**CISS245: Advanced Programming**
**Assignment 9**

Name: _____

Objectives

- Write constructors.
- Use default values for parameters.
- Write get and set methods.
- Overloaded operators.
- Pass objects to methods by reference.
- Pass objects to methods by constant reference.
- Write constant methods.
- Use this pointer to return a copy of an object.
- Write functions with object parameters.

The goal is to implement a string class. Recall from CISS240, we have the concept of C-strings. There are lots of functions provided by the compiler to operate on C-strings. (Either see CISS240 notes on C-strings or look at the C-string review section below.) The goal is is to write a string class, called `mystring`, so that `mystring` objects model C-strings and `mystring` objects will be easier to work with than C-string. For instance if you have two C-strings `s` and `t` and you want to concatenate them (i.e., join them) into `u`, you have to do

```
u[0] = '\0';
strcat(u, s);
strcat(u, t);
```

Using our `mystring` class, we can do this:

```
u = s + t;
```

You are given the following (possibly incomplete files and possible incorrect files – these are meant to help you get started):

- The test codes is given. It must be used in your project.
- The header file is included.
- The implementation file contains the implementation of `operator<<`.

Make sure you read the whole document before you dive into the code.

Important warning: Again, the files are meant to be skeleton files and might not be complete and might have deliberate missing details or even errors. The goal of the skeleton files it to give you something to work with.

If you're doing a copy-and-paste of the given code, note that some characters might be changed by PDF to other characters. In particular the `'-'`, character might actually not be the dash character. (There are two ASCII characters that look like `-`.) Looking at the compiler error message will help you find these minor annoying issues so that you can correct them.

All methods must be constant whenever possible. All parameters which are objects (or struct variables) must be pass by reference or pass by constant reference as much as possible. (I already mentioned these two points in my notes and in class.)

Let me know ASAP if you see a typo.

Q1. Strings

Review your notes on C-strings right away. The following is a quick review and is not complete.

A C-string (also called a null-terminated string is a character array containing the character `'\0'`. The character `'\0'` has an ASCII integer value of 0. Try this:

```
char s[10] = {'a', 'b', 'c', '\0', 'd', 'e'};
std::cout << s[0] << ' ' << int(s[0]) << '\n'; // prints 'a' and ASCII
                                               // value of 'a'
std::cout << s[3] << ' ' << int(s[3]) << '\n'; // prints '\0' and the
                                               // ASCII value of '\0'
```

(Google "ascii table".)

Note that `s` is a character array of size 10. The first 6 characters are initialized. The last 4 are therefore initialized to the character with integer value (ASCII value) of 0, i.e., the remaining 4 are initialzed to `'\0'`. The character `'\0'` is also called the null character since the ASCII value is 0.

When we call `s` a string, what we really meant is the `"abc"`, i.e., 4 characters with the last being character `'\0'`. The point of the `'\0'` character is simply to tell the computer when the string data (characters) end. In other words, you can (and should) think `'\0'` as a sentinel value in the array. The `'\0'` is therefore used to indicate "when the string ends within the character array". This is used for instance in printing `s`:

```
char s[10] = {'a', 'b', 'c', '\0', 'd', 'e'};
std::cout << s << std::endl;
```

You see that the printing of the characters stop once the printing operation sees the character `'\0'`, i.e., the `'\0'` terminates the print operation. You can therefore tell that the `operator<<` probably looks "something like this":

```
std::ostream & operator<<(std::ostream & cout, const char s[])
{
    int i = 0;
    while (s[i] != '\0')
    {
        cout << s[i];
        i++;
    }
    return cout;
```

```
}
```

As you can see in the code above: `'\0'` is used to terminate (i.e., as a sentinel value) for the print operation.

Recall that the double-quotes string notation

```
char s[10] = "abc";
```

is really the same as (a shorthand for):

```
char s[10] = {'a', 'b', 'c', '\0'};
```

Make sure you see the difference between `s` as an array and s as a string:

- as an array `s` has 10 characters - the array size of `s` is 10
- as a string `s` has 3 characters - the string length of `s` is 3

There are several C-string functions that come with all C++ compilers. Here are some (again refer to my old notes on strings):

```
#include <iostream>
#include <cstring> // include prototypes for C-string functions

int main()
{
    char s[1024] = "hello world";
    char t[50] = "ABCDEF";
    char u[100] = "ABCDEF\0GHIJKL";

    std::cout << strlen(s) << '\n'   // string length function:
                                     // length of the string, i.e.,
                                     // number of characters up to but
                                     // not including '\0'

    strcat(s, t);              // string concatenation function:
    std::cout << s << '\n';    // string concatenation, i.e., join the string
                               // t to s.
    int i = strcmp(s, t);
    std::cout << i << '\n';              // compare s and t. The return
    std::cout << strcmp(t, u) << '\n';   // value is 0 if the strings
                                         // compared are the same. Otherwise
                                         // it's nonzero.
```

```
    return 0;
}
```

Make sure you study and run the above code.

There are many other C-string functions. Google "c string functions" or refer to my notes or refer to any C/C++ textbook.

Note that `s` is a character array of size 1000, then at most you can store a string of length 999 since one character is used to indicate end-of-string.

There's another way to describe a string: that's with a character array and a "length" integer variable. For instance if I have this:

```
char s[1024] = "hello world ... how are you ... ?"
int len_s = 11;
```

Then the variable `len_s` tells me that only 11 characters in the character array are considered part of the string that `s` is trying to represent. Note: `s` does have a capacity to hold 1024. Only the first 11 characters are considered the string that `s` is representing.

This style of representing strings is called a length-based representation. (We've talked about this before in CISS240.)

What are the benefits of using a length-based representation for strings? Well think about it. Suppose you want to concatenate two C-strings

```
char s[1024] = "hello world ... how are you ?";
char t[1024] = " I'm fine.";
strcat(s, t);
```

Do you realize that the characters `' '`, `'I'`, `'\''`, `'m'`, etc, from `t` must be copied to the end of `s`, overwriting the `'\0'` of `s`? That means that I have to know where is the `'\0'` in `s`. That means that before I begin concatenating `t` to `s`, I have to find `'\0'`. If `s` is a very long string, that will waste some time. In particular this will get slower and slower:

```
char s[1024] = "hello world ... how are you?";
char t[1024] = " I'm fine.";
for (int i = 0; i < 100; ++i)
{
    strcat(s,t);
```

```
}
```

Of course computers are so fast nowadays that you won't notice the difference if this is the only code in your program. If your program concats millions of strings (which is very common in a buisness application or an AI program on natural language processing), the effect will be noticeable.

For this question, we're going to build a string class that uses a length-based approach. The class will hold a character array of size 1024 and it will also hold a length variable. There's another variable that tell us that the maximum capacity is 1024 – this will be a constant since (for this assignment question), the size of the character array does not change. (Later you will work on the concept of a dynamic array.)

Strings are extremely important. Look at this document. Clearly it's made up of strings. Strings (or more accurate, languages) are used to communicate ideas and thoughts and emotions. Since human ideas are expressed strings, the ability to compute with strings is important when it comes to analyzing the thoughts of a human. NLP (natural language processing) is an area of study in computer science with the goal of understanding the structure and meaning behind strings (written or oral) prodcued by human beings.

If you want to try to play with a fake psycotherapist, go to this website and talk to Eliza and explain to "her" (it?) your problems: http://www.manifestation.com/neruotoys/eliza.php3. It's not particularly smart but it does show you that Eliza can at least understand basic English structure and therefore can pretend to be intelligent to some extent.

As an example of an object of our class, suppose I execute this:

```
mystring y("abc");
```

then `y` is a `mystring` object with three instance variables:

```
y.s_         a character array of size 1024
y.length_    an int variable
y.capacity_  an int constant (set to 1024)
```

The values placed in `y` by this constructor are:

```
y.s_         array containing 'a', 'b', 'c', ...
y.length_    3
y.capacity_  1024
```

Since `y.length_` is 3 in this case, only the first 3 characters of `y.s_` are relevant. The constructor in this case is

```
mystring::mystring(const char x[]);
```

since this one accepts a character array.

The `capacity_` instance variable can be set in the initializer of the constructor since we know that it's just 1024. A C-string is passed into the constructor as an argument. Note that in this case we know that `y.length_` is 3, but the constructor needs to scan the character array `'a'`, `'b'`, `'c'`, `'\0'`, ... to know that there are three characters before the first `'\0'`. Therefore `y.length_` cannot be directly initialized. So `y.length_` is set in the body of the constructor. The character array `y.s_` is also set in the body of the constructor. Therefore the constructor that accepts a character array looks like this (in pseudocode):

```
mystring::mystring(const char x[]);
   Initialize capacity_ to 1024
   Loop over x[i] (i = 0, 1, 2, 3, 4, 5, ...) and put x[i] into s_[i]; stop
   when x[i] is '\0':

           x[0]  x[1]  x[2] ...
             ↓     ↓     ↓
           s_[0] s_[1] s_[2] ...

   Also, set length_ to the string length of x_, i.e., length_ is set to the
   number of characters in x up to but not including the first '\0'.
```

Note that in the class definition of `mystring` in the header file, there are three constructors: one that accepts a character array, one that is the default constructor, and there's the copy constructor. Note that in the test code, there's another that accepts a character. This is also mentioned in the documentation. Therefore you will need to write a constructor that accepts a character.

Here are more examples. If I do:

```
mystring a("hello");  // using constructor that accepts a character array
```

then

```
a.s_         has array values 'h','e','l','l','o', ...
                 (we don't care about the other values)
a.length_    has value 5
a.capacity_  has value 1024
```

If I do

```
mystring b;  //using the default constructor
```

then

```
b.s_         we don't care about the values
b.length_    has value 0
b.capacity_  has value 1024
```

If I do

```
mystring c('$');   // using the constructor that accepts a character
```

then

```
c.s_         has values '$', ... (we don't care about the rest)
c.length_    has value 1
c.capacity_  has value 1024
```

The copy constructor does the obvious. For instance if object `a` is the above, then if I do this:

```
mystring d(a);
```

then

```
d.s_         has array values 'h','e','l','l','o', ...
                (we don't care about the other values)
d.length_    has value 5
d.capacity_  1024
```

## The mystring header and implementation file

a09q01/skel/mystring.h

```
/*****************************************************************************

File  : mystring.h
Author:
Date  :

The following describes how to use the mystring class.

mystring x;                     // x models the empty string ""
std::cout << x << std::endl;    // operator<< is overloaded to print
                                // mystring objects. In this case
                                // no character is printed because
                                // x.length_ is 0

mystring y("abc");              // y is the string "abc", i.e., y.s_ is
                                // "abc"
std::cout << y << std::endl;    // abc appears in the output window

mystring z('a');                // z is the string "a", i.e., z.s_ is
                                // "a".

mystring w(10, ' ');            // w.s_ is the string "          ",
                                // i.e., a string of 10 ' '.

x += y;                         // concatenates y to x. x becomes "abc"
(x += y) += y;                  // x becomes "abcabcabc"
y += z;                         // y becomes "abca"
x = y + '?';                    // x becomes "abca?"
x = y + z;                      // x becomes "abcaa"

y[0] = 'A';                     // y becomes "Abca"

y[4] = 'z'                      // y is still "Abca" since the length_
                                // is still 4, i.e., y[4] is outside
                                // the boundary of the array.

y.resize(5);                    // y becomes "Abcaz".
y.resize(10, '$');              // y becomes "Abcaz$$$$$"
y.resize(3, '-');               // y becomes "Abc"

x = z;                          // x becomes "a".
y = "hello world";              // y becomes "hello world".

std::cout << (x == y) << std::endl;   // false, i.e., 0 is printed
std::cout << (x == "a") << std::endl; // true, i.e., 1 is printed
std::cout << (x == 'a') << std::endl; // true, i.e., 1 is printed
```

```
x = y.substr(6, 2);            // y.substr(6, 2) returns a substring
                               // of y (as a mystring object) at index
                               // 6 of length 2, i.e., x becomes
                               // "wo".
x = "hello world";
y = "world";
z = "columbia";
std::cout << x.find(y) << '\n';     // 6 is printed
std::cout << x.find(z) << '\n';     // -1 is printed
std::cout << x.find(' ') << '\n';   // 5 is printed
std::cout << x.find("ell") << '\n'; // 1 is printed

**********************************************************************/
#ifndef MYSTRING_H
#define MYSTRING_H

#include <iostream>


class mystring
{
public:
    mystring(const char x[])
        : capacity_(1024)  // NOTE: capacity is a const. You MUST initialize
                           // it with an initializer list
    {
        // Copy the relevant characters x[0], ... to s[0], ...
        // and set the length_ appropriately. For instance if x is "abc"
        // the length_ is set to 3.
    }

    mystring(char c)
    {
        // Copy c to s[0]. Set capacity_ and length_ accordingly.
    }

    mystring()  // The default constructor must initialize the object to
                // model an empty string, i.e., s_ is "".
                // 1. Do you need to set capacity to a value?
                // 2. What must you set length_ to?
    {}

    mystring(const mystring & x); // The copy constructor copies the
                                  // relevant characters in x.s_ to
                                  // the object being constructed.
                                  // The length_ of the object constructed
                                  // is set to the length_ of x.

    bool operator==(const mystring &); // Returns true iff the string
                                  // modeled by the object is the
```

```
                                // same as the string modeled by
                                // the parameter.
    bool operator!=(const mystring &);
    bool empty() const;         // Returns true iff the string modeled
                                // is "".

    mystring & operator=(const mystring & x); // After assignment, the
                                // (*this) == x is true.
    mystring & operator=(char c); // object becomes string of length_ 1
                                // with the parameter as the character
                                // at index 0.

    void clear();               // After this, the object models "".

    char operator[](int) const;   // bracket operator to access the
    char & operator[](int);       // character of the string

    mystring & operator+=(const mystring &); // concatenation operators
    mystring & operator+=(char c);
    mystring operator+(const mystring &);
    mystring operator+(char);

    void resize(int i);           // Sets the length_ to i.
    void resize(int i, char c);   // Sets the length_ to i and if i is
                                // greater than the original length,
                                // then the new characters are set to c.

    int find(char c, int start = 0); // Returns the index of the first
                                // occurrence of c in string s_. If c is
                                // not found, -1 is returned.
                                // Returns the index of the first
                                // occurrence of the parameter.

    int find(const mystring & s, int start = 0); // Returns the index of the
                                // first occurrence of s in string s_.
                                // If s is not found, -1 is returned.
                                // Returns the index of the first
                                // occurrence of the parameter.

    mystring substr(int i, int len); // Return a mystring object
                                // using the substring of s_ starting
                                // at index i and length_ len.

private:
    const int capacity_;
    char s_[1024];
    int length_;
};

std::ostream & operator<<(std::ostream &, const mystring &);
std::istream & operator>>(std::istream &, mystring &);
```

```
bool operator== (const char[], const mystring &);

#endif
```

a09q01/skel/mystring.cpp

```
/*****************************************************************

File  : mystring.cpp
Author:
Date  :

*****************************************************************/

#include <iostream>
#include "mystring.h"

std::ostream & operator<<(std::ostream & cout, const mystring & x)
{
       // Get the length of x, say you store it in local variable len.
       // Print the string in x, i.e., x.s[0], ..., x.s[len - 1].
       // Finally, return the reference cout.
}

std::istream & operator>>(std::istream & cin, mystring & x)
{
       // Get a string from the user, say you store it in local character
       // array t of size 1024. (use cin and t.)
       // Copy t[i], ... to x.s[i] starting with i = 0;. Stop when t[i] is
       // '\0'. Set the length of x accordingly.
       // Finally, return the reference cin.
       //
       // To minimize code duplication, it's easier to do the following:
       // 1. Clear x (see the method mystring::clear())
       // 2. Perform += to add the string in t to x (i.e., to x.s).
}
```

Aside. Note that all `mystring` objects can represent a string of at most 1024 characters. This is a problem. For instance suppose I want to do an AI chatbot. Clearly I need to process strings. If I use your `mystring`, then I'm probably wasting a lot of memory. Most people don't type that many characters for each response during a chat session. 300 is probably more than enough. On the other hand if I am writing a system to analyze web documents (like what Google is doing this instant on thousands of machines), then I probably will need way more than 1024 characters for each web document.

### Test File

The following is only a skeleton test code. Complete it and test your library thoroughly.
`a09q01/skel/main.cpp`

```
/*************************************************************************

File  : main.cpp
Author:
Date  :

Description
This is the test program for the mystring classes.

*************************************************************************/

#include <iostream>
#include <iomanip>
#include <cmath>
#include <cstdlib>
#include "mystring.h"

void test_constructor()
{
    mystring s;
    std::cout << s << std::endl;
}

void test_constructor_char()
{
    char c;
    std::cin >> c;
    mystring s(c);
    std::cout << s << std::endl;
}

void test_constructor_char_array()
{
    char s[100];
    std::cin >> s;
    mystring x(s);
    std::cout << x << std::endl;
}

void test_copy_constructor()
{
    char s[100];
    std::cin >> s;
    mystring x(s);
    mystring y(x);
```

```
    std::cout << y << std::endl;
}

void test_eq()
{
    mystring x, y;
    std::cin >> x >> y;
    std::cout << (x == y) << std::endl;
}

void test_ne()
{
    mystring x, y;
    std::cin >> x >> y;
    std::cout << (x != y) << std::endl;
}

void test_assign()
{
    mystring x, y;
    std::cin >> x;
    y = x;
    std::cout << y << std::endl;
}

void test_empty()
{
    mystring x;
    std::cout << x.empty() << std::endl;
}

void test_bracket_const()
{
    mystring x;
    int i = 0;
    std::cin >> x >> i;
    std::cout << x[i] << std::endl;
}

void test_bracket()
{
    mystring x;
    int i = 0;
    char c;
    std::cin >> x >> i >> c;
    x[i] = c;
    std::cout << x << std::endl;
}

void test_pluseq_mystring()
{
```

```
    mystring x, y, z;
    std::cin >> x >> y >> z;
    (x += y) += z;
    std::cout << x << std::endl;
}

void test_pluseq_char()
{
    mystring x;
    char c, d;
    std::cin >> x >> c >> d;
    (x += c) += d;
    std::cout << x << std::endl;
}

void test_plus()
{
    mystring x, y;
    std::cin >> x >> y;
    std::cout << x + y << std::endl;
}

void test_plus_char()
{
    mystring x, y;
    char c;
    std::cin >> x >> c;
    std::cout << x + c << std::endl;
}

void test_resize()
{
    mystring x;
    int i = 0;
    std::cin >> x >> i;
    x.resize(i);
    std::cout << x << std::endl;
}

void test_resize_char()
{
    mystring x;
    int i = 0;
    char c;
    std::cin >> x >> i >> c;
    x.resize(i, c);
    std::cout << x << std::endl;
}

void test_find_char_int()
{
```

```
    mystring x;
    char c;
    int i = 0;
    std::cin >> x >> c >> i;
    std::cout << x.find(c, i) << std::endl;
}

void test_find_mystring_int()
{
    mystring x, y;
    int i = 0;
    std::cin >> x >> y >> i;
    std::cout << x.find(y, i) << std::endl;
}

void test_substr()
{
    mystring x, y;
    int i, len;
    std::cin >> x >> i >> len;
    std::cout << x.substr(i, len) << std::endl;
}

int main()
{
    int option;
    std::cin >> option;
    switch(option):
    {
        case 1:
            test_constructor();
            break;
        case 2:
            test_constructor_char();
            break;
        case 3:
            test_constructor_char_array();
            break;
        case 4:
            test_copy_constructor();
            break;
        case 5:
            test_input();
            break;
        case 6:
            test_eq();
            break;
        case 7:
            test_ne();
            break;
        case 8:
```

```
            test_assign();
            break;
        case 9:
            test_empty();
            break;
        case 10:
            test_bracket_const();
            break;
        case 11:
            test_bracket();
            break;
        case 12:
            test_pluseq_mystring();
            break;
        case 13:
            test_pluseq_char();
            break;
        case 14:
            test_plus();
            break;
        case 15:
            test_plus_char();
            break;
        case 16:
            test_resize();
            break;
        case 17:
            test_resize_char();
            break;
        case 18:
            test_find_char_int();
            break;
        case 19:
            test_find_mystring_int();
            break;
        case 20:
            test_substr();
            break;
    }

    return 0;
}
```

Here are the test option numbering.

1. Print and `mystring()`
2. Print and `mystring(char)`
3. Print and `mystring(char [])`
4. Print and copy constructor
5. Input
6. `operator==`
7. `operator=!`
8. `operator=`
9. `empty`
10. `char operator[](int) const`
11. `char & operator[](int)`
12. `mystring & operator+=(const mystring &)`
13. `mystring & operator+=(char)`
14. `mystring operator+(const mystring &)`
15. `mystring operator+(char)`
16. `void resize(int i)`
17. `void resize(int i, char c)`
18. `int find(char c, start = 0)`
19. `int find(const mystring &)`
20. `mystring substr(int i, int len)`

There are no tests for nonmember functions other than input and output which are tested in 1–5.

### Reminder: Giving access to instance varaiable by returning a reference to an instance varaiable

If you want to set the value of an instance variable, you can do something like this:

```
#include <iostream>

class X
{
public:
    void set_i(int newi)
    {
        i = newi;
    }

    int i;  // make this public for this experiment
};

int main()
{
    X x;
    x.set_i(5);
    std::cout << x.i << '\n'; // you should get 5
    return 0;
}
```

Another way to achieve the same this is to give the client using your class (in this case client = main) access to the instance variable directly. You do that by returning a reference to the instance variable.

```
#include <iostream>

class X
{
public:
    void set_i(int newi)
    {
        i = newi;
    }
    int & iref()  // obviously this method cannot be constant
    {
        return i;
    }

    int i; // make this public for this experiment
```

```
};

int main()
{
    X x;
    x.set_i(5);
    std::cout << x.i << '\n';  // you should get 5
    x.iref() = 6;              // lefthand side of = is a reference to x.i
    std::cout << x.i << '\n';  // you should get 6
    return 0;
}
```

Make sure you see the difference between `X::set_i(int)` and `X::iref()`. The former sets the value of `i` for you. The client has an indirect write access to `i`. For `X::iref()`, the client has direct access to `i`.

If an instance variable is an array, say `a`, and you want to give a client access to `a[0]`, then you give the client a public method that returns a reference to `a[0]`, not the value of `a[0]`.

Make sure you review the notes on references and pointers.

The above is enough information for this assignment. Read on if want to know a bit more ...

If you have the following in C++ (actually for most languages):

```
x = a + b + c;
```

Notice that there's a difference between the way you use the names on the right (i.e., `a`, `b`, and `c`) and the name on the left (i.e., `x`). Of course `a`, `b`, `c`, `x` all refer to "boxes" containing some value (more accurately, they refer to some part of your computer's memory). Howerver note that for `a`, `b`, `c` , you RETRIEVE (read access) the values at `a`, `b`, and `c`. For `x`, you PUT A VALUE INTO (write access) the box corresponding to `x`. What actually happens is this: you read the values for `a`, `b`, `c` and say `a`, `b`, `c` have values 1, 2, 3, then the exression `a+b+c` generates 6. Now `a`, `b`, `c` themselves have values which are of course at some fixed memory locations. However the value 6 does not have a fixed loaction - it's temporary, so we say that 6 is an **r-value**. On the other hand `x` on the left of the assignment, since it refers to the box of `x`, or more preciesely to a chunck of memory, is an **l-value**.

Informally, you can think of a `r-value` as "something that can be assigned a value and appear on the left of an assignment operator" while a `l-value` is "something that cannot be assigned, is a value, and appears on the right of the assignment operator". That's enough for right now. The real picture is slightly more complicated.

Knowing the above is important because some compilers will use the above terminology

(`l-value` and `r-value`) for error messages. For instance try to compile this program and then read the error message:

```
int main()
{
    2 = 5;
    return 0;
}
```

If you're using g++, it will give you an error message that basically says that the lefthand side of the assignment generates the value 2 which is not a `l-value`, i.e., it cannot be assigned a value.