# Tutorial 1 – C Tutorial: Pointers, Strings, Exec (v0.11)

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 in TCSS422.

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

**Tutorial Submission**

Each question is worth 3 points, for a total of 24 points. Tutorial #1 is scored in the "Tutorials/Quizzes/In-class Activities" category (15%) of TCSS 422.

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 reasonable 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. VSCode for Linux from Microsoft is quite popular. Installation instructions are available here: https://code.visualstudio.com/docs/setup/linux

The invoke these editors, open a terminal in Linux and type the name of the text editor:

```
pico exec.c             # pico and nano are similar
vi exec.x               # vim is a variant of vi
gedit exec.c &          # & runs the GUI in the background w/o blking the shell
code exec.c             # for ms vs-code
```

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, 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, then 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, run the program.
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 your program.

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

Now, check out how to 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);
```

Next print out the address of your 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.

Compiler warnings should appear !
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 user program memory addresses are virtual. They are not actual physical memory addresses, but virtual addresses that the operating system, with the help of the CPU, translates to a physical address.

Now, while ignoring the compile warnings, run the program:

```
$./exec myarg1 myarg2
```

You should see that your pointer, points to some virtual memory location, and when using this pointer, you will see the value of the string that is stored there.

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

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

Try printing out the NULL character that terminates the string to test this.

The first arg is 6 characters long. The GOAL is to print the 7th character after the char * pointer's address. This should be a null. Arrays in C start with index 0. To print the 7th character, add 6 to the pointer. To actually see the 7th character, when adding 6 to the pointer, you'll 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, 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 !

The variable char * argv[] is an array of NULL-terminated character arrays.
Another way of saying this is, ***"it's an array of Strings"*** !

## 6. Creating an array of strings in C

Now, try making your own *array of NULL-terminated character arrays… (i.e. strings)*

Do this using dynamic memory.
First allocate memory space on the heap using malloc.

This requires including the stdlib.h header file in your program.
Add this statement at the top, with the other include statement.

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

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***…)

Create your own array of NULL-terminated character arrays. *(array of strings)*

You'll hijack the Strings in argv[], and store them BACKWARDS in the array!
And this can all be done without using the *strcpy()* function...

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

In your array set up the following mapping:

| **original** | **new** |
|---|---|
| argv[0]=./exec | newarray[0]=myarg2 |
| argv[1]=myarg1 | newarray[1]=myarg1 |
| argv[2]=myarg2 | newarray[2]=./exec |

Do you see the switch?

Declare your new array (newarray) of NULL-terminated character arrays, (e.g. our *array of Strings*).
Use argc to help determine the array size that is needed.

After your declaration of myarg, add a declaration for "newarray":

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

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

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

Use j to store the (char *) of each string into newarray 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. You will need to subtract 1.

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

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

At the end of your program, print out "newarray" which describes things backwards.

Add the following two lines right before the return statement:

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

Now go ahead and make and rerun the program to check out "newarray":

```
$./exec myarg1 myarg2
```

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

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

You've now made your own string array in C to reverse argv's strings!
All done without using strcpy(). But wait, do you really have a new ***copy*** of these strings?
Let's check. Try changing one character of a string in "newarray", and then print it out and argv as well.

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

```
*(newarray[0]+1)='b';
for (int i=0;i<argc;i++)
  printf("newarray[%d]=%s -- argv[%d]=%s\n",i,newarray[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 provide a new, unique copy for each of the strings? Yes or No

What do you see?
Changing newarray should also have changed argv!
Why is that?

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

To correct this problem and copy-by-value instead of by-reference, you will need to make a copy of the strings. Start by using strcpy(). Another alternative in C, is to use sprintf(). Sprintf is a clever routine that allows formatted output to be "print" to a string! It is quite useful…

Using the man pages (or google), determine which header file should be included to use the strcpy() function. Add this header !

Now look at the manpage for strcpy. Note that it requires two string pointers (char *).

**Function prototype for strcpy:**

```
char *strcpy(char *dest, const char *src);
```

There are 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 saying argv[0], C actually uses the address, so char * and argv[n] are interchangeable. )*

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

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

Make the new string equal to the length of the old string argv[i].
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 newarray[j] = argv[i], assign it to the new string "s":

```
    newarray[j] = s;
```

Notice that the char pointers are recycled each time you iterate in the for loop.

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

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

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

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

Start a new C program. Name the new file **"execvp.c"** Enter code 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 include “unistd.h”. This header file includes support for the exec routines.

Declare 4 character arrays with a fixed length of 255 characters each.
Now, using the starter code, 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)**

The program needs to acquire four strings from the user.
The first string will be the command to run. 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.

Try using “fscanf”. This function scans a file stream to acquire input. Use “fscanf” to read a string from the user. Provide three arguments to fscanf. The first is the name of the file stream to read from. To read from the console, specify ***“stdin”***. Next specify the format of data to read in string form (i.e. “%s”). “%s” is used when reading a string. The format strings provided to the scanf family of functions are similar to those for printf. Finally, the third argument is the character array (string) where you will store input read from the user.

To read the first string into the cmd variable, use the following code:

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

The printf statement immediately before fscanf() describes what parameter the user is to provide.
Include these user prompts in your program.
Now, recall that a string array in C, is 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 automatically.
To TEST 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.
If a string as a length of 4 characters, let’s say “text”, 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 ALWAYS the ASCII integer value for the last character?

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

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

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

Your 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.

Now, allocate a string array on the heap.
Define a constant named ARGCNT for the maximum number of arguments.
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 multiplied 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 to be used as an argument when invoking the specified command.

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

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

Here the zero is superfluous (not required). It can be used as a placeholder.

**Question 7**. How can the second pointer in the character array be assigned?

Determine how to assign the other elements of the string array.
Hint, having the "+ 0" is not 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, iterates through each array position and printouts out what is assigned there.
This will be the input that is sent 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 your 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 the tutorial is to invoke execvp() with the string array you have just built.

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

**Question 8.** Using your args string array, what is the C syntax to reference the string of the command to be run?

Use an index value of 0.

The second argument can simply be the string array that was 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.