Classes with Resources Review

Workshop 9 (out of 10 marks - 3% of your final grade)

In this workshop, you are to design and code a class that encapsulates a resource.

LEARNING OUTCOMES

Upon successful completion of this workshop, you will have demonstrated the abilities

- to allocate dynamic memory for an encapsulated resource upon instantiation
- to manage the lifetime of the encapsulated resource
- to overload the member functions to support copying and assignment operations
- to describe to your instructor what you have learned in completing this workshop

SUBMISSION POLICY

The "in-lab" section is to be completed during your assigned lab section. It is to be completed and submitted by the end of the workshop period. If you attend the lab period and cannot complete the in-lab portion of the workshop during that period, ask your instructor for permission to complete the in-lab portion after the period. If you do not attend the workshop, you can submit the “in-lab” section along with your “at-home” section (with a penalty; see below). The “at-home” portion of the lab is due on the day of your next scheduled workshop (23:59).

All your work (all the files you create or modify) must contain your name, Seneca email and student number.

You are responsible to regularly back up your work.

Late submission penalties:

- In-lab submitted late, with at-home: Maximum of 20/50 for in-lab and Maximum of 50/50 for at home

- Workshop late for one week: in-lab, at-home and reflection must all be submitted for maximum of 50 / 100

- Workshop late for more than one week: in-lab, at-home and reflection must all be submitted for maximum of 30 / 100

- If any of in-lab, at-home or reflection is missing the mark will be zero.

INTRODUCTION

Since the beginning of the course, you have been using strings in almost every single workshop. You have used C-style strings, also known as null-terminated strings which are just character arrays with an ASCII 00 character the end to tell you where the strings ends. You used the functions in <cstring> to manipulate them, functions such as **std::strcpy** and **std::strlen**. Most importantly, you have used the **new** and **delete** keywords to manage the memory for your strings.

Today, you will apply all the C++ techniques you have learned to write a **String** class that encapsulates these operations. It will automatically initialize the string to a safe empty state, allow you to copy the string without worrying about memory management, and allow you to do common string operations (joining 2 strings, extracting a substring from inside it, modifying it) using familiar C++ operator syntax.

In-Lab

ASSUMPTIONS FOR IN-LAB

Make the following assumptions to make the in-lab part easier.

- Assume that all strings being input will be less than 50 characters in length. Indetermine length strings will be done at-home.
- Since you will never have to join more than 2 strings, allocate all strings as char arrays of length 100. Do not use dynamic memory(new or delete). That will be done at-home.
- You do not have to do input checking on null pointers. Assume valid inputs. Arbitrary inputs will be handled during the at-home.
- Assume all strings are char (UTF-8) strings. Unicode strings (wide-char strings) to represent other languages will not be handled in this workshop, not in the lab and not in the home.

DESIGN

For the in-lab part, your String class should store a string as a C-style null-terminated string in a char array. This will be referred to as the "string buffer". Make it a fixed sized array of size 100. This will simplify things for you so you can focus on developing all the class members. At home you will make it a dynamic array, whose size changes to however big it has to be to store it's string.

These prototypes are also given in the header file.

Class method	Description
`String()`	**Default constructor.** Should initialize the string to the safe initial state of "". Note, for an empty string, we do not use a nullptr. We simply use a null character. Note: it can be good style to set every single character in the buffer to '\0' here, even though only the first character needs to be '\0'.
`String(const char* p)`	Construct a String from a C-style string in p. For in-lab, assume that the length of the string in p will be less than 50 chars. Note: you do not need to allocate any memory for this, for the in-lab.
`String(const String& s)`	**Copy constructor**, construct a String object and copy the buffer inside String s into it. You can assume that the buffer in s is valid, so you do not need to do pointer or length checking on it.
`String& operator=` `(const String &s2)`	**Assignment Operator** Copy the string in s2 into your class, clobbering whatever is in there. You may use the C function **strcpy**.
`int length() const`	Compute the length of the string and return it. YOu may use the Cstring library function **strlen**.
`operator const char*()`	Returns a pointer to the internal string buffer. This is

	useful when interfacing to external code that does not recognize your String class, such as **cout**.
`bool operator==` `(const String& s2) const`	Compare the String in *this to s2. You may use the C string library function **strcmp**. **Return** true if they are equal, **false** otherwise.
`bool empty() const` `operator bool() const`	Returns true if the string is the empty string.
`String& operator+=` `(const String& s2)`	Appends (concatenates) string s2 to the end of *this, and returns *this. You may use the C string library function **strcat**.
`String& operator+=` `(char c)`	Appends c to the string in *this, and returns *this
`istream& operator>>` `(istream&, String& s)`	Input operator helper function. Reads characters into s until it encounters a newline '\n'. Do not insert the newline into the string.
`ostream& operator<<` `(ostream&, const String& s)`	Output operator helper function. Prints the string s to the ostream.

EXAMPLE

Run your program with the test code given in file w9_lab.cpp. The output should be the following.

```
Testing constructors
PASS, Constructors!
Testing bool operator
PASS, operator bool().
PASS: string length works
testing operator+  :Hello World
testing operator+= :Hello World
PASS Operators test
PASS Copy constructor test passes
PASS: string concatenation up to 64 bytes
Type in the following line
the quick brown fox jumped over the lazy dog
```

```
the quick brown fox jumped over the lazy dog
PASS: iostream input
testing output. Let the submit script confirm that the output matches your
input.
the quick brown fox jumped over the lazy dog
```

IN-LAB SUBMISSION

To test and demonstrate execution of your program use the same data as the output example above.

If not on matrix already, upload **String.h, String.cpp** and **w9_lab.cpp** to your matrix account. Compile and run your code and make sure everything works properly.

Then run the following script from your account: (replace profname.proflastname with your professors Seneca userid)

```
~profname.proflastname/submit 244_w9_lab <ENTER>
```

and follow the instructions.

Please note that a successful submission does not guarantee full credit for this workshop.

> If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

At-home

ASSUMPTIONS FOR AT-HOME

- Assume that you will be given input strings of any length, and your class must be able to handle them.
- Assume that there will always be enough memory to allocate for these strings, so you do not need to check for failed memory allocations.

Use a pointer for the class-internal string buffer, allocated with **new** to store the string. The class interface should remain the same as in-lab. In other words, there is no need to change the *function declarations* for each function. However...each *function defintion* will have to be checked and modified to make sure that it works with your new class design (i.e. using a buffer pointer instead of fixed arrays.)

From here on, **Capacity** refers to the size of the internal buffer. **Length** refers to the length of the actual data in the string. Your string should always have enough capacity to store whatever string the user puts into it, so any time you modify the internal buffer you should check that

$$Capacity \geq Length + 1.$$

Equation 1: Minimum Capacity of a String

For example, you have used strings before

```
char myString[100] = "Hello World";
```

myString has a capacity of 100, and a length of 11 characters+1 for the 00 terminating null character. The string below also has capacity =100 and length =11+1.

```
char* myString=  new char[100];
strcpy(myString, "Hello World");
```

The following new methods should be defined to help you implement the class.

Class method	**Description**
`String(int capacity)`	**Default constructor.** Initialize the string to "", but give it a capacity of "capacity". Assume a default value of capacity of 1.
`String(const char* p, int capacity)`	Construct a string from a C-style string. If the user specifies a capacity of less than you need to store the string (See Equation), use the higher value.
`String(const String& s)`	Copy constructor, construct a String from another String.
`int capacity()`	Returns the size of the internal buffer (aka the "capacity")
`void resize(int newCapacity)`	Resize the array to have a capacity given by newsize. Reallocate and grow the buffer if the new Size is greater than the existing capacity. Do not shrink the capacity to less than the length of the string + 1.

	Do this without losing any of the data in the buffer!

EXAMPLE

```
Testing constructors
PASS, Constructors!
Testing bool operator
PASS, operator bool().
PASS: string length works
testing operator+  :Hello World
testing operator+= :Hello World
PASS Operators test
PASS Copy constructor test passes
PASS: string concatenation up to 8Kbytes
Type in the following line
the quick brown fox jumped over the lazy dog
the quick brown fox jumped over the lazy dog
PASS: iostream input
testing output. Let the submit script confirm that the output matches your
input.
the quick brown fox jumped over the lazy dog
```

AT-HOME SUBMISSION

To test and demonstrate execution of your program use the same data as the output example above.

If not on matrix already, upload **String.h, String.cpp** and **w9_home.cpp** to your matrix account. Compile and run your code and make sure everything works properly.

Then run the following script from your account: (replace profname.proflastname with your professors Seneca userid)

```
~profname.proflastname/submit 244_w9_home <ENTER>
```

and follow the instructions.

Please note that a successful submission does not guarantee full credit for this workshop.

If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

At-home reflection

1. A string can have a capacity of N, but have a length less than that. You have used such strings in the past, e.g. char myString[100] = “hello world”; has a **size** of 100 but a length of 11.
   Is your class coded to allocate **exactly** L + 1 bytes of storage for any string, where L = string length and 1 is the storage for the 0 character? Can your class still work if you decided to be generous and give all your strings more capacity than they need?
2. Given how hard you found it to debug your string library, is it a good idea to always make your own libraries, or to use the standard one used by everybody that are tested and true?
3. When you use your **sict::String** class in the main function, do you have to set them to the safe empty state? Do you have to clean up after them at the end of the function? Can you add 2 strings together without worrying about whether data is lost or not? How would this have made some of your previous labs easier, if you could use sict::String instead of regular char arrays?