V

# CISC 3142
# Programming Paradigms in C++
# Lab #8
# A String Class and the Canonical Form

## How to Develop and Submit your Labs

### Lab 8 — A String Class

Create a `String` class, which — while having minimal functionality — illustrates the use of and need for the canonical form.

### Overview

Here is the `.h file` for the class (note the use of the `mystring` namespace again):

```
#ifndef MYSTRING_H
#define MYSTRING_H

#include

namespace mystring {

class string {
        friend std::ostream &operator <<(std::ostream &os, const string &s);
        friend string operator +(const string &s1, const string &s2);
public:
        string(const char *cs="");
        string(const string &s);
        ~string();
        string &operator =(const string &rhs);
        char &operator [](int index);
        string &operator +=(const string &s);
        int length() const;
        void clear();
private:
        char *cs;
};

}

#endif
```

I've also supplied a string_Exception class and an app for testing your class (it is also the test driver in Codelab). Finally, I captured the expected output — you can see it in `mystring_app.stdout`.

### Implementation Notes

- While we can use namespaces to control clashes with the `std::string` class, the file names prove a bit more problematic; there is actually a `string.h` already... it's the C library for C-style strings — which if you recall is then wrapped by the `cstring` C++ library header. As such, please name your files with a `my` prefix, i.e., `mystring.h`, `mystring.cpp` and `mystring_app.cpp`.
- The `string(const char *cs="")` constructor allows one to create a `String` from C-string (and "..." literals, which are of type `const char *` — i.e., C-strings).
- Operations on the `cs` buffer are performed using the C-string functions you wrote in lab 4.2.
    - Memory allocation involves making sure the `cs` data member (i.e., the pointer to the C-string buffer) is pointing to a sufficiently sized buffer.
        - For this implementation, we will use exact-sized buffer; i.e., enough elements in the char array to hold the characters of the C-string + the null terminator
        - This is relevant for the two constructors, the assignment operator and the += and + operators.
            - Using the `string(const char *)` constructor as an example:
                - when this constructor is called, the length of the argument is obtained using strlen and a buffer of the corresponding size is allocated (this can be done within the member initialization list)
                - the contents of the argument C-string is then copied to this new buffer using `strcpy` (this needs to be done in the body of the constructor; there is no way to work it into the member intialization list)

```
string::string(const char *cs) : cs(new char[strlen(cs)+1) {    // the +1 is for the null terminator
```

- Similar logic applies to the copy constructor, the assignment operator, and the `+=` operator (you should be coding the `+` operator using the `+=` operator as shown in class): in those three cases the source buffer (i.e., the C-string to be copied, assigned, or concatenated) will be the `cs` data member of another `String` object.

- Gets you to implement a class in the canonical form
- Gives you a taste of how a string class might be implemented.

## Code Provided for this Lab