Tutorial 1 – C Tutorial: Pointers, Strings, Exec (v0.10)

The purpose of this tutorial is to review C pointers and the use of the exec routines to support programming in C required for the assignments throughout TCSS422.

Complete this tutorial using your Ubuntu Virtual Machine, or another Linux system equipped with gcc.

Tutorial #1 is optional, and points will provided as a quiz grade.

To complete tutorial #1, submit written answers to questions 1-8 as a PDF file to Canvas. MS Word or Google Docs can be used to easily create a PDF file. Submissions with reasonably answers that demonstrate understanding of the core content of the tutorial will receive full credit.

1. Create skeleton C program

Start by creating a skeleton C program.

Several text editors are available in Linux to support writing C.
The Gnome Text Editor, or "gedit" for short, is a GUI based text editor similar to notepad in Windows.
Popular command-line based text editors, sometimes called TUI editors, include: vim, vi, pico, nano, and emacs.
Optionally, try using an Integrated Development Environment for C under Linux. Search the internet to find one you like.

The invoke these editors, open a terminal in Linux and type the name of the text editor:
pico exec.c
vi exec.x
gedit exec.c &
(vim is a variant of vi)

For "gedit", a GUI-based editor, it is recommended to "background" the process. Gedit will then run, and your command-line window will still be available for other work. The "&" runs a command in the background keeping the terminal available for use.
For text-based editors, the terminal will be used for the editor.

Now, enter the following code:

```
#include<stdio.h>
int main(int argc, char *argv[])
{
  printf("hello world \n");
  printf("number of args=%d \n",argc);
```

```
 for (int i=0;i<argc;i++)
 {
  printf("arg %d=%s\n",i,argv[i]);
 }
 return 0;
}
```

2. **Create a Makefile**

Now, let's create a "Makefile" to help compile the program:

gedit Makefile &

```
CC=gcc
CFLAGS=-pthread -I. -Wall

binaries=exec

all: $(binaries)

clean:
	$(RM) -f $(binaries) *.o
```

Save the makefile, and the exec.c source file.
From the command line, let's compile the program:

make

```
$make
gcc -pthread -I. -Wall    exec.c   -o exec
$
```

3. **Try out the skeleton C program**

The code has been compiled.
Now, let's run the program.
We will provide a few arguments to the command.

./exec myarg1 myarg2

```
hello world
number of args=3
arg 0=./exec
arg 1=myarg1
arg 2=myarg2
```

The program automatically captures the command line arguments and provides them as an array of null-terminated char arrays to our program.

4. **Creating a character pointer (char *)**

Now, let's show how we can create a char pointer, to point to any of the user provided arguments.

Before the first "printf" add the following line:

```
char * myarg;
```

After the closing "}" bracket of the for-loop, add the following lines:

```
myarg = argv[0];
printf("addr of myarg=%lu val of myarg=%s\n",myarg,myarg);
```

We will print out the address of our char pointer (char *), and also the string that is described there.

A string in C is really just a character pointer to an array of null-terminated characters. The pointer points to the first character. C identifies the end of the string by walking the character array one byte at a time until reaching the NULL character which signifies the end of the string.

Now use the makefile to recompile the program.

You'll see compiler warnings.
Printf does not like printing out "myarg" as a "%lu".
"%lu" is an unsigned long integer, and can be used to print out the (virtual) memory address of the 64-bit char pointer. Remember that all of our user program's addresses are virtual. They are not actual system memory addresses, but virtual addresses that the operating system, with the help of the CPU, translates to a physical address.

Now, ignoring the compile warnings, let's run the program.

$./exec myarg1 myarg2

We see that our pointer, points to some virtual memory location, and when we use this pointer, we can see the value of the string that is stored there.

5. **Checking how strings are terminated in C**

Now, let's prove that NULL characters, which have the ASCII value of 0, actually terminate the string.

Let's print out the NULL character that terminates the string to test this.
Our first arg is 6 characters long. We want to print the 7th character after the char * pointer's address. This should be a null. Now arrays in C start with index 0, so to print

the 7th character, we'll actually add 6 to the pointer.
To actually see the 7th character, when we add 6 to the pointer, we need to add parenthesizes so C treats myarg+6 as a pointer.

Add the following print statement to your code, and make and rerun the program:

```
printf("The NULL char that terminates my string=%d\n", *(myarg+6));
```

Question 1. What do you see? What you just did is not possible in Java. Why not?

Now, let's print out every character of the first string this way.
Using the clipboard, add the following lines to your program:

```
printf("myarg char 0=%d %c\n",*(myarg+0),*(myarg+0));
printf("myarg char 1=%d %c\n",*(myarg+1),*(myarg+1));
printf("myarg char 2=%d %c\n",*(myarg+2),*(myarg+2));
printf("myarg char 3=%d %c\n",*(myarg+3),*(myarg+3));
printf("myarg char 4=%d %c\n",*(myarg+4),*(myarg+4));
printf("myarg char 5=%d %c\n",*(myarg+5),*(myarg+5));
printf("myarg char 6=%d %c\n",*(myarg+6),*(myarg+6));
```

Make and rerun you program.

You should now be able to visualize how a string is stored.
It is a NULL-terminated character array !

So our variable char * argv[] is an array of NULL-terminated character arrays.
Another way of saying this is ***"an array of Strings"*** !

6. **Creating an array of strings in C**

Now, let's say we wanted to make our own *array of NULL-terminated character arrays... (strings)*

Let's do this using dynamic memory.
We will allocate memory space on the heap using malloc.

First, include the stdlib.h in your program. Add this statement at the top, with the other include statement.

```
#include <stdlib.h>
```

Now what's interesting here, is that an array of NULL-terminated character arrays actually has **NO CHARACTERS**!

What?, *no*, really??
Yes!!

There are ***no*** characters in an array of NULL-terminated character arrays.
(In C, a NULL-terminated character array is a ***String***…)

Let's create our own array of NULL-terminated character arrays. *(array of strings)*

We will hijack the Strings in argv[], and store them BACKWARDS in the array!
And we will do this all without using the *strcpy()* function...

Question 2. What is strcpy()? (check out the man page 'man strcpy')

In our array we will set up the following mapping

original	**new**
argv[0]=./exec	ourarray[0]=myarg2
argv[1]=myarg1	ourarray[1]=myarg1
argv[2]=myarg2	ourarray[2]=./exec

Do you see the switch?

Let's declare our new array (ourarray) of NULL-terminated character arrays, (e.g. our *array of Strings*). We will use argc to help us size our array as needed.

After our declaration of myarg, let's add a declaration for "ourarray":

```
char ** ourarray = malloc(sizeof(char *) * argc);
```

Then right before our for loop, add the following j counter variable:

```
int j=argc-1;
```

We will use j to store the (char *) of each string into ourarray in reverse order.

"argc" is the argument count. The value returned is actually 3 for our sample inputs of:

$./exec myarg1 myarg2

But since arrays start with a base index of zero in C, 3 is too many, and we need to subtract 1.

Now, inside the for loop, add the following lines:

```
  ourarray[j] = argv[i];
  j--;
```

At the end of our program, we would like to print out "ourarray" which describes things backwards.

Add the following two lines right before the return statement:

```
for (int i=0;i<argc;i++)
  printf("ourarray arg %d=%s\n",i,ourarray[i]);
```

Now go ahead and make and rerun the program to check out “ourarray”:

$./exec myarg1 myarg2

If you do not get the following output, check over your code ***carefully*** and correct any problems until you do:

```
ourarray arg 0=myarg2
ourarray arg 1=myarg1
ourarray arg 2=./exec
```

Wow, how about that. We’ve made our own string array in C and reversed argv’s strings! We didn’t even use strcpy(). But wait, do we really have our own ***copy*** of these strings? Let’s check. Try changing one character of a string in “ourarray”, and then print it out and argv as well.

Add the following lines of code at the end of your program:

```
*(ourarray[0]+1)='b';
for (int i=0;i<argc;i++)
  printf("ourarray[%d]=%s -- argv[%d]=%s\n",i,ourarray[i],i,argv[i]);
```

Question 3. Does creating a string array on the heap and assigning each of the strings in the array to point to existing strings give us our own unique copy of the strings?

What do you see?
Uh oh! Changing ourarray has also changed argv!
Why is that?

7. Copying the value instead of a reference into a String array

Let’s correct this problem.
To do this, we’ll need to make a copy of the strings.
We can use strcpy(). Another alternative in C, is to use sprintf(). Sprintf is a clever routine which let’s you “print” formatted output to a string! How nice...

Using the man page, determine which header file you need to include for strcpy() and include it.

Now if we look at the manpage for strcpy, we see it requires two string pointers (char *).

Function prototype for strcpy:
char *strcpy(char *dest, const char *src);

We have several potential source strings: argv[0], argv[1], argv[2]...

*(note that argv[0] and char * can be used interchangeably. "argv[0]" is the first character of a null terminated array of characters. When we say argv[0], C actually uses the address, so char * and argv[n] are interchangeable.)*

We have the source, but not the destination. We need to create a string on the heap!
For this use malloc again...
Inside your **first for loop**, right after the printf, declare a new string on the heap:

```
char * s = malloc(sizeof(char) * strlen(argv[i]));
```

We make the new string equal to the length of the old string argv[i].
We check the length of the string with strlen().

Now, use the strcpy() command.
Look up on the man page to determine how to copy argv[i] into the new string "s".

Now, instead of assigning ourarray[j] = argv[i], assign it to the new string "s":

```
ourarray[j] = s;
```

Notice that we recycle the char pointers each time we iterate in the for loop.

Make and rerun the program to check out the array now with copied strings.

Question 4. When we change one character of ourarray[0] to 'b', do we see this change in argv as well? Yes or no?

8. **Building a string array to use execvp()**

Next, we would like to build our own array of NULL-terminated character arrays *(array of strings)* to provide to execvp() so that we can invoke another program with execvp().

Let's start a new C program as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define MAXSTR 255
#define ARGCNT 5
```

```
int main(int argc, char *argv[])
{
  char cmd[MAXSTR];
  char arg1[MAXSTR];
  char arg2[MAXSTR];
  char file[MAXSTR];

  // Additional code goes here

  return 0;
}
```

This time we are including “unistd.h”. This header file includes support for the exec routines.

We declare 4 character arrays with a fixed length of 255 characters each.

Now we would like to write a short program which uses execvp() to invoke a command. Using the man page, we see that execvp() accepts a pointer to a string, and a pointer to an string array.

```
int execvp(const char *file, char *const argv[]);
```

9. **Reading input from the console (user)**

We need to acquire four strings from the user.
The first will be for the command. The next two strings will be for command arguments. And the final string is a file name to run the command against.

There are a variety of ways to read a string from the user.
Try searching the internet for “read a string from the console in C”.

Question 5. When searching the internet, what methods do you see described to read strings from the console (user)?

There are tradeoffs for the various methods.

We will use “fscanf”. This function scans a file stream to acquire input. We would like to read a string from the user. We will provide three arguments to fscanf. The first is the name of the file stream to read from. Since we will read from the console, this is called ***“stdin”***. Next we need to specify a format string to read. We are interested in reading a string. The format strings for the scanf family of functions are similar to those for printf. A string is specified using **“%s”**. Finally the third argument will be the character array (string) where we will store the input from the user.

So to read the first string in cmd, the command use the following code:

```
printf(“cmd->”);
fscanf(stdin, “%s”, cmd);
```

We include a printf statement immediately before fscanf() to describe what parameter the user is to provide. Add these lines to your program.
Now, recall that a string array in C, is really just an array of pointers to char pointers.
Each char pointer needs to point to the first character of a string (character array) that is NULL terminated.

Conveniently, fscanf() NULL terminates character arrays for us.
To prove this, add the following line after your first fscanf:

```
printf("char %lu=%d\n",strlen(cmd),cmd[strlen(cmd)]);
```

This line, prints the character at the index position of strlen() for a string.
So if a string as a length of 4 characters, let’s say “grep”, it will print cmd[4]. But since arrays start at INDEX 0 in C, cmd[4] is actually the 5th character of the string.

Try running your program.
Try typing a variety of words for the input.

Question 6. What is the ASCII integer value for the last character ALWAYS?

Next, add lines to read arg1, arg2, and file from the user.

10. **Building a string array using references to existing strings**

Next we need to build the string array to pass to execvp().

Our string array will contain four strings (cmd, arg1, arg2, file), and will need to be terminated with the NULL character. The NULL character is required. This is what tells execvp() when reading the string array, to stop processing command line arguments. Without the NULL character, execvp() would believe the array goes on forever!
This typically leads to a segmentation fault, as execvp() will read past allocated memory on the heap, and will read random uninitialized memory locations beyond the array’s storage.

Let’s allocate a string array on the heap.
We’ve defined a constant named ARGCNT for the maximum number of arguments.
We will create a string array to store ARGCNT char pointers.

To create an array of strings on the heap, create a double char pointer.

```
char ** args = malloc( . . . );
```

Inside of malloc, provide the size of a char pointer multipled by ARGCNT.

What is the name of the function that tells us the size of a variable? Use this function to determine the size of a char pointer (char *). Search on the internet if you don't know the name of this function.

Next, assign the pointers for each SLOT in the string array. Each pointer will point to a string for invoking our command.

Since args will point to an address on the heap, we will point the first pointer of the array to cmd as follows:

```
*(args + 0) = cmd;
```

Here the zero is superfluous (not required). We use it as a placeholder.

Question 7. How can we assign the second pointer in our character array?

Determine how to assign the other elements of the string array.

Hint, having the "+ 0" is nor required, but was done as a placeholder to suggest how to assign other elements.

To verify that you've successfully read in the strings, use the following code:

```
for (int i=0;i<ARGCNT;i++)
  printf("i=%d args[i]=%s\n",i,*(args + i));
```

This for loop, loops through each array position and printouts out what is assigned there. This will be the input we send to execvp().

Assign the following:
args pointer 1 to: arg1
args pointer 2 to: arg2
args pointer 3 to: file
args pointer 4 to: NULL

The last position in our string array needs to point to a NULL character.
Use either 0 or NULL as they mean the same thing.

11. **Invoking the execvp() function**

The last step of our tutorial is to invoke execvp() with the string array we've just built.

Looking at the man page for execvp(), we see that the first argument is to be a string which is the command to execute, and the second argument is a string array which includes the command in the first position, and a list of arguments. The string array is the NULL terminated.

Question 8. Using our args string array, how can we reference just the string of the command?

We should use an index value set to 0.

The second argument can simply be the string array we've just built:

```
int status = execvp(args[0], args);
printf("STATUS CODE=%d\n",status);
```

Add these lines to your program, and make and run it.

Use the following inputs:

$ ./execvp

```
cmd->grep
arg1->-c
arg2->the
file->execvp.c
```

This completes the tutorial on string, pointers, and execvp in C.