# CS 202 - Computer Science II

## Project 7

**Due date (FIXED): Wednesday, 10/30/2019, 11:59 pm**

**Objectives:** The main objectives of this project are to test your ability to create and use dynamic memory, and to review your knowledge to manipulate classes, pointers and iostream to all extents.

**Description:**
For this project you will create your own **String** class. You may use **square bracket**-indexing, **pointers**, **references**, all **operators**, as well as the **`<string.h>`** or **`<cstring>`** library functions (however the `std::string` type is still not allowed).
The following header file extract gives the required specifications for the class:

```
//Necessary preprocessor #define(s)
...
//Necessary include(s)
//You can include <cstring> or <string.h>
...
//Class specification
class MyString{

  public:
    MyString();                                                //(1)
    MyString(const char * str);                                //(2)
    MyString(const MyString & other);                          //(3)
    ~MyString();                                               //(4)

    size_t size() const;                                       //(5)
    size_t length() const;                                     //(6)
    const char * c_str() const;                                //(7)

    bool operator== (const MyString & other) const;          //(8)
    MyString & operator= (const MyString & );               //(9)
    MyString operator+ (const MyString & other_myStr) const;   //(10)
    char & operator[] (size_t index);                      //(11a)
    const char & operator[] (size_t index) const;            //(11b)

friend ostream& operator<<(ostream& os, const MyString& myStr); //(12)

  private:
    void buffer_deallocate();                                    //(13)
    void buffer_allocate(size_t size);                           //(14)

    char * m_buffer;
    size_t m_size;
};
...
```

Specifications explained:

The **MyString** Class will contain the following **private** data members:

- **m_buffer,** a char-type pointer, pointing to the Dynamically Allocated data. *Note*: This is no longer a static array. Dynamic Memory management has to guarantee that it points to a properly allocated memory region, otherwise Segmentation Faults can occur in your program. Also, Dynamic Memory management has to guarantee that it is properly deallocated when appropriate, and deallocated-&-reallocated when its size has to change.
- **m_size**, a `size_t` member, denoting how many characters are currently allocated for m_buffer. Note that this has to be properly initialized and updated each time the dynamically allocated memory is changed.

, will have the following **private** helper methods**:**

- **(13) buffer_deallocate** – will deallocate the dynamic memory pointed to by m_buffer. *Note*: The m_size which keeps track of m_buffer has to be updated too.
- **(14) buffer_allocate** – will allocate the required `size_t size` number of `char` elements and point m_buffer to it. It also has to check whether there is an already allocated space for m_buffer, and properly deallocate it prior to reallocating the new memory required. *Note*: The m_size which keeps track of m_buffer has to be updated too. Also you should at least be checking whether the `new` expression used to allocate data succeded or failed (you can check if it evaluated to a `NULL` value). (*Hint*: You may also want to try implementing the "correct" way via exception handling for the dynamic memory allocation).

and will have the following **public** member functions:

- **(1) Default Constructor** – will instantiate a new object with no valid data. *Hint*: which member(s) need to be initialized in this case?
- **(2) Parametrized Constructor** – will instantiate a new object which will be initialized to hold a copy of the C-string **str** passed as a parameter. *Hint*: has to properly handle allocation, and to do this it will need to "examine" the input C-string to know how many items long the allocated space for m_buffer has to be.
- **(3) Copy Constructor** – will instantiate a new object which will be a separate copy of the data of the **other** object which is getting copied. *Hint*: Remember deep and shallow object copies.
- **(4) Destructor** – will destroy the instance of the object. *Hint*: Any dynamically allocated memory pointed-to by m_buffer has to be deallocated in here.

- **(5) size** will return a `size_t` type, the size of the currently allocated char buffer (number of elements of the container).
- **(6) length** will return a `size_t` type, the length of the actual string without counting the null-terminating character (same rationale as C-string length checking). *Hint*: The return value of this method and `size()` will of course be different.
- **(7) c_str** will return a pointer to a `char` array which will represent the C-string equivalent of the calling MyString object's data. This method has to be `const`-qualified. *Hint*: It has to be a `NULL`-terminated char array in order to be a valid C-string representation.
- **(8) operator==** will check if (or if not) the calling object represents the same string as another **rhs** MyString object, and return true (or false) respectively.
- **(9) operator=** will assign a new value to the calling object's string data, based on the data of

the **rhs** MyString object passed as a parameter. Returns a reference to the calling object to be used for cascading operator= as per standard practice. *Hint*: Think what needs to happen before allocating new memory for the new data to be held by the calling object.

- **(10) operator+** will concatenate the C-string equivalent data of the calling MyString object with the C-string equivalent data of the **rhs** MyString object, and use the resulting concatenated C-string to construct another MyString object and return it by-Value. Returns by-Value a MyString object. *Hint*: Think the order that these opearations need to happen, as now this method will need to construct a new MyString object internally, ensure that it holds the concatenated C-string data, and finally return it when it is done.
- **(11a) operator[]** will allow by-reference accessing of a specific character at index `size_t` **index** within the allocated **m_buffer** char array of a non-`const` qualified object. This allows to access the MyString data by reference and read/write at specific m_buffer locations. *Note*: Should not care if the index requested is more than the m_buffer size.
- **(11b) operator[] const** will allow by-reference accessing of a specific character at index `size_t` **index** within the allocated **m_buffer** char array of a `const` qualified object. This allows to access the const MyString data by reference and read at specific m_buffer locations. *Note*: Should not care if the index requested is more than the m_buffer size.

as well as a **friend** non-member function:

- **(12) operator<<** will output (to terminal or file depending on the type of ostream& os object passed as a parameter to it) the MyString data (the C-string representation held within m_buffer).

The MyString.h header file should be as per the specifications. The MyString.cpp source file you create will hold the required implementations. You are given a source file proj7.cpp which will be a test driver for your class:

```
//(1)
MyString ms_default;
//(2)
MyString ms_parametrized("MyString parametrized constructor!");
//(3)
MyString ms_copy(ms_parametrized);
//(4)
MyString * ms_Pt = new MyString("MyString to be deleted…");
delete ms_Pt;
ms_Pt = NULL;
//(5),(6)
MyString ms_size_length("Size and length test");
cout << ms_size_length.size() << endl;
cout << ms_size_length.length() << endl;
//(7)
MyString ms_toCstring("C-String equivalent successfully obtained!");
cout << ms_toCstring.c_str() << endl;
//(8)
MyString ms_same1("The same"), ms_same2("The same");
if (ms_same1==ms_same2)
  cout << "Same success" << endl;

MyString ms_different("The same (NOT)");
```

```
if (!(ms_same1==ms_different))
  cout << "Different success" << endl;
//(9)
MyString ms_assign("Before assignment");
ms_assign = MyString("After performing assignment");
//(10)
MyString ms_append1("The first part");
MyString ms_append2(" and the second");
MyString ms_concat = ms_append1+ ms_append2;
//(11)
MyString ms_access("Access successful (NOT)");
ms_access[17] = 0;
//12
cout << ms_access << endl;
```

**Do not forget to initialize pointers and/or set them to `NULL` appropriately where needed. Do not forget to perform allocation, deallocation, deallocation-&-reallocation of dynamic memory when needed! Memory accessing without proper allocation will cause Segmentation Faults. Forgetting to deallocate memory will cause Memory Leaks!**

**The completed project should have the following properties:**

- Written, compiled and tested using Linux.
- It must compile successfully on the department machines using Makefile(s), which will be invoking the g++ compiler.
- The code must be commented and indented properly.
  Header comments are required on all files and recommended for the rest of the program.
  Descriptions of functions commented properly.
- A one page (minimum) typed sheet documenting your code. This should include the overall purpose of the program, your design, problems (if any), and any changes you would make given more time.

**Turn in:** Compressed Header & Source files, Makefile(s), and project documentation.

**Submission Instructions:**

- You will submit your work via WebCampus
- The code file proj7.cpp is already provided and implements a test driver.
- If you have header file, name it proj7.h
- If you have class header and source files, name them as the respective class (MyString.h MyString.cpp) This source code structure is now mandatory.
- Compress your:
  1. Source code
  2. Makefile(s)
  3. Documentation

  Do not include executable
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  ([PA] stands for [ProjectAssignment], [#] is the Project number)
  Ex: PA7_Smith_John.zip

**Verify:** After you upload your .zip file, re-download it from WebCampus. Extract it, compile it and verify that it compiles and runs on the ECC systems.

- Code that does not compile will be heavily penalized –may even cost you your *entire* grade–. Executables that do not work 100% will receive partial grade points.
- It is better to hand in code that compiles and performs partial functionality, rather than broken code. You may use your Documentation file to mention what you could not get to work exactly as you wanted in the given timeframe of the Project.

**Late Submission:**
A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects will be accepted up to 24 hours late, with 20% penalty.