## CS2303: Systems Programming Concepts.

## C-term, 2018.

## Programming Assignment 3

Assigned: Tuesday, January 30, 2018

Due: Monday, February 5, 2018, 8:00 pm.

Please do each part of each section in sequence. Be sure to read each part of the assignment before you start doing that part.

This assignment covers topics earlier in the course, plus these new topics: arrays, character strings, dynamically-allocated memory, structs, and pointers to use with them. It also tests your ability to write code which obeys a specification written by somebody else, such as the writers of the C standard library.

**Section A**

# **Part 1**

On your VM, download the file ctest.zip. Import it into Eclipse. To do this, use the File->Import command on the top-line menu. Then pick General->Existing Projects Into Workspace (not Archive File, that won’t work!). On the next screen, click on Archive File, then browse to ctest.zip. This is the best way to import a complete project which was previously exported by Eclipse. After you have brought it in, rename the project **PA3A\_*yourusername***.

Compile the code, and run the program ctest. Use the Eclipse debugger as appropriate.

Read the code and understand how it works.

Note the values of the pointers and how they change as the program runs. Several of them point to the same area of memory, presumably the stack, and others point to another area of memory, presumably the heap. You should find that the stack uses relatively low addresses, and the heap uses relatively high addresses.

Note that the value of a pointer (and any other address printed with the **%p** conversion specifier) is printed in hexadecimal, which is base-16. The important things to remember about hex are:

* It uses digits 0-9, then A-F. Hex digit A has decimal value 10; hex digit B has decimal value 11, etc.
* Right-to-left, the weights of the digits in a hex number are 1, 16, 256, etc.

At this point, the exact values of these addresses are not important. What is important is being able to recognize which addresses are bigger than others, which are consecutive, and how far apart some of them are.

**Part 2**

In this part, you are going to write your own versions of the strlen() function, which you will call mystrlen1() and mystrlen2(). [What’s the difference? Keep reading!] Both must use the exact same calling sequence (signature) as the standard function, but with different function names. Both must work exactly like the standard strlen(). Add them to mystring.c and mystring.h..

If you are unsure of how strlen() is supposed to work, consult the class notes and/or type “man strlen”.

For those unclear on the concept ☺, “It must work exactly like the standard strlen()” means ”it must work exactly like the standard strlen().” It does not mean “it can work however you want,” or “it can work the way you wish the standard strlen() worked,” or “I’m not really sure how it is supposed to work or how it actually does work.”

Use complete Doxygen header comments for this and all future functions! So for mystrlen1 and 2, the first thing you need to do is write the header comment. Carefully specify what the function does, the parameters, the return value, and any special conditions. Only then should you write the function.

Just to make sure you do this, I am instructing the course staff not to help anyone who tries to write or debug their function without writing the header comment first. Why? ~~Because I am being mean.~~ Because these functions are very hard to get right if you are not perfectly clear on exactly how they are supposed to operate in all circumstances. I do not want to waste your time or their time.

Start by writing mystrlen1(). In this version, step through the array using a subscript, for example: **p[i]**. Make sure the function compiles without error before moving to the next step. From now on, do this whenever you write a new function – be sure it compiles before trying to call it. This rule should apply even when you are writing your code top-down, i.e. by writing a stub function, calling it, then going back and filling it in.

Now un-comment the line in main() which prints the return value of mystrlen1(), and confirm that your function works properly

Then modify mystrdup1() to use your new function instead of the standard strlen(). Be sure to use proper documentation for all your code. Compile and run it, and make sure it works.

Now write mystrlen2(). This must also accomplish the same thing as strlen(). But in this version, access the character array using incrementing pointers instead of subscripts.

Note: By “incrementing pointers” I mean having a pointer and incrementing it using the ++ operator, like this: **p++**. I do not mean having a counter variable and adding that to the start of the array, like this: **(p + i)**. This is because the C compiler treats **\*(p + i)** exactly like **p[i]**, so you do not get the potential speed advantage from using pointers.

Then add code to main() to test mystrlen2().

Then modify mystrdup2() to use your mystrlen2(). Add code to test it.

## **Part 3**

Write and use your own version of strcpy(). Be sure that it works exactly like the original, including what it does with the terminator and what it returns. If you are unsure what that is, see the man page and/or the slides from class. Again, be sure your header comment clearly says what the function does, what the parameters mean, and what the return value is.

You only need to write one version of the function, but be sure to use incrementing pointers instead of subscripts. Also be sure not to wipe out your only copy of the pointer to the beginning of the string, you might need it later. Hint, hint!

Add code to main to test your function and prove that it works the same as the original.

Note: In each part, when you modify the main program in this assignment, be sure you do it by adding to the program. Be sure to leave in the code which demonstrates the previous functions.

When I say “demonstrate the code”, I mean it must be clear to us when we run your program what each function is doing.

## **Part 4**

Create your own versions of each of the other standard string functions used in the program: strcat() and strncat()

Again, be sure they work exactly like the originals. Let me repeat that: They must work exactly like the originals, including all their shortcomings, even if you can think of a better way. As before, write the header comment for each function first, before you start coding it. Be sure the comment specifies what the function does in various circumstances, such as how many characters it copies and what it does with the terminator.

It is vitally important that you understand how each of these functions is supposed to work. Pay special attention to what strncat() does in the following “corner cases”:

* Length of the string is shorter than n.
* Length of the string is exactly equal to n.
* Length of the string is greater than n.

Add code to test your functions and prove that they work the same as the originals.

## **Part 5**

Create your own versions of strncpy() and strndup(), and add code to the main program to demonstrate how they work. Once again, be sure they work exactly like the originals. Include printf() statements so we know what the program is demonstrating. Be sure to test the various “corner cases,” such as strings with lengths shorter than, equal to, and greater than the limit.

## **Part 6**

Submit your assignment using Canvas. Submit it as a single ZIP file, exported from Eclipse (be sure to clean first), containing the following files:

1. All the source files, including header files.
2. The makefile. Be sure it can be used to “make clean” and “make docs”.
3. The Doxyfile.
4. The .project and .cproject files.
5. A plain text file called readme.txt which contains your name, a brief description of what the program does, and instructions on how to compile, link, and run it, both from inside Eclipse and from the command line.

**Section B**

**Part 1**

Download the file structest.zip, import it into Eclipse, and rename the project **PA3B\_*yourusername***. Compile and run it.

## **Part 2**

The skeleton, header comment, and prototype already exist for a function that allocates a struct of type Employee and fills its fields with data passed in as parameters. Its prototype looks like this:

struct Employee\* makeEmployee(char\* name, int birth\_year, int start\_year);

Fill in the body of the function. Be sure that this function copies the parameter string into the array in the struct. Do not try copying the pointer to the string, since that string might go away.

Then fill in the header comment and the body for the function which prints one of these structs. Be sure you have correct header comments for both functions. Prove that these two functions work by modifying your main program so that it creates several structs and prints them.

## **Part 3**

Write a function which creates a struct of that type and fills it with random data. That is, it will generate a random string (filled with printable characters) and two random integers, and then call the makeEmployee() function from Part 2. Then it will return the newly-created struct.

You will have to create several “helper” functions, such as one which returns a random integer from 0 to some specified maximum, inclusive. How can you do that? Well, there is a function call rand() in the standard C library. You can find out about it using this command:

**man –s3 rand**

This will tell you how to use this function, including which header file or files you need to #include. Why do you need the “-s3”? It turns out that there is also a command with that name, and if you just ask the man program for rand, it will give you the man page for the command instead of the function. For more about this, look at the man page for the man program itself!

When you call rand(), it returns a pseudo-random number from 0 to RAND\_MAX, which is one of the standard symbolic constants. RAND\_MAX is a very large number, so you can use the modulo operator to generate a smaller number. For example, if you have a random integer r, and you calculate **r%n**, that will give you a random integer from 0 to n-1.

You will also need a helper function which generates a random string (of specified length) of printable characters. If you want to restrict the set of printable characters (e.g., only capital letters, letters and numbers, etc.) feel free. Remember, any time you have to do the same thing in more than one place, that is a candidate for a function.

Tip: Consider first creating a function which generates a single random character. Then you can call it from inside the function which creates the random string.

Another tip: In your code, avoid using the literal numeric codes for the characters. For example, use ‘A’ instead of 65. The compiler does not care, but your code will be much more readable.

Yet another tip: Use the “man ascii” command to see the numeric codes for the various characters. Observe the patterns. Note that, for example, the code for ‘B’ is the same as the code for ‘A’, plus one. The code for ‘C’ is the code for ‘A’, plus two. And so on.

Once you have this working, if you run your program several times you will notice that you always get the same sequence of random characters. This is a deliberate choice in the design of the C library. It is intended to make testing easier by always generating the same sequence, so the program always run the same. Once you have your program totally working, if you want, at the beginning of the program add a call to the function srand(), like this:

**srand(time(0));**

That changes what is called the *seed* of the pseudo-random sequence. In this case the seed is based on the time, so every time you run the program you will get a different seed. Warning: Whenever you use this technique, remember to call srand() only once, probably at the beginning of the program. Do not call it repeatedly, such as inside a loop, because that will keep resetting the seed, which you do not want to do.

**Part 4**

Then write a function which takes one parameter, a count. It should allocate an array which can hold that many pointers to the Employee struct, then fill each element of the array with a pointer to a newly-created struct filled with random data.

Finally, write a function that takes an array of pointers and a count, and prints out all the structs. Modify your main program so it demonstrates all these functions.

Remember: Nothing in C works by magic. If you need an array to hold a character string or an array, you need to allocate it before you can put data into it. If you allocate it with malloc() or calloc(), be sure to free() it when you no longer need it.

## **Part 5**

Write a function that duplicates an array of pointers to structs of this type. Be sure you have a way to tell the function the number of elements in the array. This is known as a “shallow copy”; it duplicates the pointers but not the things pointed to.

Modify the main program to demonstrate how it works.

## **Part 6**

Create a function which takes an array of pointers to the struct, and frees (de-allocates) all the structs pointed to. Do you have to make any assumptions about the pointers? Add code to demonstrate it. Note: Once you have freed a block of memory, there is no way to prove that it has actually been freed. So, for this step, being able to run this function without error is a sufficient demonstration.

## **Part 7 (Extra Credit)**

Create another version of the function which duplicates an array of pointers to the structs (i.e. the function from Part 5). In this one, duplicate the structs, then fill the array with pointers to these new structs. This is called a “deep copy.” Demonstrate.

**Part 8**

As you did for Section A, export and submit the project to Canvas.