Extra Credit: Fun with the `str` Class

75 points - Due Friday, May 11, 11:59 PM

Assignment:

Posted on the website is the **str** class: str.cpp and str.h. This is a very simple implementation of a string class that works much like the **myArray** class we implemented - but instead of encapsulating a dynamic array of ints, **str** encapsulates a dynamic array of **char**s - in other words, a string.

The **str** class isn't bad, but there's quite a lot that can be done to make it more useful. In this assignment, you'll add some of them, and end up with a class that could be useful in real-world coding. Download **str.cpp** and **str.h**, get a handle on how they work, and then get started on the project. Talk to me if you can't figure out how it works.

Here's the way this works:

Below is a longish list of features that you could add to the **str** class. Before you panic: **you do not have to implement all of them!** You'll need to implement everything in the "Required Stuff" section; that gives you 25 points.

The other 50 will come from the "Optional Stuff" section. You can implement as many or as few of these options as you want - you'll need 50 points worth of stuff for full credit, but since this is an extra-credit project, nothing beyond the 25 points of "Required Stuff" is necessary.

This project (hopefully) won't be that hard, but when working with classes and pointers, there can be all sorts of unpredictable gotchas. It would be a bad idea to wait until the last minute to start; leave yourself time to ask questions if you need to. If you run into trouble, take time to think through what's going on with the code! Write lots of comments to explain your code. Use references and const where appropriate - aim for a robust and efficient class.

With the exception of reverse and uppercase/lowercase, most of these functions can be implemented by using the C Standard Library functions declared in string.h.

What It's Worth:

This project is entirely extra credit. It's worth up to an **extra 5%** of your grade.

How to get the extra 5%:

In order to get **any** credit for this project, you **must** have turned in **all** of the previous projects. If I don't have a submission for one of your previous projects, then you will receive no credit for this one. I'll still accept submissions for any of the earlier projects - even if you get little or no credit on them, simply turning them in (in a reasonable state of completion) will allow you to get credit for this extra credit project.

Required Stuff:

Test Program (15 pts): This will be your main program, the part with the `main()` function that drives everything else. You'll need to write a sample program that demonstrates all the features that you've added to the `str` class. This needn't be anything fancy. You'll need to write this code anyway, to test the features as you add them, so save the tests you do and make sure the final program that you turn in incorporates all of them.

Destructor (5 pts): As it stands, `str` has a giant memory leak (it doesn't ever delete the data buffer!) Add a destructor that fixes this.

Constructor for char* (5 pts): Define a constructor that takes a **`char*`** as an input and constructs a `str` using that input.

Optional Stuff:

Constructor for int (10 pts): Define a constructor that takes an **`int`** as an input. (Use the function `itoa` for this; when calling `itoa`, make sure to use a buffer large enough to accommodate any integer that could be used as input.)

String Repeat Constructor (15 pts): Define a constructor that takes a string and an integer that defines how many times the string is to be repeated, so you can construct a `str` like this:

```
str test( “AB”, 5 );
```

This would construct a `str` with the contents “ABABABABAB”. You can either implement this as a separate constructor or you can combine it with the `char*` constructor by giving the repeat parameter a default value of 1.

Concatenation Operator (30 pts): Here's the prototype for an addition operator:

```
str str::operator+( const str& s );
```

Overload this function so that it concatenates the contents of the current object with the parameter `s`, and returns the result in a new `str`. You'll also need to implement `operator+=`, which has this prototype:

```
void str::operator+=( const str& s );
```

Do *not* copy and paste code from `operator+` when you're writing operator+=; instead have `operator+=` call `operator+` to do all the work.

<u>Array Access Operator</u> (10 pts): Since C strings are just an array of characters, you can access individual characters using square brackets (e.g., `string[3]`). Overload the `[]` operator to add this functionality to the `str` class. Here's the function prototype:

```
char& str::operator[]( int index );
```

Note that this returns a *reference* to the correct character, which allows you to use the `[]` syntax to set as well as get characters in the `str`. Make sure that index is within the bounds of the array - if the user tries to access an index that's less than zero or greater than the size of the array, return a reference to the first or last element instead.

<u>Reverse</u> (20 pts): Define a member function (that takes no parameters) which reverses the order of the characters in the string.

<u>Uppercase / Lowercase</u> (20 pts): Define two member functions: one that makes the contents of the string uppercase, and one that makes the contents lowercase. Any character that is not a letter should be ignored here.

<u>Clear</u> (5 pts): Write a member function that clears/deletes the contents of the string. How you handle this is up to you, but after your function is called, the string should appear to be empty.

<u>Integer Conversion Operator</u> (10 pts): Write a conversion operator that will allow C++ to automatically convert an instance of `str` to an `int`. You'll likely find the function `atoi` useful here. Not all strings can neatly be converted to integers, of course; by convention, those that can't give a result of zero (`atoi` does this bit for you).

<u>Substring Search</u> (15 pts): Write a member function that takes another `str` as a parameter and attempts to locate that substring within the current string. This function should return the position that the substring is located. For example: if `sample` is a `str` that contains the string "hello", then calling your function with a parameter of "ll" should return 2, as the substring "ll" starts at position 2 in the string "hello". If the substring doesn't exist within the main string, then the function should return -1. The function `strstr` will help you muchly, although you'll need to do a tiny bit of pointer arithmetic to convert the result into an integer.

<u>Comparison Operators</u> (15 pts): The function prototype for `operator==` is:

```
bool operator==( const str& s );
```

Use the `strcmp` function to implement this operator. Also implement the `<, <=, >, >=,` and `!=` operators. (This may seem like a lot, but they're not all that different from each other.)

<u>Copy-on-write</u> (45 pts): Currently, when we copy one instance of `str` into another (via the copy constructor or `operator=`), we create a new memory buffer and then copy the contents of the old buffer into the new one. This is all well and good, but if we don't plan on *changing* the new string, then we just wasted a lot of time copying that data, and we end up with two identical chunks of data hanging around in memory.

An alternative to this is a technique called copy-on-write. Using this technique, when duplicating a `str`, we copy only the pointer (just as the default C++ implementation of the copy constructor does!), so that the two `str` instances share the same memory buffer. The tricky bit is this: if any method in the class attempts to modify the contents of the `str`, *at that point* the buffer is duplicated, and *then* the modifications are made to the newly created buffer, leaving the original string unmodified in the old buffer. The advantage to this technique is that as copying instances of `str` becomes very fast and efficient as long as no changes are made to the copy. Even if changes *are* made, the overall performance of the `str` is no worse than the original implementation.

This is a somewhat advanced technique. To make it work, you'll need to do a decent-sized overhaul of the internal workings of the class, and you'll need a pretty in-depth understanding of the internal workings of the str class; consequently it's worth most of the optional points.

Something Else (?? pts): Got another idea? Ask me. I'll let you know how many points I think it's worth.

What to turn in: a directory containing...

- Your code, of course! You'll need to include your test program, as well as your copy of str.h and str.cpp.
- A README file containing any relevant info, such as your name and compilation instructions (include the exact command you used to compile your code). For this assignment the README should **say which options you chose to implement,** just so I don't have to try and figure it out. Again, please include in the README the approximate amount of time you spent on this assignment.

Non-specific Notes

- Refer back to the syllabus for hints about writing good C++ programs (or programs in any language): only work on small pieces of code at a time, and test them well before moving on. Compile lots, and fix the warnings and errors. Write lots of comments.
- **Be sure your code compiles on the CS department's Linux machines before turning it in**. Even if you write it elsewhere (on a different OS, a different IDE, etc), give it a quick compile and test on a lab machine before you turn it in. Remember, if your code does not compile, it will get (at most) 50% credit.
- You must submit your code using the department's electronic submit procedure. A link to instructions for doing this is on the class website. Our course number (for the purposes of the submit program) is **c109**.
- Just a reminder: the CLAS academic (dis)honesty policies will be enforced. You're allowed to discuss the assignment with others, but not to collaborate or share code. Nor are you allowed to find or use code off the Internet.