Notes 1/10

It’s customary to not use “using namespace std” in header files. This is because sometimes you will reuse headers many times, and tying in a namespace to it can cause problems if in your other cpp files you want to use a different namespace. Just declare the namespace at the top of each cpp file, not the header.

Remember you can never pass an array to a function – only a pointer to the first element of the array

Void f(const char []) == Void f(const char\*)

Lets make a new class of string with different functionality. We will use cstrings as the basis.

Class String  
{  
 public:  
 String(const char\* value);  
 String();  
 …  
 private:  
 char m\_text[100];  
 int m\_len;  
}

Notice this would work, but there is a problem – it can only be a max of 100 characters. Plus, if you only use a couple of the characters you are wasting a ton of space. Dynamic allocation would probably be a better choice, because you only use the exact amount of space that you need.

Perhaps instead of an *m\_len* member to keep track of the length of the string, we could put a zerobyte at the end of the character array to represent the end of it. Upside of this is getting rid of an unneeded integer, but the downside is that if you want to know the length you need to call a function to find out. Both ways are doable, it just depends on the project and usage. You could do both if you wanted to.

To choose, you could run performance measurements and make a decision as to which is the best choice for your program. Using both methods combined might make other functions faster but use more time every time you create a String.

Make sure to document any invariants in your class. For this one, taking the “both” option, m\_text must point to a dynamically allocated array with length m\_len + 1, and m\_len must have a value greater than or equal to zero. There also has to be a zerobyte at the end or the string will not satisfy the “both” conditions.

The empty string in this case could again be two different things – m\_text == pointer to a zero byte, and m\_len == 0, or m\_text == null and m\_len == 0. Advantages of the first way are that it is uniform with the other length strings, while the second way takes up less storage. Again we can choose which one we want. The first one would be much more easy to implement, while the second might be much more efficient. In a project, the expression “the needs of the many outweigh the needs of the one” comes to mind – you doing the extra work may make the entire project run much better when calling that function. However lets choose the first one for now because it is much easier.

Now we have to remember to switch m\_text from an array of chars to a pointer to a char, because we are going to dynamically allocate to it.

String::String(const char\* value)  
{  
 m\_len = strlen(value);  
 m\_text = value;  
}  
  
Now m\_text points to value, and m\_len has the integer length of the value. This is WRONG, however. What happens when the memory that used to hold *value* changes to something else? m\_text is still pointing to the same memory address, so it will change. This will completely fuck up the program.

Do this:

String::String(const char\* value)  
{  
 m\_len = strlen(value);  
 m\_text = new char[m\_len+1]; //add one so you have room for the zerobyte  
 strcpy(m\_text, value);   
}

Now we have successfully dynamically allocated a new string. What about if we call the function with no input value? Default constructor:

String::String()  
{  
 m\_len = 0;  
 m\_text = new char[1];  
 m\_text[0] = ‘\0’;  
}

However, we can combine these 2 bits of code because they are so similar. If we changed it to this:

String::String()  
{   
 m\_len = strlen(“”);  
 m\_text = new char[m\_len+1];  
 strcpy(m\_text, “”);  
}

**You are allowed, if you want to, to declare a default value for a function if the caller leaves out the arguments.** You do it like this:

Class String  
{  
 public:  
 String(const char\* value **= “”**); //in this case the default value of value is the empty string.  
 …  
 private:  
 char\* m\_text;  
 int m\_len;  
}

You can do this with any number of arguments – f(double a, double b, double c **= 10**) is valid. One restriction – if you call a function you can only leave off the trailing arguments. For example: f(double a = 10, double b = 10, double c = 20) If you leave off an a or a c, that’s fine, but not b unless you just call f(). It’s impossible to only declare the middle one.

Now we have to check for problems with String’s constructor. It will fail if passed a null pointer, so we should bulletproof it like so:

String::String(const char\* value) //the const means we are allowed to change what char points to, but not the characters pointed to.  
{  
 *if (value == NULL)  
 value = “”;*  
 m\_len = strlen(value);  
 …  
}  
  
Because this class uses dynamically allocated memory, we need to write a destructor for it.

String::~String()  
{  
 delete [] m\_text;  
 }

Void f(String u) //remember, u is passed by value – copied, then discarded at the end of f  
{  
}  
  
Since u is passed by value, some constructor must be called for the String class. This is called the copy constructor and it only triggers when a String is passed by value. Luckily, in C++, if you don’t declare a copy constructor for a class the compiler writes one for you. Basically it just copies each and every value of String into the new one. Be careful though – since String contains a pointer object, if you modify the copy you will modify the original! Also, since u has a destructor and points to the same value as the original, when you leave the function the original string will be destroyed!

So we need to write a copy constructor for this class. We want to create a new u that can be modified without changing the original values.   
  
Class String  
{  
 public:  
 String(const char\* value);  
 String(); **String(const String& other);**   
 ~String;  
 …  
 private:  
 char m\_text[100];  
 int m\_len;  
}

String::String(const String& other) //this function is called with String other, a constant reference to a String object.  
 : m\_len(other.m\_len)  
{  
 m\_text = new char[m\_len+1]; //m\_text needs a new array it can modify without changing other  
 strcpy(m\_text, other.m\_text);  
}

Now when we call f(String u) and pass a String by value, this constructor will call this function instead of a compiler generated copy function.

Sidenote – when writing copy constructors, you are allowed to talk about private functions of the class being copied. **Member functions of the String class can talk about private members of ANY String class.**

Remember, you only really need to make a copy constructor if the class contains some kind of resource – if you need to write a destructor, you will need to write a copy constructor almost all of the time.

Void f(String u) //calls the copy constructor.  
{  
 String t(“Wow”);  
 …  
 t = u; //doesn’t call the copy constructor! It isn’t being *constructed*, it’s being copied.  
 …  
}  
  
The *t = u*  term is a big problem, because it leaves a memory leak and leads to all the problems that not having a copy constructor had. We need to override the compiler again here and declare and *assignment operator* for the class.  
  
Class String  
{  
 Public:  
 … **String& operator=(const String& rhs);**  …  
}  
  
We are changing the assignment operator (=) to have a different meaning for the string class.  
  
String& String::operator=(const String& rhs)  
{  
 delete [] m\_text; //cleans up the left hand side of the = first.  
  
 m\_len = rhs.m\_len; //copies the righthand m\_len to the left  
 m\_text = new char[m\_len+1]; //makes a new array to hold the chars  
 strcpy(m\_text, rhs.m\_text); //copies the righthand letters to the left  
  
 return \*this;  
}  
  
This will work until you try to assign an object to itself – like *s = s.* This is called *aliasing* and can happen in a variety of ways, like *a[i] = a[j]* or *s1 = s2*. Aliasing is one of the most common sources of bugs, and you should check it whenever it is possible in a problem. Luckily the fix is almost always trivial.

String& String::operator=(const String& rhs)  
{  
 **if (this != &rhs)** //checks that the objects are not identical already. If so, no need to do anything  
 {  
 delete [] m\_text; //cleans up the left hand side of the = first.  
  
 m\_len = rhs.m\_len; //copies the righthand m\_len to the left  
 m\_text = new char[m\_len+1]; //makes a new array to hold the chars  
 strcpy(m\_text, rhs.m\_text); //copies the righthand letters to the left  
  
 return \*this;  
 }  
}  
  
There is, however, a much more modern and sleek way of changing the assignment operator.

String& String::operator=(const string& rhs)  
{  
 if (this != rhs)  
 {  
 String temp(rhs); //first create a temporary variable that is a copy of rhs  
 //swap the pointers between rhs and m\_text  
 //swap the ints between rhs and m\_text  
   
 }  
 }  
  
Usually people write a “swap” function that swaps between 2 strings that can be called in this situation.