

Project 4 - MyString class with dynamic memory [60 points]

See also client program, data file, and sample output

Write a string class. To avoid conflicts with other similarly named classes, we will call our version MyString. This object is designed to make working with sequences of characters a little more convenient and less error-prone than handling raw c-strings, (although it will be implemented as a c-string behind the scenes). The MyString class will handle constructing strings, reading/printing, and accessing characters. In addition, the MyString object will have the ability to make a full deep-copy of itself when copied. Note: all MyStrings must always be stored in a dynamic char array that is exactly the correct size to store the string.

The C++ library has numerous functions for handling C-strings. These functions (such as strlen, strcpy, strcmp and cin.getline) perform various tests and manipulations, and require the `<cstring>` header file be included. As a refresher, now would be a good time to revisit C-strings, see [Gaddis section 10.1 - 10.6](#) and check the link here to reference [Table 10-3 Some of the C-String Functions in <cstring>](#) .

Since this is a multifile class ADT project, you must place your header file and implementation file in a namespace. Normally one would call a namespace something more likely to be unique, but for purposes of convenience we will call our **namespace cs_mystring** .

I have provided sample class function prototypes as a way of assisting you in constructing the MyStrings class specification structure. Although not necessary the sample function prototype parameter lists will provide identifier names that maybe helpful as you work on the function implementation details. The client code as written along with a sample program output will also provide a good resource to identify where the class member function was invoked and what is the expected outcome.

Your class must have only one private data member, a c-string implemented as a dynamic array such as char *str. In particular, you must not use a data member to keep track of the size or length of the MyString.

Phase 1: This is the first phase of a two phase programming project.

Here is a list of the operations this class must support:

- **A length member function (public)** that returns the number of characters in the string. Use strlen().

Here is a sample member function prototype for the MyString class specification ...

int length() const;

- **The Default and other Class Constructors (public):** Construction of a MyString from a const c-string. You should copy the string data, not just store a pointer to an argument passed to the constructor. Constructing a MyString with no arguments creates an empty MyString object (i.e. ""). A MyString object should be implemented efficiently (space-wise) which is to say you should not have a fixed-size buffer of chars, but instead allocate space for chars on an as-needed basis. Use strcpy().

Here are sample function prototypes of two constructor that should be included in the MyString class specification to correctly handle the char* str private data member (pointer to a dynamic array)

MyString(); // default constructor

MyString(const char* inString); // parameterized constructor

- **Class nonmember friend ostream operator function (public):** Printing a MyString to a stream using an overloaded << (insertion) operator, which should simply print out its characters. Use <<. Here is a sample nonmember friend ostream function prototype for the MyString class specification ...

friend std::ostream& operator << {std::ostream &out, const MyString printMe);

*Note: std::ostream object (namespace std) is needed as the MyString class specification is inside the namespace cs_mystring block

- **Class member overloaded square brackets [] operator (public):** MyString object should overload the square brackets [] operator to allow direct access to the individual characters of the string. This operation should range-check and assert if the index is out of bounds. You will write two versions of the [] operator, a const version that allows read access to the chars, and a non-const version that returns the client a reference to the char so they can change the value.

Here is are the sample member overloaded square brackets [] operator function prototype for the MyString class specification ...

Char operator[] (int index) const;

char& operator [] (inty index);

- **Class nonmember friend binary Boolean operator functions (public):** All six of the relational operators (<, <=, >, >=, ==, !=) should be overloaded and supported. They should be able to compare MyString objects to other MyStrings as well as MyStrings to c-strings. The ordering will be based on ASCII values. You can think of this as essentially alphabetical order; however, because of the way that ASCII values are defined, uppercase letters will always come before lowercase letters, and punctuation will make things even more complicated. Confused? You don't need to worry about any of this: just use the results of calling the strcmp() function. MyStrings or c-strings should be able to appear on either side of the comparison operator.

Here is a sample nonmember friend Boolean operator function prototype for the MyString class specification

Friend bool operator < (const MyString left0p, const MyString right0p);

Don't forget to include the "Big Three": Since the MyString class (ADT) contains a char pointer member variable to be used for a dynamic array you will discover that **there are 3 member functions that must be included in any class that uses dynamic memory**. These three functions listed below are commonly referred to as the **"big-3"**.

- **Include a destructor**
- **Overload the assignment operator**
- **Include a copy constructor**

Why overload the = assignment operator for dynamic arrays: When we do a member-wise assignments (which is the default in C++) between class objects such as myname = yourname of class objects that have pointers as data members we are doing what is called a "shallow copy". It is called a shallow copy because we are copying only the pointers, instead of also copying the variables that the pointers are pointing to. What we need to do is tell C++ to have the assignment operator perform a deep copy instead of a shallow copy. We do this by overloading the assignment operator. The function algorithm should also correctly process self-assignment of class objects such as name = name;

Why and when a Copy constructor is used: C++ provides a way for us to override C++'s default copy mechanism (which does a shallow copy). If we provide a constructor that takes a single classType argument, C++ will use that constructor to make a copy instead of using the default memberwise copy. This special constructor is called a copy constructor.

There may be some confusion at this point about when we need an assignment operator and when we need a copy constructor, since the two situations are essentially the same. The

difference is that the overloaded assignment operator is used exclusively for situations where the actual assignment operator is used in a client program. The copy constructor is used when C++ has to make a copy of an object in situations where the assignment operator is not being used.

There are three situations in which C++ uses the copy constructor to make a copy if one is available. The first is pass-by-value parameters. The second is the return statement. When a return statement is executed in C++, a copy of the return value is made and placed among the local variables of the calling function. The third situation is initialization. It is important to understand that C++ makes a distinction between assignment and initialization. When you initialize a variable at declaration, the assignment operator is not invoked.

The role of the destructor: What happens to memory that has been allocated on the heap when execution reaches the end of the function ? Recall that all local variables on the stack (the automatic variables) are deallocated at this time, so the local variables will be properly deallocated. But any class related dynamic variables that are pointing on the heap do not get automatically deallocated, and so they become inaccessible and we have a memory leak.

To fix this problem, C++ provides a function called a destructor that is called automatically when an object on the stack is deallocated (this normally happens when execution reaches the end of a function). We just need to write a destructor that will deallocate any memory on the heap that the object is pointing to. Destructors must always have the same name as the class, except that a tilde (~) precedes the name. Destructors, just like constructors, do not have return types. And destructors never have arguments.

Here are sample function prototypes of the "big-3" that should be included in the MyString class specification to correctly handle the char* str private data member (pointer to a dynamic array)

`~MyString(); // class destructor`

`MyString operator=(const MyString& right); // = operator used in client code`

`MyString(const MyString& copyMe); //copy constructor makes a deep copy`

As you work on the algorithms for the class member functions, keep in mind that you may use all of the c-string functionality provided by C++. This will include the `strlen()`, `strcmp()`, and `strcpy()` functions, along with the overloaded insertion operator for c-strings. These functions are all covered in detail in the text. When you use `strcpy()` treat it as a void function despite the fact that it has a return value. Do not use `strncpy()`, `strncat()`, or `strncmp()` since they are not implemented in all versions of C++. You may NOT use anything from the C++ string class!!

Phase 2: In this phase you'll be making the following refinements to the class that you wrote in Phase 1.

- **Class nonmember friend istream function (public):** Just like the >> operator that reads C-strings, your >> operator should skip any leading spaces and then read characters into the string up to the first whitespace character.

To help with character storage in a temporary non-dynamic array, I suggest to include the following const in your class public access area ...

Static const int MAX_INPUT_SIZE = 127;

For reasons of convenience, we will impose a limit of 127 on the number of characters this function will read. This is so you can temporarily read into a non-dynamic array and then copy what you need into your data member, which will be a dynamic array. Note that this does not mean that all MyStrings will always have a maximum of 127 characters. For example, you might get a MyString with more than 127 characters by using the MyString constructor or by concatenating two MyStrings.

Hint: Don't try to read character by character in a loop. Use the extraction operator to do the reading of the input into a non-dynamic array, then use strcpy() to copy it into your data member. Make sure to allocate the correct amount of memory.

Hint: if you use the extraction operator as suggested above, you will not have to skip leading whitespace, because the extraction operator does that for you. Don't also forget the last statement return in;

Here is a sample nonmember friend istream operator function prototype for the MyString class specification

friend std::istream& operator >> (std::istream& in, MyString& readMe);

*Note: std::istream object (namespace std) is needed as the MyString class specification is inside the namespace cs_mystring block. Make sure the MyString readMe is a reference parameter and does not have a const key word as this function is designed to read data from a source and write to private data in a MyString class object.

- **Class member void readline function (public):** The readline() function will allow the client programmer to specify the delimiting character (the character at which reading will stop). It should work just like the getline() function works for c-strings; that is, it should place everything up to but not including the delimiting character into the calling object, and it should also consume (and discard) the delimiting character. This will be a void function that will take two arguments, a stream and the delimiting

character. It should not skip leading spaces. The limit of 127 characters imposed on the >> function above also applies to this function.

Hint: Don't try to read character by character in a loop. Use the `in.getline()` function to do the reading of the input into a non-dynamic array, then use `strcpy()` to copy it into your data member.

- **Class nonmember friend binary arithmetic operator + function (public):**
Overload the + operator to do MyString concatenation. The operator must be able to handle either MyString objects or C-strings on either side of the operator. Be careful with the memory management here. You'll have to allocate enough memory to hold the new MyString. I suggest using `strcpy()` to get the left operand into the result MyString, and then `strcat()` to append the right operand. Both `strcpy()` and `strcat()` should be used as if they are void, even though they do have return values.

Here is a sample nonmember friend arithmetic operator function prototype for the Fraction class specification ...

friend MyString operator + (const MyString left0p, const MyString right0p);

- **Class member operator += function (public):** Overload the shorthand += to combine concatenation and assignment. Only MyStrings can be on the left-hand side of a += operation, but either MyStrings or C-strings may appear on the right side. If you pay close attention to what the += operator does perhaps take a peek at the algorithm used for the similar += operation in the Fraction class project, these may be the easiest points of the semester.
Here is a sample member arithmetic assignment operator function prototype for the MyString class specification ...

MyString operator += (const MyString right0p);

- **About program format, style and documentation [10 points]:** Don't forget to read the [class related documentation](#) section of the prior project module, where commenting guidelines for classes are described in detail.
(1) In particular directly above your class specification you should have a rather large multiline comment where every public member function (and friend function) prototype, however simple, must have a precondition (if there is one) and a postcondition listed in the header file.
(2) The only documentation required in your class implementation file is that it should include a multiline comment called a class invariant. In addition to that although not required you are welcome include member function descriptions, data flow

comments and pre/post conditions for each function. Only the most complex of your function definitions will need additional comment (and that would appear in the implementation file).

(3) Use meaningful identifiers, proper program format and style in both the class specification and implementation files

Unfortunately, Visual C++ will, under its default settings, report an error when you try to use `strcpy()` or `strcat()`, even though they are standard C++. You can prevent this by adding this line as the first line in your file:

`#pragma warning(disable:4996)`

You must place your header file and implementation file in a namespace. Normally one would call a namespace something more likely to be unique, but for purposes of convenience we will call our namespace "cs_mystring".

What to submit when Project#4 is completed as per specifications:

Upload the MyString class specification file: **`mystring.h`**

Upload the MyString class implementation file: **`mystring.cpp`**

Make sure to include a multiline comment at the end of the specification file that contains the output produced when your source code was run as a multifile project with the client program provided. Do not turn in a copy of the client program provided.

Assignments submitted with changes made to the client program (however small) will receive a score of 0.

Additional Requirements and Hints:

The name of your class must be "MyString". No variations will work.

Your class must be split into two files, a header file and an implementation file.

You must place your header file and implementation file in a namespace. Normally one would call a namespace something more likely to be unique, but for purposes of convenience we can call our namespace "cs_mystring". General information regarding this additional feature in your project is provided on the a [namespace information page](#).

Use exactly one data member, a char *pointer.

In some respects, you can model the syntax of your `operator>>()` function header after the `operator<<()` function syntax you have seen in class lecture sessions. Just use `istream` in the place of `ostream` (since we are doing input instead of output). Also, keep in mind that the right operand will NOT be `const` since the extraction operator's purpose is to modify the right operand.

Your project solution should have about 19 functions (including friend functions). All but four of them (istream >>, readFile, operator = and operator +=) should be no more than 4 lines long. I'm not saying yours has to be like this, but it shouldn't be way off. The four listed functions may have about 6 program statements in each. You may use any helpful portions in the algorithms examples (demo1 & demo2 zip folders) provided during the class Zoom lecture sessions 4/28/20 & 4/30/20 that provide a review of this project. Do not use as any resource from a website if it includes a myString class (or CString class).

Getting Started

Here are some suggestions for those of you who have trouble just figuring out where to start with Project 4. Remember to use iterative development. That means start with the smallest, simplest subset or section of the MyString class specification/implementation that you can work on and complete and make sure it works by commenting out sections of client code, keeping only the function calls in the client code that relate to what you just worked on. Once program output matches the sample output then start adding other updates to the MyString class specification and implementation one at a time (preferably the simple things first, if possible).

Start with everything in one file if you're not 100% comfortable with using three files: class declaration at the top, then member function definitions, then main. Start with just a default constructor and a stream insertion operator. Work incrementally through the bulleted list of member functions, testing with portions of the client code commented out and let me know if you get stuck along the way. I won't always provide the actual code but perhaps a few pointers that may be helpful.

If you are having trouble reading data files in your MyString class ADT multi-file project setup, please take a look at some hints about reading input files provided at the end of the prior class projects.

Keep me posted if you encounter file not found errors and we can try to resolve the issue together via CANVAS email. Hope the guidelines and specifications are helpful and let me know if you need some clarification along the way.

I am sure you will learn a bit more about Object Oriented Programming (OOP) as you put together the details for this multifile class implementation for - A new MyString class ADT with a dynamic memory allocation component that should work like a string. Enjoy this new programming challenge !

Submit Your Work

Name your source code files mystring.h and mystring.cpp. Execute the given client program and copy/paste the output into the bottom of the mystring.h file, making it into a comment. Use the Assignment Submission link to submit the two files. You don't need to submit the client file. When you submit your assignment there will be a text field in which you can add a note to me (called a "comment", but don't confuse it with a C++ comment). In this "comments" section of the submission page let me know whether the program works as required.

Keep in mind that if your code does not compile you will receive a 0.