1. Ministry of Science and Education of Russian Federation
2. Peter the Great St.Petersburg Polytechnic University
3. —
4. Institute of cubersecurity and information security

**LABORATORY WORK № 1**

1. **«Class MyString»**
2. course «OOP»
3. Student
4. Gr. 4851003/80802 Bahir M. N.

<*signature*>

1. Instructor

Chernov A.Y

<*signature*>

1. Saint-Petersburg
2. 2020

**Table of contents**

[Tasks 3](#_Toc63270755)

[Theory and results 4](#_Toc63270756)

[Base functional 5](#_Toc63270757)

[**Constructors** 5](#_Toc63270758)

[**Operators** 8](#_Toc63270759)

[**Overloaded methods** 12](#_Toc63270760)

[**Other methods** 17](#_Toc63270761)

[**Testing** 18](#_Toc63270762)

[Conclusion 21](#_Toc63270763)

[APPENDIX 1 22](#_Toc63270764)

[APPENDIX 2 29](#_Toc63270765)

[APPENDIX 3 33](#_Toc63270766)

[APPENDIX 4 42](#_Toc63270767)

# **Tasks**

In this work, you need to implement your class for representing a character string in C ++, thus getting acquainted with object-oriented programming, implementing your methods, testing them, as well as making the code as universal as possible and minimizing it by using methods in a bundle, which will help to avoid duplication and complicating the code, and also minimizing the risk of errors.

The use of containers and STL algorithms is not allowed.

It is necessary to extend the functionality of the Python interpreter with an implemented class.

As an additional task, it is necessary to develop a base template class of the dynamic array DynArray (analogue of vector). Memory allocation for elements should be done in a single unit, not in blocks

For each method, a test suite for the built-in data type should be developed.

The String class must inherit from DynArray and have two forms BasicString (for single-byte strings) and WideString (for multibyte strings). Converting BasicString to WideString through the constructor and assignment operator, respectively, must be supported.

# **Theory and results**

In this work you need to implement your class that represents a string in C ++. This class and its methods make it as easy as possible for the programmer to work with arrays of characters or strings, as the methods implemented provide the developer with a number of functions that are most useful when implementing his own code or project. The programmer does not have to oversee the allocation and cleaning of the memory, since everything is already in the classroom. Among other things, there are a sufficient number of methods to potentially change the string that allow you to use various algorithms to change the string if a specific code requires it.

In the C language, for example, there is a library <string.h>, which can also offer the programmer some functions for working with character arrays. The problem, however, is that some functions in this library can be unsafe and lead to errors. They lead to code that doesn't work, but in skilled hands they can have more serious consequences. For example, using the strcpy () function in the absence of any validation mechanisms could cause the buffer to overflow and overwrite data on the code stack that an attacker could use to execute arbitrary code

# **Base functional**

We will provide descriptions of the developed methods in the classes:

## **Constructors**

|  |  |
| --- | --- |
| **Declaration** | **Description** |
| BasicString(); | Default constructor for empty string. |
| BasicString(const char \*str); | A constructor for creating an object of a class that allocates memory for a string and fixes its size in the appropriate field. |
| BasicString(const WideString& str); | Allows you to convert from WideString to String |
| BasicString(std::string str); | Constructor to work with std::string. |
| BasicString(const char\* str, size\_t count); | Allows you to create a line that consists only of characters “str”, “сount” size. |
| BasicString(const BasicString &str); | Copy constructor. |
| BasicString(BasicString&& str); | Move constructor |
| BasicString(size\_t count, char c); | Allows you to create a line that consists only character, “сount” size. |
| BasicString(unsigned int num); | Convert num to hex str |
| ~BasicString(); | Destructor to destroy an object. |

*Table 1 – constructors BasicString*

|  |  |
| --- | --- |
| **Declaration** | **Description** |
| WideString(); | Default constructor for empty string. |
| WideString(const char \*str); | A constructor for creating an object of a class that allocates memory for a string and fixes its size in the appropriate field. |
| WideString(const BasicString& str); | Allows you to convert from String to WideString |
| WideString(const wchar\_t\* str, size\_t count); | Allows you to create a line that consists only of characters “str”, “сount” size. |
| WideString(const WideString &str); | Copy constructor. |
| WideString(BasicString&& str); | Move constructor |
| WideString(size\_t count, wchar\_t c); | Allows you to create a line that consists only character, “сount” size. |
| WideString(unsigned int num); | Convert num to hex str |
| ~WideString(); | Destructor to destroy an object. |

*Table 2 – constructors WideString*

|  |  |
| --- | --- |
| **Declaration** | **Description** |
| explicit dynarray(const Allocator& alloc = Allocator()) | Default Constructor |
| dynarray(size\_type count, const T& value, const Allocator& alloc = Allocator()) | Allows you to create DynArray with value and size count. |
| explicit dynarray(size\_type count) | Allows you to create DynArray with size count. |
| dynarray(inputiterator first, inputiterator last, const Allocator& alloc = Allocator()) | Constructor that uses iterators |
| dynarray(dynarray&& other, const Allocator& alloc) | Move Constructor |
| dynarray(const dynarray& other, const Allocator& alloc) | Copy Constructor |
| ~dynarray() | Deconstructor |

*Table 3 – constructors DynArray*

With all of this you can create class objects. The constructors allocate the required amount of memory themselves, and the destructor removes it if necessary. All of this allows you to create a objects of different elements and types, which increases the versatility of use and saves the programmer from work the new[] and delete[] operators.

## **Operators**

|  |  |
| --- | --- |
| **Declaration** | **Description** |
| BasicString& operator =(const BasicString &other); | Copy Assignment |
| BasicString& operator =(const char \*str); | Assignment, works with a pointer to an object const char. |
| BasicString& operator =(char c); | Assignment, allows you to assign a character to a class string field. |
| BasicString & operator=(const WideString& str); | Assignments working with WideString. |
| BasicString& operator=(std::string str); | Assignments working with std::string. |
| BasicString& operator= (BasicString&& str); | Move Assignment |
| BasicString operator+ (const BasicString& left, const BasicString& right); | Operator "+", which allows you to concatenate two lines into one. |
| BasicString operator+ (const BasicString& left, const char\* right); | The addition operator that allows you to concatenate a string of a class object and сonst char \*string. |
| BasicString operator+ (const BasicString& left, const std::string& right); | The addition operator that allows you to concatenate a string of a class object and std::string. |
| BasicString& operator+= (const char\* str); | Combined assignment and plus operator, which assigns the current line to the class, the string that is combined with the object const char \*string. |
| BasicString operator +=(std::string str); | Combined assignment and plus operator, which assigns the current line to the class, the string that is combined with the object std::string. |
| BasicString& operator+= (const BasicString& str); | Combined assignment and plus operator, which assigns the current line to the class, the string that is combined with the object BasicString. |
| bool operator== (const BasicString& str) const; | This operator allows you to check string equality. |
| bool operator!= (const BasicString& str) const; | This operator allows you to check string inequality. |
| bool operator> (const BasicString& str) const; | This operator allows you to compare strings to see which of them is greater. |
| bool operator< (const BasicString& str) const; | This operator allows you to compare strings for which one is smaller |
| bool operator>= (const BasicString& str) const; | This operator allows you to compare strings to see which of them is greater or equal. |
| bool operator<= (const BasicString& str) const; | This operator allows you to compare strings for which one is smaller or equal |
| friend std::ostream& operator<< (std::ostream& os, BasicString& str); | This operator allows you to output a string. |
| friend std::istream& operator >> (std::istream& is, BasicString& str); | This operator allows you to read a string. |

*Table 4 – operators BasicString*

|  |  |
| --- | --- |
| **Declaration** | **Description** |
| WideString& operator =(const WideString &str); | Copy Assignment |
| WideString& operator =(const wchar\_t \*str); | Assignment, works with a pointer to an object const char. |
| WideString & operator =(wchar\_t c); | Assignment, allows you to assign a character to a class string field. |
| WideString & operator=(const BasicString& str); | Assignments working with BasicString. |
| WideString & operator= (WideString && str); | Move Assignment |
| WideString operator+ (const WideString & left, const WideString & right); | Operator "+", which allows you to concatenate two lines into one. |
| WideString operator+ (const WideString & left, const wchar\_t \* right); | The addition operator that allows you to concatenate a string of a class object and сonst char \*string. |
| WideString operator+ (const WideString & left, const std::wstring& right); | The addition operator that allows you to concatenate a string of a class object and std::string. |
| WideString & operator+= (const wchar\_t \* str); | Combined assignment and plus operator, which assigns the current line to the class, the string that is combined with the object const char \*string. |
| WideString & operator+= (const WideString & str); | Combined assignment and plus operator, which assigns the current line to the class, the string that is combined with the object WideString. |
| bool operator== (const WideString & str) const; | This operator allows you to check string equality. |
| bool operator!= (const WideString & str) const; | This operator allows you to check string inequality. |
| bool operator> (const WideString & str) const; | This operator allows you to compare strings to see which of them is greater. |
| bool operator< (const WideString & str) const; | This operator allows you to compare strings for which one is smaller |
| bool operator>= (const WideString & str) const; | This operator allows you to compare strings to see which of them is greater or equal. |
| bool operator<= (const WideString & str) const; | This operator allows you to compare strings for which one is smaller or equal |
| friend std::ostream& operator<< (std::ostream& os, WideString & str); | This operator allows you to output a string. |
| friend std::istream& operator >> (std::istream& is, WideString & str); | This operator allows you to read a string. |

*Table 5 – operators WideString*

|  |  |
| --- | --- |
| **Declaration** | **Description** |
| dynarray& operator= (const dynarray& other) | Copy Assignment |
| dynarray& operator=(dynarray&& other) | Move Assignment |
| bool operator== (dynarray<T, Alloc>& lhs, dynarray<T, Alloc>& rhs) | This operator allows you to check dynarray equality. |
| bool operator!= (dynarray<T, Alloc>& lhs, dynarray<T, Alloc>& rhs) | This operator allows you to check string inequality. |
| reference operator[] (size\_type pos) | This operator allows you to return element by pos. |

*Table 6 – operators DynArray*

These operators allow you to perform a fairly large number of operations: assignment, glueing, comparison, input and output, and so on. At the same time, the user also does not need to monitor memory, and also does not use functions such as

These operators allow you to perform a relatively large number of operations: allocation, concatenation, comparison, input and output. At the same time, the user does not need to monitor memory or use functions such as strcmp(), strcpy(), strcat()

## **Overloaded methods**

|  |  |
| --- | --- |
| **Declaration** | **Description** |
| void insert(Index i, size\_t count, char c); | Allows you to add a certain number of characters starting from a certain index. |
| void insert(Index i, const char\* mid\_str); | Allows you to add a row starting at a specific index |
| void insert(Index i, char\* mid\_str, size\_t count); | Allows you to add a specific number of characters from a string starting at a specific index. |
| void insert(Index i, const std::string& str); | Allows you to add a row starting at a specific index, with an object like std::string. |
| void insert(Index i, const std::string& str, size\_t count); | Allows you to add a specific number of characters from a string starting at a specific index, with an object like std::string. |
| void insert(const\_iterator pos, inputiterator first, inputiterator last); | With iterators |
| void erase(Index i, size\_t count); | Removes a specified number of characters from the given index. |
| void erase(size\_t count, iterator i); | Same with iterators |
| void append(size\_t count, char c); | Add a specific amount of a given character to the end of a string. |
| void append(const char\* str); | Adds a string to the end of a string. |
| void append(const std::string& str); | Adds a string to the end of a string, with an object like std::string. |
| void append(const char\* str, Index i, size\_t count); | Add a specific number of characters from a string, starting at a given index. |
| void append(const std::string& str, Index i, size\_t count); | Add a specific number of characters from a string, starting at a given index, with an object like std::string. |
| void replace(Index i, size\_t count, const char\* str); | Writes a specified number of characters from the given one, starting the line starting at a specified character. |
| void replace(Index i, size\_t count, const std::string& str); | Writes a specified number of characters from the given one, starting the line starting at a specified character, with an object like std::string. |
| BasicString substr(Index i, size\_t count) const; | Returns a substring of a string starting at the given index, limited by the given size. |
| BasicString substr(Index i) const; | Returns a substring of a string starting at a specific index. |
| Index find(const char\* substr, Index i\_) const; | Searches for a substring in a string starting at a specific index. |
| Index find(const char\* substr) const; | Searches for a substring in a string |
| Index find(const std::string& substr) const; | Searches for a substring in a string, with an object like std::string. |
| Index find(const std::string& substr, Index i\_) const; | Searches for a substring in a string starting at a specific index, with an object like std::string. |

*Table 7 – overloaded methods BasicString*

|  |  |
| --- | --- |
| **Declaration** | **Description** |
| void insert(Index i, size\_t count, wchar\_t c); | Allows you to add a certain number of characters starting from a certain index. |
| void insert(Index i, const wchar \* mid\_str); | Allows you to add a row starting at a specific index |
| void insert(Index i, wchar \* mid\_str, size\_t count); | Allows you to add a specific number of characters from a string starting at a specific index. |
| void insert(Index i, const std::string& str); | Allows you to add a row starting at a specific index, with an object like std::string. |
| void insert(Index i, const std::string& str, size\_t count); | Allows you to add a specific number of characters from a string starting at a specific index, with an object like std::string. |
| void insert(const\_iterator pos, inputiterator first, inputiterator last); | With iterators |
| void erase(Index i, size\_t count); | Removes a specified number of characters from the given index. |
| void erase(size\_t count, iterator i); | Same with iterators |
| void append(size\_t count, wchar c); | Add a specific amount of a given character to the end of a string. |
| void append(const wchar \* str); | Adds a string to the end of a string. |
| void append(const std::string& str); | Adds a string to the end of a string, with an object like std::string. |
| void append(const char\* str, Index i, size\_t count); | Add a specific number of characters from a string, starting at a given index. |
| void append(const std::string& str, Index i, size\_t count); | Add a specific number of characters from a string, starting at a given index, with an object like std::string. |
| void replace(Index i, size\_t count, const wchar \* str); | Writes a specified number of characters from the given one, starting the line starting at a specified character. |
| void replace(Index i, size\_t count, const std::string& str); | Writes a specified number of characters from the given one, starting the line starting at a specified character, with an object like std::string. |
| WideString substr(Index i, size\_t count) const; | Returns a substring of a string starting at the given index, limited by the given size. |
| WideString substr(Index i) const; | Returns a substring of a string starting at a specific index. |
| Index find(const char\* substr, Index i\_) const; | Searches for a substring in a string starting at a specific index. |
| Index find(const char\* substr) const; | Searches for a substring in a string |

*Table 8 – overloaded methods WideString*

|  |  |
| --- | --- |
| **Declaration** | **Description** |
| void assign(size\_type count, const T& value) | Assigns new values to the elements of the vector, replacing the old ones. |
| void assign(inputiterator first, inputiterator last) | Same with iterators |
| iterator insert(const\_iterator pos, const T& value) | Inserts new elements before the element at the specified position(copy assignment) |
| iterator insert(const\_iterator pos, T&& value) | Inserts new elements before the element at the specified position(move assignment) |
| iterator insert(const\_iterator pos, size\_type count, const T& value) | Inserts new elements before the element and size at the specified position |
| void insert(const\_iterator pos, inputiterator first, inputiterator last) | Assigns new values to the elements of the vector, replacing the old ones with iterators |
| iterator erase(const\_iterator pos) | Erase element on pos |
| iterator erase(const\_iterator first, const\_iterator last) | Same with iterators |
| void push\_back(const T& value) | Add a new element to the vector(copy assignment) |
| void push\_back(T&& value) | Add a new element to the vector(move assignment) |
| void pop\_back() | Delete a last element to the vector |
| void resize(size\_type count) | Resizes the container so that it contains n elements. |
| void resize(size\_type count, const value\_type& value) | Resizes the container so that it contains n elements with value |
| void shrink\_to\_fit() | Resize capacity to size |
| void swap(dynarray& other) | Swaps one vector with other vector |

*Table 9 – overloaded methods DynArray*

## **Other methods**

|  |  |
| --- | --- |
| size\_t size() const; | Getter for private field size. |
| size\_t capacity() const; | Getter for private field capacity. |
| void clear(); | Fills in str with \0. |
| char \* shrink\_to\_fit(); | Set the capacity to size. |
| char \*c\_str() const; | Getter for private field str. |
| char\* data(); | Return new array of c\_str data. |
| bool empty() const; | Return true if size 0. |

*Table 10 – other methods BasicString*

|  |  |
| --- | --- |
| size\_t size() const; | Getter for private field size. |
| size\_t capacity() const; | Getter for private field capacity. |
| void clear(); | Fills in str with \0. |
| Wchar\_t \* shrink\_to\_fit(); | Set the capacity to size. |
| Wchar\_t \*c\_str() const; | Getter for private field str. |
| Wchar\_t \* data(); | Return new array of c\_str data. |
| bool empty() const; | Return true if size 0. |

*Table 11 – other methods WideString*

|  |  |
| --- | --- |
| size\_type size() const | Getter for private field size. |
| size\_type capacity() const | Getter for private field capac. |
| void clear(); | Fills vector with 0. |
| void shrink\_to\_fit() | Set the capacity to size. |
| bool empty() const | Return true if size 0. |

*Table 12 – other methods DynArray*

With these methods the user can quickly and easily get information about the object he is working with, the amount of allocated memory, the amount of space already used, is the object data empty or not? It is also possible to change the size of the allocated memory, which means that the user can no longer work with functions: strcmp(), strlen() and operators new[], delete[].

Methods were also implemented using iterators.

## **Testing**

At this point we will test the developed methods, the results are shown:

|  |  |  |
| --- | --- | --- |
| **Name** | **Input** | **Result** |
| size\_t size() const; | **BasicString a (“123”)** | **3** |
| size\_t capacity() const; | **BasicString a( “123”)** | **16** |
| char \* shrink\_to\_fit(); | **BasicString a( “123”) =>Size 3, cap = 16** | **Size 3, cap = 3** |
| void insert(Index i, size\_t count, char c); | BasicString a(” 1111”),  i=3,count =1,c=’o’ | 11o1 |
| void erase(Index i, size\_t count); | BasicString a() 1111111111”),  i=0, count =10 | “” |
| void append(const char\* str); | BasicString a(”1”),  str=”12” | 112 |
| void replace(Index i, size\_t count, const char\* str); | BasicString a(”123”),  i=0, count=2, str=”33” | 333 |
| BasicString substr(Index i, size\_t count) const; | BasicString a(”123”),  i=1, count=1 | 2 |
| Index find(const char\* substr) const; | BasicString a(”123”),  Substr=“2” | 2 |
| BasicString& operator =(const BasicString &other); | BasicString a(”123”);  BasicString b(”124”);  a=b; | 124 |
| BasicString & operator=(const WideString& str); | BasicString a(”123”);  WideString b(L”124”);  a=b; | a=”124” |
| BasicString operator+ (const BasicString& left, const char\* right); | BasicString a(“123”);  Const char b[2]={‘2’,’3’} | 12323 |
| bool operator== (const BasicString& str) const; | BasicString a(“123”);  BasicString b(“123”); | true |
| bool operator> (const BasicString& str) const; | BasicString a=”123”  BasicString b=”1234” | false |
| bool operator< (const BasicString& str) const; | BasicString a=”123”  BasicString b=”1234” | true |

*Table 13 – testing BasicString*

|  |  |  |
| --- | --- | --- |
| **Name** | **Input** | **Result** |
| size\_t size() const; | **WideString a (“123”)** | **3** |
| size\_t capacity() const; | **WideString a( “123”)** | **8** |
| char \* shrink\_to\_fit(); | **WideString a( “123”) =>Size 3, cap = 8** | **Size 3, cap = 3** |
| void insert(Index i, size\_t count, char c); | **WideString** a(” 1111”),  i=3,count =1,c=’o’ | 11o1 |
| void erase(Index i, size\_t count); | **WideString** a() 1111111111”),  i=0, count =10 | “” |
| void append(const wchar\_t\* str); | **WideString** a(”1”),  Const wchar\_t str[2]={L’1’,L’2’} | 112 |
| void replace(Index i, size\_t count, const wchar \* str); | **WideString** a(”123”),  i=0, count=2, str=”33” | 333 |
| **WideString** substr(Index i, size\_t count) const; | **WideString** a(”123”),  i=1, count=1 | 2 |
| Index find(const wchar \* substr) const; | **WideString** a(”123”),  Substr=L“2” | 2 |
| **WideString** & operator =(const **WideString** &other); | **WideString** a(”123”);  **WideString** b(L”124”);  a=b; | 124 |
| **WideString** & operator=(const BasicString& str); | **WideString** a(”123”);  BasicString b(L”124”);  a=b; | a=”124” |
| **WideString** operator+ (const **WideString** & left, const char\* right); | **WideString** a(“123”);  Const char b[2]={‘2’,’3’} | 12323 |
| bool operator== (const **WideString** & str) const; | **WideString** a(“123”);  **WideString** b(“123”); | true |
| bool operator> (const **WideString** & str) const; | **WideString** a=”123”  **WideString** b=”1234” | false |
| bool operator< (const **WideString** & str) const; | **WideString** a=”123”  **WideString** b=”1234” | true |

*Table 14 – testing WideString*

|  |  |  |
| --- | --- | --- |
| **Name** | **Input** | **Result** |
| size\_type size() const | dynarray<int> f(10,5); | **5** |
| size\_type capacity() const | dynarray<int> f(10,5); | **10** |
| void clear(); |  |  |
| void shrink\_to\_fit() | f.assign(5, 10);  f.push\_back(5)-> size= 6,capacity=12 | **Size=6, capacity = 6** |
| void assign(size\_type count, const T& value) | dynarray<int> f(10,5);  f.assign(5,10) | F[0]=10 |
| iterator insert(const\_iterator pos, const T& value) | dynarray<int> f(10,5);  dynarray<int>::iterator pos = f.begin();  f.insert(pos,1,2); | F[0]=2 |
| iterator erase(const\_iterator pos) | dynarray<int> f(5,10);  dynarray<int>::iterator pos = f.begin();  f.insert(pos,1,2);  pos = f.begin();  f.insert(pos,1,2); | F[0]=10 |
| void swap(dynarray& other) | dynarray<int> ff(10);  dynarray<int> f(5,10);  f[0]=10;  ff[0]=0;  f.swap(ff) | f[0]=0;  ff[0]=10; |
| dynarray& operator= (const dynarray& other) | dynarray<int> ff(10);  dynarray<int> f(5,10);  f[0]=10  f=ff | f[0]=0 |
| bool operator== (dynarray<T, Alloc>& lhs, dynarray<T, Alloc>& rhs) | dynarray<int> ff(10);  dynarray<int> f(5,10);  f==ff | false |
| reference operator[] (size\_type pos) | dynarray<int> f(5,10);  f[1] | 10 |

*Table 15 – testing DynArray*

# **Conclusion**

As part of the laboratory work, knowledge of the basics of object-oriented programming was gained. The BasicString, WideString, DynArray class was developed.

APPENDIX 1

DynArray class:

Dynarray.h

#define \_CRT\_SECURE\_NO\_WARNINGS

#pragma once

#include <iostream>

#include <stdexcept>

#include <utility>

#include <iterator>

#include <memory>

#include <cstddef> // for size\_t

#include <initializer\_list>

//using std::rel\_ops::operator!=;

//Class DynArray

template <class T, class Allocator = std::allocator<T>>

class dynarray

{

public:

//Member type def

typedef T value\_type;

typedef size\_t size\_type;

typedef T\* iterator;

typedef const T\* const\_iterator;

typedef std::reverse\_iterator<iterator> reverse\_iterator;

typedef std::reverse\_iterator<const\_iterator> const\_reverse\_iterator;

typedef Allocator allocator\_type;

typedef ptrdiff\_t difference\_type;

typedef T\* pointer;

typedef const T\* const\_pointer;

typedef T& reference;

typedef const T& const\_reference;

//Constructors, Destructors, Copy, Move

explicit dynarray(const Allocator& alloc = Allocator()) :

\_size(0), \_capacity(\_size),

\_alloc(alloc), \_array(\_alloc.allocate(\_capacity))

{}

dynarray(size\_type count,

const T& value,

const Allocator& alloc = Allocator()) :

\_size(count), \_capacity(\_size),

\_alloc(alloc), \_array(\_alloc.allocate(\_capacity))

{

std::uninitialized\_fill\_n(begin(), \_size, value);

}

explicit dynarray(size\_type count) :

\_size(count), \_capacity(\_size),

\_alloc(), \_array(\_alloc.allocate(\_capacity))

{

std::uninitialized\_fill\_n(begin(), \_size, T());

}

template <class inputIterator>

dynarray(inputIterator first, inputIterator last,

const Allocator& alloc = Allocator(),

typename std::enable\_if<!std::is\_integral

<inputIterator>::value>::type\* = nullptr) :

\_size(std::distance(first, last)), \_capacity(\_size),

\_alloc(alloc), \_array(\_alloc.allocate(\_capacity))

{

std::uninitialized\_copy(first, last, begin());

}

dynarray(std::initializer\_list<T> init,

const Allocator& alloc = Allocator()) :

dynarray(init.begin(), init.end(), alloc)

{}

dynarray(const dynarray& other) :

\_size(other.\_size), \_capacity(\_size),

\_alloc(other.\_alloc), \_array(\_alloc.allocate(\_capacity))

{

std::uninitialized\_copy(other.begin(), other.end(), begin());

}

dynarray(const dynarray& other, const Allocator& alloc) :

\_size(other.\_size), \_capacity(\_size),

\_alloc(alloc), \_array(\_alloc.allocate(\_capacity))

{

std::uninitialized\_copy(other.begin(), other.end(), begin());

}

dynarray(dynarray&& other) :

\_size(std::move(other.\_size)), \_capacity(std::move(other.\_capacity)),

\_alloc(std::move(other.\_alloc)), \_array(std::move(other.\_array))

{

other.\_size = 0;

other.\_capacity = 0;

other.\_array = nullptr;

}

dynarray(dynarray&& other, const Allocator& alloc) :

\_size(std::move(other.\_size)), \_capacity(std::move(other.\_capacity)),

\_alloc(alloc), \_array(std::move(other.\_array))

{

other.\_size = 0;

other.\_capacity = 0;

other.\_array = nullptr;

}

~dynarray()

{

clear();

if (\_array && \_capacity)

\_alloc.deallocate(\_array, \_capacity);

\_size = 0;

\_capacity = 0;

\_array = nullptr;

}

allocator\_type get\_allocator() const

{

return \_alloc;

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

//Operator=

dynarray& operator= (const dynarray& other)

{

if (this != &other)

dynarray(other).swap(\*this);

return \*this;

}

dynarray& operator= (dynarray&& other)

{

if (this != &other)

{

\_size = std::move(other.\_size);

\_capacity = std::move(other.\_capacity);

\_alloc = std::move(other.\_alloc);

\_array = std::move(other.\_array);

other.\_size = 0;

other.\_capacity = 0;

other.\_array = nullptr;

}

return \*this;

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

//Element access

reference at(size\_type pos)

{

if (pos >= \_size)

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

return \_array[pos];

}

const\_reference at(size\_type pos) const

{

if (pos >= \_size)

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

return \_array[pos];

}

reference operator[] (size\_type pos)

{

return \_array[pos];

}

const\_reference operator[] (size\_type pos) const

{

return \_array[pos];

}

reference front()

{

return \*begin();

}

const\_reference front() const

{

return \*begin();

}

reference back()

{

return \*(end() - 1);

}

const\_reference back() const

{

return \*(end() - 1);

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

//Capacity

bool empty() const

{

return \_size == 0;

}

size\_type size() const

{

return \_size;

}

size\_type capacity() const

{

return \_capacity;

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

//Modifiers

void assign(size\_type count, const T& value) //assign (ref)

{

dynarray(count, value).swap(\*this);

}

template <class inputiterator>

void assign(inputiterator first, inputiterator last)

{

dynarray(first, last).swap(\*this);

}

void clear()

{

for (iterator i = end(); i != begin(); )

\_alloc.destroy(--i);

\_size = 0;

}

size\_type max\_size() const

{

return size\_type(-1) / sizeof(T);

}

void reserve(size\_type size)

{

if (capacity() < size)

{

size\_type n = \_size;

pointer newelements = \_alloc.allocate(size);

std::uninitialized\_copy(begin(), end(), newelements);

clear();

if (\_array && \_capacity)

\_alloc.deallocate(\_array, \_capacity);

\_array = newelements;

\_size = n;

\_capacity = size;

}

}

iterator insert(const\_iterator pos, const T& value)

{

return insert(pos, 1, value);

}

iterator insert(const\_iterator pos, T&& value)

{

size\_type n = std::distance(cbegin(), pos);

if (size() + 1 > capacity())

reserve(2 \* (size() + 1));

std::copy\_backward(begin() + n, end(), end() + 1);

++\_size;

back() = std::move(value);

return begin() + n;

}

iterator insert(const\_iterator pos, size\_type count, const T& value)

{

size\_type n = std::distance(cbegin(), pos);

if (size() + count > capacity())

reserve(2 \* (size() + count));

std::copy\_backward(begin() + n, end(), end() + count);

std::fill(begin() + n, begin() + n + count, value);

\_size += count;

return begin() + n;

}

template <class inputIterator>

typename std::enable\_if<!std::is\_integral<inputIterator>::value, iterator>::

type insert(const\_iterator pos, inputIterator first, inputIterator last)

{

size\_type n = std::distance(cbegin(), pos);

size\_type count = std::distance(first, last);

if (size() + count > capacity())

reserve(2 \* (size() + count));

std::copy\_backward(begin() + n, end(), end() + count);

std::copy(first, last, begin() + n);

m\_size += count;

return begin() + n;

}

iterator erase(const\_iterator pos)

{

size\_type beg = std::distance(cbegin(), pos);

if (pos + 1 != cend())

std::copy(begin() + beg + 1, end(), begin() + beg);

--\_size;

//\_alloc.destroy(end());

return begin() + beg;

}

iterator erase(const\_iterator first, const\_iterator last)

{

size\_type size = std::distance(first, last);

size\_type beg = std::distance(cbegin(), first);

iterator loc = std::copy(begin() + beg + size, end(), begin() + beg);

for (iterator i = end(); i != loc; )

//\_alloc.destroy(--i);

return begin() + beg;

}

void push\_back(const T& value)

{

insert(end(), value);

}

void push\_back(T&& value)

{

insert(end(),1, std::move(value));

}

void pop\_back()

{

if (!empty())

{

--\_size;

\_alloc.destroy(end());

}

}

void resize(size\_type count)

{

resize(count, T());

}

void resize(size\_type count, const value\_type& value)

{

if (count < size())

erase(begin() + count, end());

else if (count > size())

insert(end(), count - size(), value);

}

void shrink\_to\_fit()

{

if (\_array && size() != capacity())

{

\_capacity = \_size;

}

}

void swap(dynarray& other) //swap (member function)

{

if (this != &other)

{

std::swap(\_size, other.\_size);

std::swap(\_capacity, other.\_capacity);

std::swap(\_alloc, other.\_alloc);

std::swap(\_array, other.\_array);

}

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

//iterators

iterator begin()

{

return \_array;

}

const\_iterator begin() const

{

return \_array;

}

const\_iterator cbegin() const

{

return \_array;

}

iterator end()

{

return \_array + \_size;

}

const\_iterator end() const

{

return \_array + \_size;

}

const\_iterator cend() const

{

return \_array + \_size;

}

reverse\_iterator rbegin()

{

return reverse\_iterator(end());

}

const\_reverse\_iterator rbegin() const

{

return const\_reverse\_iterator(end());

}

const\_reverse\_iterator crbegin() const

{

return const\_reverse\_iterator(end());

}

reverse\_iterator rend()

{

return reverse\_iterator(begin());

}

const\_reverse\_iterator rend() const

{

return const\_reverse\_iterator(begin());

}

const\_reverse\_iterator crend() const

{

return const\_reverse\_iterator(begin());

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//

private:

size\_type \_size;

size\_type \_capacity;

allocator\_type \_alloc;

pointer \_array;

template <class Integer>

void \_initialize(size\_type count, const Integer& value, std::true\_type)

{

\_size = count;

\_capacity = 2 \* \_size;

\_array = \_alloc.allocate(\_capacity);

std::uninitialized\_fill\_n(begin(), \_size, value);

}

template <class inputiterator>

void \_initialize(inputiterator first, inputiterator last, std::false\_type)

{

distance(first, last, \_size);

\_capacity = 2 \* \_size;

\_array = \_alloc.allocate(\_capacity);

std::uninitialized\_copy(first, last, begin());

}

};

template<class T, class Alloc>

bool operator== (dynarray<T, Alloc>& lhs, dynarray<T, Alloc>& rhs) //operator ==(!=) (non-member function)

{

return (lhs.size() == rhs.size() && std::equal(lhs.begin(), lhs.end(), rhs.begin()));

}

template<class T, class Alloc>

bool operator!= (dynarray<T, Alloc>& lhs, dynarray<T, Alloc>& rhs) //operator ==(!=) (non-member function)

{

return !(lhs.size() == rhs.size() && std::equal(lhs.begin(), lhs.end(), rhs.begin()));

}

template <class T, class Alloc>

void swap(dynarray<T, Alloc> &lhs, dynarray<T, Alloc> &rhs) //swap (non member function)

{

lhs.swap(rhs);

}

APPENDIX 2

BasicString class:

BasicString.h

#pragma once

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "dynarray.h"

#include <iostream>

#include <fstream>

#include <string>

#include <cstring>

#include <cmath>

#include <comdef.h> //for wchar to char

#include <wchar.h>

#include <iterator>

class BasicString;

class WideString;

class BasicString : public dynarray<char> {

public:

typedef unsigned short int Index;

typedef char\* iterator;

typedef const char\* const\_iterator;

typedef std::reverse\_iterator<iterator> reverse\_iterator;

typedef std::reverse\_iterator<const\_iterator> const\_reverse\_iterator;

BasicString();

BasicString(const BasicString& str);

BasicString(const WideString& str);

BasicString(BasicString&& str);

BasicString(const char\* str);

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

BasicString(const std::string& str);

BasicString(const char\* str, size\_t count);

BasicString(size\_t count, char c);

BasicString(unsigned int num);

~BasicString();

char\* c\_str() const;

BasicString& operator= (const char\* str);

BasicString& operator= (const std::string& str);

BasicString& operator= (const BasicString& str); // copy assigment

BasicString& operator= (BasicString&& str); // move assigment

BasicString& operator= (char c);

BasicString& operator= (const WideString& str);

BasicString& operator+= (const char\* str);

BasicString& operator+= (const std::string& str);

BasicString& operator+= (const BasicString& str);

bool operator== (const BasicString& str) const;

bool operator!= (const BasicString& str) const;

bool operator> (const BasicString& str) const;

bool operator< (const BasicString& str) const;

bool operator>= (const BasicString& str) const;

bool operator<= (const BasicString& str) const;

friend std::ostream& operator<< (std::ostream& os, BasicString& str);

friend std::istream& operator >> (std::istream& is, BasicString& str);

void clear();

void shrink\_to\_fit();

char operator[] (Index i) const;

char\* data();

size\_t size() const;

size\_t capacity() const;

bool empty() const;

template <class inputiterator>

void insert(const\_iterator pos, inputiterator first, inputiterator last);

void insert(Index i, const char\* mid\_str);

void insert(Index i, size\_t count, char c);

void insert(Index i, char\* mid\_str, size\_t count);

void insert(Index i, const std::string& str);

void insert(Index i, const std::string& str, size\_t count);

void insert(size\_t count, iterator i, char c,int value);

void insert(const std::string& str, iterator i, int value);

void append(size\_t count, char c);

void append(const char\* str);

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

void append(const std::string& str);

void append(const std::string& str, Index i, size\_t count);

Index find(const char\* substr) const;

Index find(const char\* substr, Index i\_) const;

Index find(const std::string& substr) const;

Index find(const std::string& substr, Index i\_) const;

void replace(Index i, size\_t count, const char\* str);

void replace(Index i, size\_t count, const std::string& str);

void replace(size\_t count, iterator i, char\* str);

void replace(size\_t count, iterator i, const std::string& str);

BasicString substr(Index i, size\_t count) const;

BasicString substr(Index i) const;

void erase(Index i, size\_t count);

void erase(size\_t count, iterator i);

int to\_int() const;

char at(int index) const;

iterator begin() const;

iterator end() const;

const\_iterator cbegin();

const\_iterator cend();

reverse\_iterator rbegin();

reverse\_iterator rend();

const\_reverse\_iterator rbegin() const;

const\_reverse\_iterator rend() const;

const\_reverse\_iterator crbegin() const;

const\_reverse\_iterator crend() const;

private:

void change\_capacity(size\_t new\_capacity);

size\_t round\_size(size\_t size);

char\* c\_str\_;

char\* data\_;

size\_t size\_, capacity\_;

static const int default\_size=16;

};

BasicString operator+ (const BasicString& left, const char\* right);

BasicString operator+ (const BasicString& left, const BasicString& right);

BasicString operator+ (const BasicString& left, const std::string& right);

std::ostream& operator<< (std::ostream& os, BasicString& str);

std::istream& operator >> (std::istream& is, BasicString& str);

***//WideString***

class WideString : dynarray<wchar\_t> {

public:

typedef unsigned short int Index;

typedef wchar\_t\* iterator;

typedef const wchar\_t\* const\_iterator;

typedef std::reverse\_iterator<iterator> reverse\_iterator;

typedef std::reverse\_iterator<const\_iterator> const\_reverse\_iterator;

WideString();

WideString(const BasicString& str);

WideString(const WideString& str);

WideString(WideString&& str) ;

WideString(const wchar\_t\* str);

WideString(std::initializer\_list<wchar\_t>& il);

WideString(const wchar\_t\* str, size\_t count);

WideString(size\_t count, wchar\_t c);

WideString(unsigned int num);

~WideString();

wchar\_t\* c\_str() const;

WideString& operator= (const wchar\_t\* str);

WideString& operator= (const WideString& str); // copy assigment

WideString& operator= (WideString&& str); // move assigment

WideString& operator= (wchar\_t c);

WideString& WideString::operator= (const BasicString& str);

WideString& operator+= (const wchar\_t\* str);

WideString& operator+= (const std::string& str);

WideString& operator+= (const WideString& str);

bool operator== (const WideString& str) const;

bool operator!= (const WideString& str) const;

bool operator> (const WideString& str) const;

bool operator< (const WideString& str) const;

bool operator>= (const WideString& str) const;

bool operator<= (const WideString& str) const;

friend std::ostream& operator<< (std::ostream& os, WideString& str);

friend std::istream& operator >> (std::istream& is, WideString& str);

void clear();

void shrink\_to\_fit();

wchar\_t operator[] (Index i) const;

wchar\_t\* data();

size\_t size() const;

size\_t capacity() const;

bool empty() const;

void insert(Index i, const wchar\_t\* mid\_str);

void insert(Index i, size\_t count, wchar\_t c);

void insert(Index i, wchar\_t\* mid\_str, size\_t count);

void insert(Index i, const std::string& str);

void insert(Index i, const std::string& str, size\_t count);

void insert(size\_t count, iterator i, wchar\_t c,int value);

void insert(const std::string& str, iterator i,int value);

void append(size\_t count, wchar\_t c);

void append(const wchar\_t\* str);

void append(const wchar\_t\* str, Index i, size\_t count);

void append(const std::string& str);

void append(const std::string& str, Index i, size\_t count);

Index find(const wchar\_t\* substr) const;

Index find(const wchar\_t\* substr, Index i\_) const;

Index find(const std::string& substr) const;

Index find(const std::string& substr, Index i\_) const;

void replace(Index i, size\_t count, const wchar\_t\* str);

void replace(Index i, size\_t count, const std::string& str);

void replace(size\_t count, iterator i, wchar\_t\* str);

WideString substr(Index i, size\_t count) const;

WideString substr(Index i) const;

void erase(Index i, size\_t count);

void erase(size\_t count, iterator i);

int to\_int() const;

wchar\_t at(int index) const;

template <class inputiterator>

void insert(const\_iterator pos, inputiterator first, inputiterator last);

iterator begin() const;

iterator end() const;

const\_iterator cbegin();

const\_iterator cend();

reverse\_iterator rbegin();

reverse\_iterator rend();

const\_reverse\_iterator rbegin() const;

const\_reverse\_iterator rend() const;

const\_reverse\_iterator crbegin() const;

const\_reverse\_iterator crend() const;

private:

void change\_capacity(size\_t new\_capacity);

size\_t round\_size(size\_t size);

wchar\_t\* c\_str\_;

wchar\_t\* data\_;

size\_t size\_, capacity\_;

static const int default\_size=8;

};

WideString operator+ (const WideString& left, const wchar\_t\* right);

WideString operator+ (const WideString& left, const WideString& right);

WideString operator+ (const WideString& left, const std::wstring& right);

std::ostream& operator<< (std::ostream& os, WideString& str);

std::istream& operator >> (std::istream& is, WideString& str);

APPENDIX 3

*BasicString.cpp*

#include "BasicString.h"

BasicString::BasicString()

: size\_(0),

capacity\_(default\_size),

c\_str\_(new char[default\_size]),

data\_(nullptr)

{

c\_str\_[0] = '\0';

}

BasicString::BasicString(const BasicString& str) // copy constructor

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

capacity\_(str.capacity\_),

c\_str\_(new char[str.capacity\_]),

data\_(nullptr)

{

std::strcpy(this->c\_str\_, str.c\_str\_);

}

BasicString::BasicString(const WideString& str) // copy constructor

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

capacity\_(str.capacity()),

c\_str\_(new char[str.capacity()]),

data\_(nullptr)

{

strcpy(this->c\_str\_, \_bstr\_t(str.c\_str()));

}

BasicString::BasicString(BasicString&& str) : // move constructor

size\_(str.size()),

capacity\_(str.capacity()),

c\_str\_(new char[str.capacity()]),

data\_(nullptr)

{

\*this = std::move(str);

}

BasicString::BasicString(const char\* str)

{

size\_ = std::strlen(str);

capacity\_ = round\_size(size\_);

c\_str\_ = new char[capacity\_];

std::strcpy(this->c\_str\_, str);

data\_ = nullptr;

}

BasicString::BasicString(std::initializer\_list<char>& il)

: size\_(il.size()),

capacity\_(round\_size(il.size())),

c\_str\_(new char[round\_size(il.size())]),

data\_(nullptr)

{

std::strcpy(this->c\_str\_, il.begin());

}

BasicString::BasicString(const std::string& str)

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

capacity\_(str.capacity()),

c\_str\_(new char[str.capacity()]),

data\_(nullptr)

{

std::memcpy(this->c\_str\_, str.c\_str(), size\_);

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

}

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

{

size\_ = count;

capacity\_ = round\_size(size\_);

c\_str\_ = new char[capacity\_];

std::memcpy(this->c\_str\_, str, count);

data\_ = nullptr;

}

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

{

size\_ = count;

capacity\_ = round\_size(size\_);

c\_str\_ = new char[capacity\_];

std::memset(c\_str\_, c, count);

data\_ = nullptr;

}

BasicString::BasicString(unsigned int num)

{

char c;

auto byte\_size{ [](unsigned int num) -> int {

return 1 + static\_cast<int> (std::log2(num)) / 8;

} };

size\_ = byte\_size(num) \* 2 + 2; // 1 byte in hex = 2 symbols. + 2 symbols for '0', 'x'

capacity\_ = round\_size(size());

c\_str\_ = new char[capacity()];

c\_str\_[size()] = '\0';

int i = size() - 1;

while(i >= 2) {

c = (num & 0xf);

if (c <= 9) {

c\_str\_[i] = c + '0';

}

else {

c\_str\_[i] = c - 10 + 'a';

}

num >>= 4;

--i;

}

c\_str\_[0] = '0';

c\_str\_[1] = 'x';

data\_ = nullptr;

}

BasicString::~BasicString()

{

delete[] c\_str\_;

c\_str\_ = nullptr;

if (data\_ != nullptr) {

delete[] data\_;

data\_ = nullptr;

}

}

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

{

if (this->capacity\_ < std::strlen(str)) {

change\_capacity(round\_size(std::strlen(str)));

}

std::strcpy(this->c\_str\_, str);

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

return\*this;

}

BasicString& BasicString::operator= (const WideString& str)

{

if (this->capacity\_ < wcslen(str.c\_str())) {

change\_capacity(round\_size(wcslen(str.c\_str())));

}

strcpy(this->c\_str\_, \_bstr\_t(str.c\_str()));

this->size\_ = wcslen(str.c\_str());

return\*this;

}

BasicString& BasicString::operator= (const std::string& str)

{

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

}

BasicString& BasicString::operator= (const BasicString& str) // copy assigment

{

if (this != &str) {

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

}

else {

return\*this;

}

}

BasicString& BasicString::operator= (BasicString&& str) // move assigment

{

if (this != &str) {

delete[] this->c\_str\_;

this->c\_str\_ = str.c\_str\_;

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

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

str.c\_str\_ = nullptr;

str.size\_ = 0;

str.capacity\_ = 0;

}

return\*this;

}

BasicString& BasicString::operator= (char c)

{

this->c\_str\_[0] = c;

this->size\_ = 1;

return\*this;

}

BasicString& BasicString::operator+= (const char\* str)

{

if (this->capacity\_ <= this->size\_ + std::strlen(str)) {

change\_capacity(round\_size(this->size\_ + std::strlen(str)));

}

std::strcat(this->c\_str\_, str);

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

return\*this;

}

BasicString& BasicString::operator+= (const std::string& str)

{

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

}

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

{

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

}

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

{

if (this == &str) {

return true;

}

return (std::strcmp(this->c\_str\_, str.c\_str()) == 0);

}

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

{

return !(\*this == str);

}

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

{

return std::strcmp(this->c\_str(), str.c\_str\_) > 0;

}

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

{

return std::strcmp(this->c\_str(), str.c\_str\_) < 0;

}

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

{

return std::strcmp(this->c\_str(), str.c\_str\_) >= 0;

}

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

{

return std::strcmp(this->c\_str(), str.c\_str\_) <= 0;

}

void BasicString::clear()

{

\*this = BasicString();

}

void BasicString::shrink\_to\_fit()

{

change\_capacity(size() + 1);

}

char BasicString::operator[] (Index i) const

{

return c\_str\_[i];

}

size\_t BasicString::size() const

{

return size\_;

}

size\_t BasicString::capacity() const

{

return capacity\_;

}

bool BasicString::empty() const

{

return (size() == 0);

}

char\* BasicString::data()

{

if (data\_ != nullptr) {

delete[] data\_;

}

data\_ = new char[size()];

std::memcpy(data\_, c\_str\_, size());

return data\_;

}

void BasicString::insert(Index i\_, const char\* mid\_str)

{

if (i\_ > size()) {

throw "Out of range";

}

if (size\_ + std::strlen(mid\_str) >= capacity\_) {

change\_capacity(round\_size(size\_ + std::strlen(mid\_str) + 1));

}

for (int i = size\_ + std::strlen(mid\_str) - 1; i >= i\_ + std::strlen(mid\_str); --i) {

c\_str\_[i] = c\_str\_[i - std::strlen(mid\_str)];

}

for (int i = i\_, j = 0; i < i\_ + std::strlen(mid\_str); ++i, ++j) {

c\_str\_[i] = mid\_str[j];

}

size\_ += std::strlen(mid\_str);

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

}

void BasicString::insert(Index i, size\_t count, char c)

{

char\* mid\_str = new char[((count/16) + 1)\*16];

std::memset(mid\_str, c, count);

mid\_str[count] = '\0';

insert(i, mid\_str);

delete[] mid\_str;

}

void BasicString::insert(Index i, char\* mid\_str, size\_t count)

{

std::memcpy(mid\_str, static\_cast<const char\*>(mid\_str), count);

mid\_str[count] = '\0';

insert(i, mid\_str);

}

template <class inputiterator>

void BasicString::insert(const\_iterator pos, inputiterator first, inputiterator last) //insert(iterators)

{

size\_type temp = pos - begin();

size\_type n = distance(first, last);

if (\_size + n > \_capacity)

resize(\_size + n);

else

\_size += n;

pos = begin() + temp;

for (iterator i = end() - 1; i != pos + n - 1; --i)

\*i = \*(i - n);

for (iterator i = pos; i != pos + n && first != last; ++i)

\*i = \*first++;

}

void BasicString::insert(Index i, const std::string& str)

{

insert(i, str.c\_str());

}

void BasicString::insert(Index i, const std::string& str, size\_t count)

{

insert(i, str.c\_str(), count);

}

void BasicString::insert(size\_t count,iterator i, char c,int value)

{

insert(i - c\_str\_, count, c);

}

void BasicString::insert(const std::string &str, iterator i ,int value)

{

insert(i - c\_str\_, str);

}

void BasicString::append(size\_t count, char c)

{

char\* tmp\_str = new char[((count/16)+1)\*16];

std::memset(tmp\_str, c, count);

tmp\_str[count] = '\0';

append(tmp\_str, (int)0, count);

delete[] tmp\_str;

}

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

{

append(str, 0, std::strlen(str));

}

void BasicString::append(const char\* str, Index i, size\_t count)

{

if (size\_ + count > capacity\_) {

change\_capacity(round\_size(size\_ + count + 1));

}

std::strncpy(this->c\_str\_ + size\_, str + i, count);

size\_ += count;

c\_str\_[size()] = '\0';

}

void BasicString::append(const std::string& str)

{

append(str.c\_str(), 0, str.size());

}

void BasicString::append(const std::string& str, Index i, size\_t count)

{

append(str.c\_str(), i, count);

}

BasicString::Index BasicString::find(const char\* substr) const

{

return find(substr, 0);

}

BasicString::Index BasicString::find(const char\* substr, Index i\_) const

{

char c;

for (int i = i\_; i <= size\_ - std::strlen(substr); ++i) {

c = \*(c\_str\_ + i + std::strlen(substr));

\*(c\_str\_ + i + std::strlen(substr)) = 0;

if (!std::strcmp(c\_str\_ + i, substr)) {

\*(c\_str\_ + i + std::strlen(substr)) = 0;

return i;

}

\*(c\_str\_ + i + std::strlen(substr)) = c;

}

return static\_cast <Index> (-1);

}

BasicString::Index BasicString::find(const std::string& substr) const

{

return find(substr.c\_str(), 0);

}

BasicString::Index BasicString::find(const std::string& substr, Index i\_) const

{

return find(substr.c\_str(), i\_);

}

void BasicString::replace(Index i, size\_t count, const char\* str)

{

if (i + count > size\_) {

throw "Out of range";

}

erase(i, count);

insert(i, str);

}

void BasicString::replace(Index i, size\_t count, const std::string& str)

{

replace(i, count, str.c\_str());

}

void BasicString::replace(size\_t count, iterator i , char \*str)

{

replace(i - c\_str\_, count, str);

}

void BasicString::replace(size\_t count, iterator i, const std::string &str)

{

replace(i - c\_str\_, count, str);

}

BasicString BasicString::substr(Index i, size\_t count) const

{

if (i + count >= size\_) {

throw "Out of range";

}

BasicString ret;

ret.insert(0, this->c\_str() + i, count);

return ret;

}

BasicString BasicString::substr(Index i) const

{

if (i >= size\_) {

throw "Out of range";

}

BasicString ret;

ret.insert(0, this->c\_str() + i);

return ret;

}

void BasicString::erase(Index i\_, size\_t count)

{

if ((i\_ < 0) || (i\_ > size\_)) {

throw "Out of range";

}

for (int i = i\_; i < size\_ - count; ++i) {

c\_str\_[i] = c\_str\_[i + count];

}

for (int i = size\_ - count; i < size\_; ++i) {

c\_str\_[i] = '\0';

}

size\_ -= count;

}

void BasicString::erase(size\_t count, iterator i)

{

erase(i - c\_str\_, count);

}

int BasicString::to\_int() const

{

return atoi(c\_str());

}

char BasicString::at(int i) const

{

if ((i < 0) || (i >= size\_)) {

throw "Out of range";

}

return c\_str\_[i];

}

BasicString::iterator BasicString::begin() const

{

return iterator(&c\_str\_[0]);

}

BasicString::iterator BasicString::end() const

{

return iterator(c\_str\_ + size\_);

}

BasicString::const\_iterator BasicString::cbegin()

{

return const\_iterator(&c\_str\_[0]);

}

BasicString::const\_iterator BasicString::cend()

{

return const\_iterator(&c\_str\_[size\_]);

}

BasicString::reverse\_iterator BasicString::rbegin()

{

return reverse\_iterator(end());

}

BasicString::const\_reverse\_iterator BasicString::rbegin() const

{

return const\_reverse\_iterator(end());

}

BasicString::const\_reverse\_iterator BasicString::crbegin() const

{

return const\_reverse\_iterator(end());

}

BasicString::reverse\_iterator BasicString::rend()

{

return reverse\_iterator(begin());

}

BasicString::const\_reverse\_iterator BasicString::rend() const

{

return const\_reverse\_iterator(begin());

}

BasicString::const\_reverse\_iterator BasicString::crend() const

{

return const\_reverse\_iterator(begin());

}

char\* BasicString::c\_str() const

{

return c\_str\_;

}

void BasicString::change\_capacity(size\_t new\_capacity)

{

if (capacity() == new\_capacity) {

return;

}

char \* tmp = c\_str\_;

c\_str\_ = nullptr;

c\_str\_ = new char[new\_capacity];

if (new\_capacity > capacity()) {

std::strcpy(c\_str\_, tmp);

}

else {

std::strncpy(c\_str\_, tmp, new\_capacity - 1);

}

delete[] tmp;

capacity\_ = new\_capacity;

if (size() >= capacity()) {

size\_ = capacity() - 1;

}

}

size\_t BasicString::round\_size(size\_t size)

{

return ((((size) / default\_size) + 1) \* default\_size);

}

BasicString operator+ (const BasicString& left, const char\* right)

{

BasicString res = left;

return (res += right);

}

BasicString operator+ (const BasicString& left, const BasicString& right)

{

return (left + right.c\_str());

}

std::ostream& operator<< (std::ostream& os, const BasicString& str)

{

return (os << str.c\_str());

}

std::istream& operator >> (std::istream& is, BasicString& str)

{

str.clear();

char tmp[100] = { 0 };

int i;

char c;

for (i = 0; (c = is.get()) != '\n'; ++i) {

tmp[i] = c;

}

tmp[i] = '\0';

str.insert(str.size(), tmp, sizeof(tmp));

return is;

}

APPENDIX 4

*WideString.cpp:*

#include "BasicString.h"

WideString::WideString()

: size\_(0),

capacity\_(default\_size),

c\_str\_(new wchar\_t[default\_size]),

data\_(nullptr)

{

c\_str\_[0] = '\0';

}

WideString::WideString(const WideString& str) // copy constructor

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

capacity\_(str.capacity\_),

c\_str\_(new wchar\_t[str.capacity\_]),

data\_(nullptr)

{

std::wcscpy(this->c\_str\_, str.c\_str\_);

}

WideString::WideString(const BasicString& str) // copy constructor

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

capacity\_(str.capacity()),

c\_str\_(new wchar\_t[str.capacity()]),

data\_(nullptr)

{

wcscpy(this->c\_str\_, \_bstr\_t(str.c\_str()));

}

WideString::WideString(WideString&& str):

size\_(str.size()),

capacity\_(str.capacity()),

c\_str\_(new wchar\_t[str.capacity()]),

data\_(nullptr) // move constructor

{

\*this = std::move(str);

}

WideString::WideString(const wchar\_t\* str)

{

size\_ = std::wcslen(str);

capacity\_ = round\_size(size\_);

c\_str\_ = new wchar\_t[capacity\_];

std::wcscpy(this->c\_str\_, str);

data\_ = nullptr;

}

WideString::WideString(std::initializer\_list<wchar\_t>& il)

: size\_(il.size()),

capacity\_(round\_size(il.size())),

c\_str\_(new wchar\_t[round\_size(il.size())]),

data\_(nullptr)

{

std::wcscpy(this->c\_str\_, il.begin());

}

WideString::WideString(const wchar\_t\* str, size\_t count)

{

size\_ = count;

capacity\_ = round\_size(size\_);

c\_str\_ = new wchar\_t[capacity\_];

std::memcpy(this->c\_str\_, str, count);

data\_ = nullptr;

}

WideString::WideString(size\_t count, wchar\_t c)

{

size\_ = count;

capacity\_ = round\_size(size\_);

c\_str\_ = new wchar\_t[capacity\_];

std::memset(c\_str\_, c, count);

data\_ = nullptr;

}

WideString::WideString(unsigned int num)

{

wchar\_t c;

auto byte\_size{ [](unsigned int num) -> int {

return 1 + static\_cast<int> (std::log2(num)) / 8;

} };

size\_ = byte\_size(num) \* 2 + 2; // 1 byte in hex = 2 symbols. + 2 symbols for '0', 'x'

capacity\_ = round\_size(size());

c\_str\_ = new wchar\_t[capacity()];

c\_str\_[size()] = '\0';

int i = size() - 1;

while(i >= 2) {

c = (num & 0xf);

if (c <= 9) {

c\_str\_[i] = c + '0';

}

else {

c\_str\_[i] = c - 10 + 'a';

}

num >>= 4;

--i;

}

c\_str\_[0] = '0';

c\_str\_[1] = 'x';

data\_ = nullptr;

}

WideString::~WideString()

{

delete[] c\_str\_;

c\_str\_ = nullptr;

if (data\_ != nullptr) {

delete[] data\_;

data\_ = nullptr;

}

}

WideString& WideString::operator= (const wchar\_t\* str)

{

if (this->capacity\_ < std::wcslen(str)) {

change\_capacity(round\_size(std::wcslen(str)));

}

std::wcscpy(this->c\_str\_, str);

this->size\_ = std::wcslen(str);

return\*this;

}

WideString& WideString::operator= (const BasicString& str)

{

if (this->capacity\_ < std::strlen(str.c\_str())) {

change\_capacity(round\_size(std::strlen(str.c\_str())));

}

std::wcscpy(this->c\_str\_, \_bstr\_t(str.c\_str()));

this->size\_ = std::strlen(str.c\_str());

return\*this;

}

WideString& WideString::operator= (const WideString& str) // copy assigment

{

if (this != &str) {

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

}

else {

return\*this;

}

}

WideString& WideString::operator= (WideString&& str) // move assigment

{

if (this != &str) {

delete[] this->c\_str\_;

this->c\_str\_ = str.c\_str\_;

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

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

str.c\_str\_ = nullptr;

str.size\_ = 0;

str.capacity\_ = 0;

}

return\*this;

}

WideString& WideString::operator= (wchar\_t c)

{

this->c\_str\_[0] = c;

this->size\_ = 1;

return\*this;

}

WideString& WideString::operator+= (const wchar\_t\* str)

{

if (this->capacity\_ <= this->size\_ + std::wcslen(str)) {

change\_capacity(round\_size(this->size\_ + std::wcslen(str)));

}

std::wcscat(this->c\_str\_, str);

this->size\_ += std::wcslen(str);

return\*this;

}

WideString& WideString::operator+= (const std::string& str)

{

return (\*this += \_bstr\_t(str.c\_str()));

}

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

{

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

}

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

{

if (this == &str) {

return true;

}

return (std::wcscmp(this->c\_str\_, str.c\_str()) == 0);

}

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

{

return !(\*this == str);

}

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

{

return std::wcscmp(this->c\_str(), str.c\_str\_) > 0;

}

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

{

return std::wcscmp(this->c\_str(), str.c\_str\_) < 0;

}

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

{

return std::wcscmp(this->c\_str(), str.c\_str\_) >= 0;

}

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

{

return std::wcscmp(this->c\_str(), str.c\_str\_) <= 0;

}

void WideString::clear()

{

\*this = WideString();

}

void WideString::shrink\_to\_fit()

{

change\_capacity(size() + 1);

}

wchar\_t WideString::operator[] (Index i) const

{

return c\_str\_[i];

}

size\_t WideString::size() const

{

return size\_;

}

size\_t WideString::capacity() const

{

return capacity\_;

}

bool WideString::empty() const

{

return (size() == 0);

}

wchar\_t\* WideString::data()

{

if (data\_ != nullptr) {

delete[] data\_;

}

data\_ = new wchar\_t[size()];

std::memcpy(data\_, c\_str\_, size());

return data\_;

}

void WideString::insert(Index i\_, const wchar\_t\* mid\_str)

{

if (i\_ > size()) {

throw "Out of range";

}

if (size\_ + std::wcslen(mid\_str) >= capacity\_) {

change\_capacity(round\_size(size\_ + std::wcslen(mid\_str) + 1));

}

for (int i = size\_ + std::wcslen(mid\_str) - 1; i >= i\_ + std::wcslen(mid\_str); --i) {

c\_str\_[i] = c\_str\_[i - std::wcslen(mid\_str)];

}

for (int i = i\_, j = 0; i < i\_ + std::wcslen(mid\_str); ++i, ++j) {

c\_str\_[i] = mid\_str[j];

}

size\_ += std::wcslen(mid\_str);

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

}

void WideString::insert(Index i, size\_t count, wchar\_t c)

{

wchar\_t\* mid\_str = new wchar\_t[((count / 8) + 1) \* 8];

std::wmemset(mid\_str, c, count);

mid\_str[count] = '\0';

insert(i, mid\_str);

delete[] mid\_str;

}

void WideString::insert(Index i, wchar\_t\* mid\_str, size\_t count)

{

wmemcpy(mid\_str, static\_cast<const wchar\_t\*>(mid\_str), count);

mid\_str[count] = '\0';

insert(i, mid\_str);

}

template <class inputiterator>

void WideString::insert(const\_iterator pos, inputiterator first, inputiterator last) //insert(iterators)

{

size\_type temp = pos - begin();

size\_type n = distance(first, last);

if (\_size + n > \_capacity)

resize(\_size + n);

else

\_size += n;

pos = begin() + temp;

for (iterator i = end() - 1; i != pos + n - 1; --i)

\*i = \*(i - n);

for (iterator i = pos; i != pos + n && first != last; ++i)

\*i = \*first++;

}

void WideString::insert(Index i, const std::string& str)

{

insert(i, \_bstr\_t(str.c\_str()));

}

void WideString::insert(Index i, const std::string& str, size\_t count)

{

insert(i, \_bstr\_t(str.c\_str()), count);

}

void WideString::insert(size\_t count, iterator i, wchar\_t c,int value)

{

insert(i - c\_str\_, count, c);

}

void WideString::insert(const std::string &str, iterator i,int value)

{

insert(i - c\_str\_, \_bstr\_t(str.c\_str()));

}

void WideString::append(size\_t count, wchar\_t c)

{

wchar\_t\* tmp\_str = new wchar\_t[((count / 8) + 1) \* 8];

wmemset(tmp\_str, c, count);

tmp\_str[count] = '\0';

append(tmp\_str, (int)0, count);

delete[] tmp\_str;

}

void WideString::append(const wchar\_t\* str)

{

append(str, 0, std::wcslen(str));

}

void WideString::append(const wchar\_t\* str, Index i, size\_t count)

{

if (size\_ + count > capacity\_) {

change\_capacity(round\_size(size\_ + count + 1));

}

std::wcsncpy(this->c\_str\_ + size\_, str + i, count);

size\_ += count;

c\_str\_[size()] = '\0';

}

void WideString::append(const std::string& str)

{

append(\_bstr\_t(str.c\_str()), 0, str.size());

}

void WideString::append(const std::string& str, Index i, size\_t count)

{

append(\_bstr\_t(str.c\_str()), i, count);

}

WideString::Index WideString::find(const wchar\_t\* substr) const

{

return find(substr, 0);

}

WideString::Index WideString::find(const wchar\_t\* substr, Index i\_) const

{

wchar\_t c;

for (int i = i\_; i <= size\_ - std::wcslen(substr); ++i) {

c = \*(c\_str\_ + i + std::wcslen(substr));

\*(c\_str\_ + i + std::wcslen(substr)) = 0;

if (!std::wcscmp(c\_str\_ + i, substr)) {

\*(c\_str\_ + i + std::wcslen(substr)) = 0;

return i;

}

\*(c\_str\_ + i + std::wcslen(substr)) = c;

}

return static\_cast <Index> (-1);

}

WideString::Index WideString::find(const std::string& substr) const

{

return find(\_bstr\_t(substr.c\_str()), 0);

}

WideString::Index WideString::find(const std::string& substr, Index i\_) const

{

return find(\_bstr\_t(substr.c\_str()), i\_);

}

void WideString::replace(Index i, size\_t count, const wchar\_t\* str)

{

if (i + count > size\_) {

throw "Out of range";

}

erase(i, count);

insert(i, str);

}

void WideString::replace(Index i, size\_t count, const std::string& str)

{

replace(i, count, \_bstr\_t(str.c\_str()));

}

void WideString::replace(size\_t count, iterator i , wchar\_t \*str)

{

replace(i - c\_str\_, count, str);

}

WideString WideString::substr(Index i, size\_t count) const

{

if (i + count >= size\_) {

throw "Out of range";

}

WideString ret;

ret.insert(0, this->c\_str() + i, count);

return ret;

}

WideString WideString::substr(Index i) const

{

if (i >= size\_) {

throw "Out of range";

}

WideString ret;

ret.insert(0, this->c\_str() + i);

return ret;

}

void WideString::erase(Index i\_, size\_t count)

{

if ((i\_ < 0) || (i\_ > size\_)) {

throw "Out of range";

}

for (int i = i\_; i < size\_ - count; ++i) {

c\_str\_[i] = c\_str\_[i + count];

}

for (int i = size\_ - count; i < size\_; ++i) {

c\_str\_[i] = '\0';

}

size\_ -= count;

}

void WideString::erase(size\_t count, iterator i)

{

erase(i - c\_str\_, count);

}

int WideString::to\_int() const

{

return \_wtoi(c\_str());

}

wchar\_t WideString::at(int i) const

{

if ((i < 0) || (i >= size\_)) {

throw "Out of range";

}

return c\_str\_[i];

}

WideString::iterator WideString::begin() const

{

return iterator(&c\_str\_[0]);

}

WideString::iterator WideString::end() const

{

return iterator(c\_str\_ + size\_);

}

WideString::const\_iterator WideString::cbegin()

{

return const\_iterator(&c\_str\_[0]);

}

WideString::const\_iterator WideString::cend()

{

return const\_iterator(&c\_str\_[size\_]);

}

WideString::reverse\_iterator WideString::rbegin()

{

return reverse\_iterator(end());

}

WideString::const\_reverse\_iterator WideString::rbegin() const

{

return const\_reverse\_iterator(end());

}

WideString::const\_reverse\_iterator WideString::crbegin() const

{

return const\_reverse\_iterator(end());

}

WideString::reverse\_iterator WideString::rend()

{

return reverse\_iterator(begin());

}

WideString::const\_reverse\_iterator WideString::rend() const

{

return const\_reverse\_iterator(begin());

}

WideString::const\_reverse\_iterator WideString::crend() const

{

return const\_reverse\_iterator(begin());

}

wchar\_t\* WideString::c\_str() const

{

return c\_str\_;

}

void WideString::change\_capacity(size\_t new\_capacity)

{

if (capacity() == new\_capacity) {

return;

}

wchar\_t \* tmp = c\_str\_;

c\_str\_ = nullptr;

c\_str\_ = new wchar\_t[new\_capacity];

if (new\_capacity > capacity()) {

std::wcscpy(c\_str\_, tmp);

}

else {

std::wcsncpy(c\_str\_, tmp, new\_capacity - 1);

}

delete[] tmp;

capacity\_ = new\_capacity;

if (size() >= capacity()) {

size\_ = capacity() - 1;

}

}

size\_t WideString::round\_size(size\_t size)

{

return ((((size) / default\_size) + 1) \* default\_size);

}

WideString operator+ (const WideString& left, const wchar\_t\* right)

{

WideString res = left;

return (res += right);

}

WideString operator+ (const WideString& left, const WideString& right)

{

return (left + right.c\_str());

}

WideString operator+ (const WideString& left, const std::wstring& right)

{

return (left + right.c\_str());

}

std::ostream& operator<< (std::ostream& os, const WideString& str)

{

return (os << str.c\_str());

}

std::istream& operator >> (std::istream& is, WideString& str)

{

str.clear();

char tmp[100] = { 0 };

int i;

char c;

for (i = 0; (c = is.get()) != '\n'; ++i) {

tmp[i] = c;

}

tmp[i] = '\0';

str.insert(str.size(), tmp, sizeof(tmp));

return is;

}