

}

*return* file;

}

**Operator>> (std::basic\_ifstream)**

This operator overloads the input stream operator (>>) for reading a MyString object from a file using an std::basic\_ifstream (file input stream).

std::basic\_ifstream<char>& *operator*>>(std::basic\_ifstream<char>& file, MyString& str) *//* *Operator* *for* *reading* *a* *myString* *object* *from* *a* *file*

{

*//* *Determine* *the* *maximum* *size* *of* *the* *buffer* *to* *read*

*const* size\_t maxBufferSize = 4096;

char buffer[maxBufferSize];

str.clear();

*//* *Read* *the* *data* *from* *the* *file* *and* *add* *it* *to* *the* *myString* *string*

*while* (file)

{

file.read(*buffer*, maxBufferSize);

size\_t bytesRead = *static\_cast*<size\_t>(file.gcount());

*if* (bytesRead > 0)

{

str.append(buffer);

}

}

*return* file;

}

**at(size\_t index)**

The function is used to access the character at a specified index within the string. It takes an index as its argument and returns the character at that position.

char MyString::**at**(size\_t index) const // return the index element of string if exist, otherwise throw an exception

{

if (index >= size\_)

{

throw std::out\_of\_range("Index out of range");

}

return str\_[index];

}

**to\_int()**

The to\_int function in the MyString class converts the string representation of an integer, stored within the MyString object, into an actual integer value.

long long int MyString::**to\_int**() // perform conversion from string to integer types: char, short,int an other

{

for (size\_t i = 0; i < size\_; ++i)

{

if (!isdigit(str\_[i]))

throw std::out\_of\_range("!isdigit(str\_[i])");

}

return std::stoi(str\_);

}

**to\_float()**

float MyString::**to\_float**() *const*

{

*//* *Pointer* *to* *the* *character* *that* *ends* *the* *parsing*

char\* endptr;

*//* *Use* *strtof* *to* *parse* *the* *string* *as* *a* *floating-point* *number*

float result = strtof(str\_, *&endptr*);

*//* *Check* *for* *parsing* *errors*

*if* (\*endptr != '\0')

{

*throw* std::invalid\_argument("Invalid float conversion");

}

*return* result;

}

2.13 AHO-CORASICK

The Aho-Corasick string matching algorithm, which is used to efficiently find multiple keywords in a given text.

**BohrNode** is a class representing a node of the Bohr (trie) structure used in the Aho-Corasick algorithm for finding keywords in text. Bohr nodes contain children, a reference to the failure state, a flag indicating whether the node marks the end of a keyword, and a field to store the length of the keyword.

**AhoCorasick** is a class representing the Aho-Corasick algorithm itself. It has methods for adding keywords, building the structure of the Bohr trie, and finding keywords in text.

**AddKeyword** is a method for adding a keyword to the Bohr trie. It creates nodes for each letter of the keyword, sets the is\_end\_of\_pattern\_ flag for the last letter of the keyword, and stores the length of the keyword.

**Build** is a method for building suffix links (failure links) in the Bohr. These links allow quick traversal between nodes in the Bohr in case of failed comparisons.

**FindKeywords** is a method for finding keywords in text. It uses the Bohr trie to search for all occurrences of keywords in the text and returns the positions and lengths of the found keywords.

**Transition** is a method for transitioning between nodes in the Bohr trie based on the characters in the text. It is used during the keyword search process.

**MyString::find** is a method that allows invoking keyword search in text by passing an AhoCorasick object.

std::vector<std::pair<size\_t, size\_t>> MyString::**find**(*const* AhoCorasick& ahoCorasick) *const*

{

*return* ahoCorasick.FindKeywords(\**this*);

}

AhoCorasick::**AhoCorasick**() {

root\_ = *new* BohrNode();

}

*//* *Method* *for* *adding* *a* *keyword* *to* *the* *Bohr*

void AhoCorasick::**AddKeyword**(*const* MyString& keyword)

{

BohrNode\* node = root\_;

*//* *Loop* *through* *the* *characters* *of* *the* *keyword* *and* *create* *corresponding* *nodes* *in* *the* *Bohr*

*for* (size\_t i = 0; i < keyword.size(); ++i)

{

char ch = keyword[i];

*//* *If* *there* *is* *no* *node* *with* *the* *current* *character,* *create* *one*

*if* (node->children\_.find(ch) == node->children\_.end())

{

node->children\_[ch] = *new* BohrNode();

}

*//* *Move* *to* *the* *next* *node*

node = node->children\_[ch];

}

*//* *Set* *the* *flag* *indicating* *the* *end* *of* *the* *keyword* *and* *store* *its* *length*

node->is\_end\_of\_pattern\_ = *true*;

node->keyword\_length\_ = keyword.size(); *//* *Set* *the* *length* *of* *the* *keyword*

}

*//* *Building* *the* *suffix* *links* *in* *the* *Bohr*

void AhoCorasick::**Build**()

{

std::queue<BohrNode\*> q;

*//* *Initialize* *the* *failure* *link* *of* *the* *root* *node* *to* *itself* *for* *its* *children*

*for* (*auto*& kv : root\_->children\_)

{

BohrNode\* child = kv.second;

child->fail\_ = root\_;

q.push(child);

}

*while* (!q.empty())

{

BohrNode\* node = q.front();

q.pop();

*//* *Process* *the* *children* *of* *the* *current* *node* *to* *set* *the* *failure* *links*

*for* (*auto*& kv : node->children\_)

{

char ch = kv.first;

BohrNode\* child = kv.second;

q.push(child);

*//* *Search* *for* *the* *failure* *link* *for* *the* *current* *node*

BohrNode\* failNode = node->fail\_;

*while* (failNode != *nullptr* && failNode->children\_.find(ch) == failNode->children\_.end())

{

failNode = failNode->fail\_;

}

*//* *Set* *the* *failure* *link* *for* *the* *current* *node*

*if* (failNode == *nullptr*)

{

child->fail\_ = root\_;

}

*else*

{

child->fail\_ = failNode->children\_[ch];

}

}

}

}

*//* *Finding* *keywords* *in* *the* *text*

std::vector<std::pair<size\_t, size\_t>> AhoCorasick::**FindKeywords**(*const* MyString& text) *const*

{

BohrNode\* cur\_state = root\_;

std::vector<std::pair<size\_t, size\_t>> matches;

size\_t textLength = text.size();

*for* (size\_t i = 0; i < textLength; ++i)

{

char currentChar = text[i];

cur\_state = Transition(*cur\_state*, currentChar);

*//* *Process* *the* *is\_end\_of\_pattern\_* *flag* *for* *the* *current* *node*

BohrNode\* tempState = cur\_state;

*while* (tempState != root\_)

{

*if* (tempState->is\_end\_of\_pattern\_)

{

*//* *Found* *a* *keyword* *at* *the* *current* *position*

*//* *You* *can* *save* *or* *use* *its* *position* *here*

size\_t keywordPosition = i - tempState->keyword\_length\_ + 1;

matches.push\_back(std::make\_pair(*keywordPosition*, *cur\_state->keyword\_length\_*));

}

tempState = tempState->fail\_;

}

}

*return* matches;

}

*//* *Transitioning* *between* *Bohr* *nodes* *based* *on* *text* *characters*

BohrNode\* AhoCorasick::**Transition**(BohrNode\* state, char character) *const*

{

*while* (state != *nullptr* && state->children\_.find(character) == state->children\_.end())

{

state = state->fail\_;

}

*if* (state == *nullptr*)

{

*return* root\_;

}

*return* state->children\_[character];

}

2.14 ITERATORS

In C++, iterators play a key role when working with containers such as strings, the myString class being developed, and other data structures. Iterators provide a convenient way to access container elements sequentially, which allows you to efficiently perform various operations, such as iterating, searching, changing and deleting elements.

The myString class being developed includes a variety of methods for working with strings. An interesting addition to the class being developed would be iterators that would allow the user to easily iterate over the characters of the string being developed or perform other manipulations with its contents. This gives the user flexibility in using the class being developed and simplifies interaction with its contents.

The use of iterators in C++ is especially important when working with standard containers, such as std::vector, std::list, and others. However, creating custom iterators for custom classes, such as the myString being developed, can also significantly improve the usability and readability of the code.

In this paper, the iterators iterator, const\_iterator, reverse\_iterator, const\_reverse\_iterator were written and some methods for the myString class were overloaded.

2.15 PYTHON WRAPPING

In C++, there are several ways to wrap functions for use from Python.

Boost.Python: Boost.Python is a library from the Boost library designed to integrate C++ with Python. It provides a high-level interface for creating wrappers of C++ functions and classes for use from Python.

pybind11: pybind11 is a lightweight and flexible library for creating C++ wrappers for Python without the need for a Boost compiler. It provides a simple syntax and good support for data transfer between C++ and Python.

Wrapper using extern "C" \_\_declspec (dllexport): This approach is used to create a library that can be easily used from other languages, including Python. The extern "C" keywords ensure that the C++ compiler uses a function calling convention compatible with the C language. \_\_declspec (dllexport) is used to export functions from DLL to Windows.

First I chose pybind11. We need to configure the environment so that we can work with Python. At first, there was an attempt to connect Python to the Qt project and connect the pybind11 library, but nothing good came of it, I could not fix the error in which the Qt studio did not see the Python.h file.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок – Процесс настройки проекта

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок – Процесс настройки проекта

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 4 – Процесс настройки проекта

Then an attempt was made to create a library in C++ using extern "C" \_\_declspec (dllexport) and connect it to Python.

Изображение выглядит как текст, программное обеспечение, снимок экрана, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок – Процесс создания dll: изменение свойств проекта, специфика написания функций на C++

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок – Процесс создания dll: h-файл

Then the resulting dll was copied to an empty folder and this folder was opened using Visual Code.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 7 – Использование dll

Methods of the MyString class were wrapped in a similar way.

As you can see, I tested the ability to export C-functions from the dll. But in order to fully and conveniently use the library, you can register a class at the Python level.

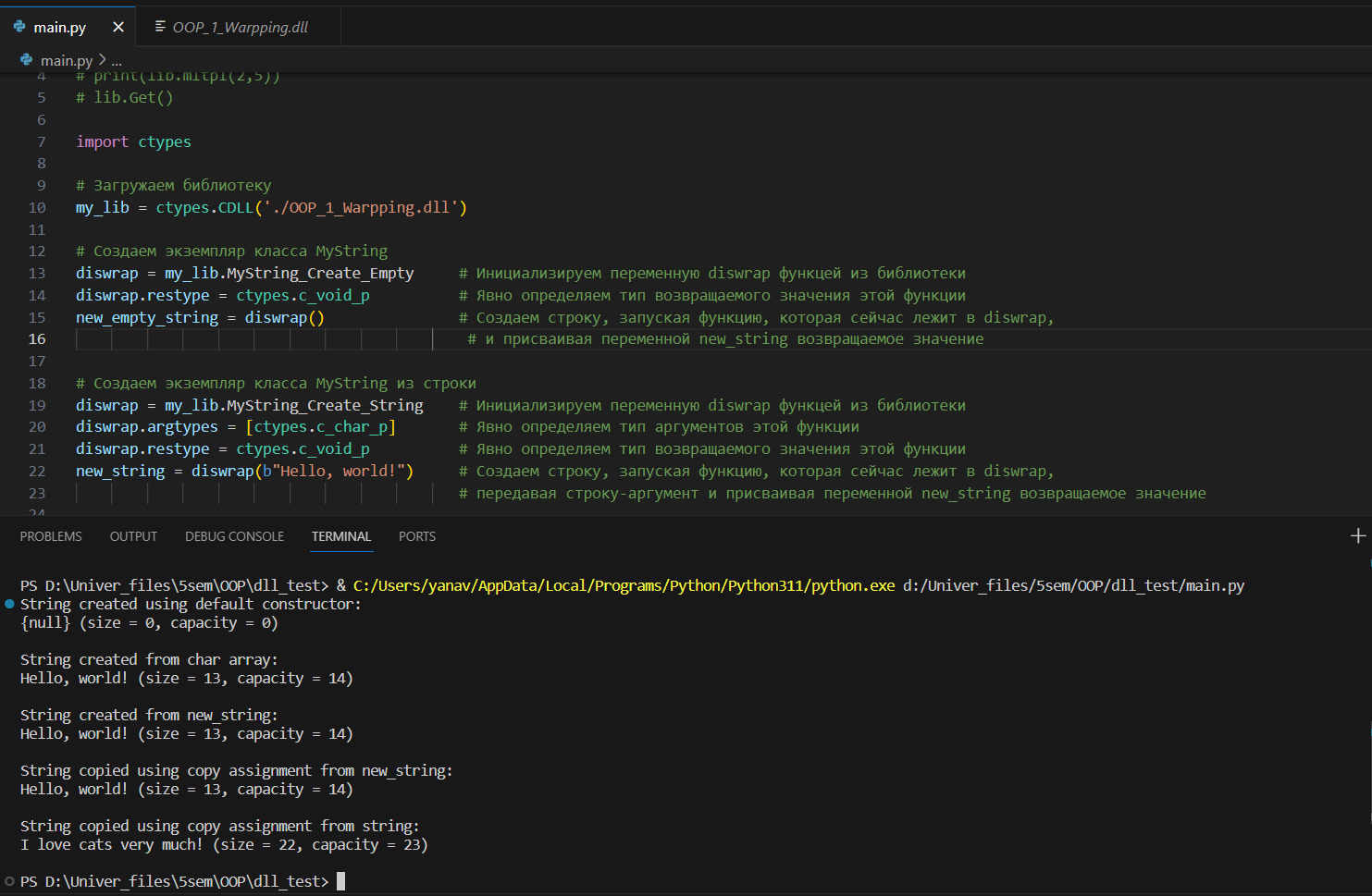


Рисунок – Тестирование функционала библиотеки

3 CONCLUSION

During the development of the MyString class, a powerful string library capable of working with various data types and providing various string operations has been implemented. Class provides a convenient interface for creating, modifying, and processing strings. In the course of completing this project, a deeper understanding of various fundamental concepts in programming and software development were gained.

4 REFERENCES

1. Шилдт, Герберт. С++: базовый курс. : Пер. с англ. – Спб. : ООО “Диалектика”, 2019. – 624 стр. : ил. – Парал. тит. англ.
2. Алгоритм Ахо-Корасик [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/articles/198682/> , свободный.

APPENDIX A

Header file «MyString.h»

#ifndef MYSTRING\_H

#define MYSTRING\_H

#include <iostream>

#include <cstring>

#include <initializer\_list>

#include <cstring>

#include <stdexcept>

#include <fstream>

#include <unordered\_map>

#include <vector>

#include <queue>

*class* **AhoCorasick**;

*class* **MyString**

{

*private*:

char\* str\_ = *nullptr*;

size\_t size\_ = 0;

size\_t capacity\_ = 0;

*//* *The* *functions* *below* *are* *internal* *(auxiliary)* *for* *operators.*

*//* *I* *put* *them* *in* *private* *so* *that* *only* *member-function* *can* *use* *them*

void **CreateCopyArray**(*const* char\* arr, size\_t str\_size); *//* *for* *constructors*

void **OperatorAssignment**(*const* char\* arr, size\_t arr\_size); *//* *for* *operator=*

void **OperatorConcatenate**(MyString& ob, *const* char\* arr, size\_t str\_size); *//* *for* *operator+*

*public*:

**MyString**();

**MyString**(*const* char\* arr);

**MyString**(std::initializer\_list<char> il);

**MyString**(*const* std::string& arr);

**MyString**(*const* char\* arr, size\_t count);

**MyString**(size\_t count, char ch);

**MyString**(*const* MyString& arr);

**MyString**(MyString&& ob);

**MyString**(int val);

**MyString**(float floatValue);

MyString & *operator*=(*const* MyString& ob);

MyString & *operator*=(*const* char\* arr);

MyString & *operator*=(*const* std::string& arr);

MyString & *operator*=(*const* char ch);

MyString & *operator*=(MyString&& ob);

MyString *operator*+(*const* MyString& ob);

MyString *operator*+(*const* char\* arr);

MyString *operator*+(*const* std::string& arr);

MyString & *operator*+=(*const* char\* arr);

MyString & *operator*+=(*const* std::string& arr);

char & *operator*[](size\_t index) *const*;

*const* char & *operator*[](size\_t index);

bool *operator*>(*const* MyString& other) *const*;

bool *operator*<(*const* MyString& other) *const*;

bool *operator*>=(*const* MyString& other) *const*;

bool *operator*<=(*const* MyString& other) *const*;

bool *operator*!=(*const* MyString& other) *const*;

bool *operator*==(*const* MyString& other) *const*;

*const* char\* **c\_str**() *const*;

*const* char\* **data**() *const*;

size\_t **size**() *const*;

size\_t **length**() *const*;

size\_t **capacity**() *const*;

bool **empty**() *const*;

void **clear**();

void **shrink\_to\_fit**();

char **at**(size\_t index) *const*;

long long int **to\_int**() *const*;

float **to\_float**() *const*;

*friend* std::ostream & *operator*<<(std::ostream& s, *const* MyString& ob);

*friend* std::istream & *operator*>>(std::istream& s, MyString& ob);

*friend* std::basic\_ofstream<char>& *operator*<<(std::basic\_ofstream<char>& file, *const* MyString& str);

*friend* std::basic\_ifstream<char>& *operator*>>(std::basic\_ifstream<char>& file, MyString& str);

void **insert**(size\_t index, size\_t count, char ch);

void **insert**(size\_t index, *const* char\* str);

void **insert**(size\_t index, *const* char\* str, size\_t count);

void **insert**(size\_t index, *const* std::string& arr);

void **insert**(size\_t index, *const* std::string& arr, size\_t count);

void **insert**(size\_t index, *const* MyString& inserted);

void **erase**(size\_t index, size\_t count);

void **append**(size\_t count, char ch);

void **append**(*const* char\* str);

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

void **append**(*const* std::string& arr);

void **append**(*const* std::string& arr, size\_t index, size\_t count);

void **replace**(size\_t index, size\_t count, *const* char\* new\_str);

void **replace**(size\_t index, size\_t count, *const* std::string& arr);

MyString **substr**(size\_t index) *const*;

MyString **substr**(size\_t index, size\_t count) *const*;

size\_t **find**(*const* char\* substr) *const*;

size\_t **find**(*const* char\* substr, size\_t index) *const*;

size\_t **find**(*const* std::string& arr) *const*;

size\_t **find**(*const* std::string& arr, size\_t index) *const*;

std::vector<std::pair<size\_t, size\_t>> **find**(*const* AhoCorasick& ahoCorasick) *const*;

void **PrintString**();

~**MyString**();

};

*class* **BohrNode** {

*public*:

std::unordered\_map<char, BohrNode\*> children\_;

BohrNode\* fail\_;

bool is\_end\_of\_pattern\_;

size\_t keyword\_length\_; *//* *Добавляем* *поле* *для* *хранения* *длины* *ключевого* *слова*

**BohrNode**() : fail\_(*nullptr*), is\_end\_of\_pattern\_(*false*), keyword\_length\_(0) {}

};

*class* **AhoCorasick** {

*public*:

**AhoCorasick**();

void **AddKeyword**(*const* MyString& keyword);

void **Build**();

std::vector<std::pair<size\_t, size\_t>> **FindKeywords**(*const* MyString& text) *const*;

*private*:

BohrNode\* root\_;

BohrNode\* **Transition**(BohrNode\* state, char character) *const*;

};

#endif *//* *MYSTRING\_H*