

Argument Passing Through the Command Line

Objective

To pass information from the shell command line to a running program and to understand the operating system's roll during this interaction.

Description

Most of the utilities/commands in Linux are written in C. Many of these commands accept input(s) from the user, in the form of flags and arguments, and react differently to different inputs (e.g. gcc -o temp sample1.c). The OS also maintains a list of variables about the system running environment such as your home directory, terminal type, search path,... etc. The values are collectively known as the environment.

The OS reads the user input from the command line, creates standard structures and passes them, and the environment, to the command. If the programmer of this command can access these structures, by declaring arguments in the definition of the main program. A common convention uses **argc** and **argv**, **envp** as follows:

```
void main(int argc, char **argv, char **envp) {...}
```

where:

- **int argc**: argument count, which is the number of tokens typed on the command line.
- **char **argv**: argument vector, which is an array of these tokens.
- **Char **envp**: environment variables and their values.

Now, **argc**, **argv** and **envp** are local variables the programmer can read and use to direct the execution of the command.

Submission:

Submit one compressed file that contains the following programs:

0. Printing all environment variables passed to the command “MyEnvironment”

Your first deliverable is a C program (MyEnvironment.c) that prints all the environment variables of your system (see figure below)

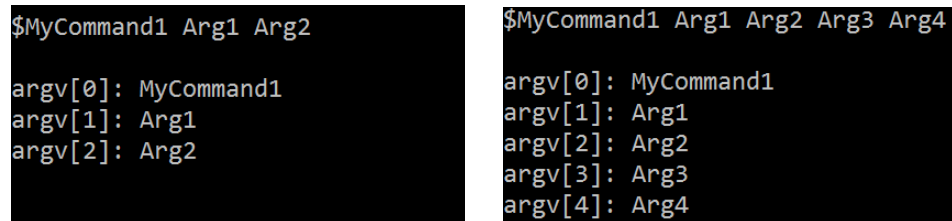
```
SHELL=/bin/bash
TERM=xterm-256color
USER=elnasan
NAME=DESKTOP-TAF0QD7
HOSTTYPE=x86_64
.....
PWD=/home/elnasan/Labs/Lab3
LANG=en_US.UTF-8
SHLVL=1
HOME=/home/elnasan
LOGNAME=elnasan
XDG_DATA_DIRS=/usr/local/share:/usr/share:/var/lib/napd/desktop
.....
OLDPWD=/home/elnasan/Labs
```

Figure 1 – Partial list of environment variables

1. Printing all arguments passed through the command “MyCommand1”

Your first deliverable is a C program (MyCommand1.c) that:

- prints all the arguments that are passed to it through the command line (see figure below)



```
$MyCommand1 Arg1 Arg2
argv[0]: MyCommand1
argv[1]: Arg1
argv[2]: Arg2

$MyCommand1 Arg1 Arg2 Arg3 Arg4
argv[0]: MyCommand1
argv[1]: Arg1
argv[2]: Arg2
argv[3]: Arg3
argv[4]: Arg4
```

Figure 1 – Sample runs of MyCommand1 with different inputs

2. Populating a fixed size argument array with the values passed through the command line (MyCommand2)

Your second deliverable is a C program (MyCommand2.c) that:

- Populates each entry in the fixed-size argument array with a default value (“defaultArg”).
- Updates the corresponding default values for the arguments with the values entered by the user.
- Prints the arguments
- Assume:
 - o argumentArray is defined as follows:


```
#define MAX_NUM_OF_ARGS 5
#define MAX_ARG_SIZE 256
char argumentArray[MAX_NUM_OF_ARGS][MAX_ARG_SIZE];
```

 The first argument in the argument array is the command itself
 (argumentArray[0] = argv[0]);

For example: \$MyCommand2 Arg1 Arg2

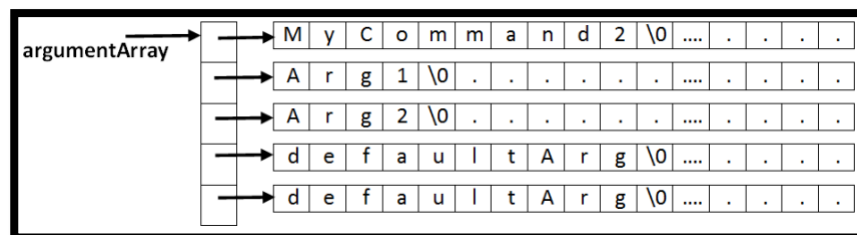


Figure 2- argumentArray populated after running: \$MyCommand2 Arg1 Arg2

- o Print an error message if the user enters more than 4 arguments (see figure below)

```

$MyCommand2 Arg1 Arg2

argumentArray[0]: MyCommand2
argumentArray[1]: Arg1
argumentArray[2]: Arg2
argumentArray[3]: defaultArg
argumentArray[4]: defaultArg

$MyCommand2 Arg1 Arg2 Arg3 Arg4

argumentArray[0]: MyCommand2
argumentArray[1]: Arg1
argumentArray[2]: Arg2
argumentArray[3]: Arg3
argumentArray[4]: Arg4

$MyCommand2 Arg1 Arg2 Arg3 Arg4 Arg5

Usage : MyCommand2 Arg1 Arg2 ...
Your arguments exceeded the maximum of arguments (4)

```

Figure 3- Output of MyCommand2

3. Modify your code in MyCommand2 to support variable size argument array to accommodate variable length arguments passed through the command line (MyCommand3)

Your third deliverable is a C program (MyCommand3.c) that:

- The argumentArray is defined as follows:
`char **argumentArray = NULL;`
- Allocate a place holder for each element in the argumentArray:
`argumentArray = (char**) malloc((argc+1) * sizeof(char*));`
- Allocate the right size for each argument (based on the user input):
`argumentArray[argIndex] = (char*)malloc((strlen(argv[argIndex]) + 1) * sizeof(char));`
- Copy the values typed by the user to the argumentArray (Figure 4).

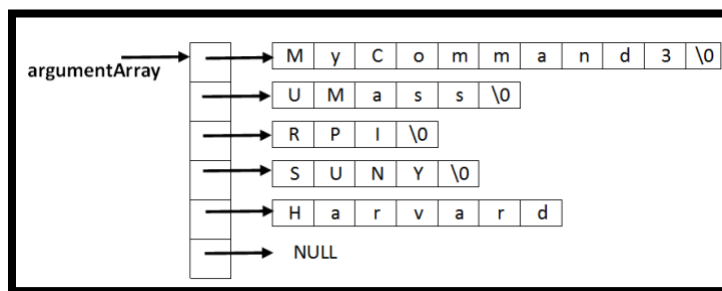


Figure 4- argumentArray populated after running:
`$MyCommand3 UMass RPI SUNY Harvard`

- Print the arguments (see example in Figure 5).

```
$MyCommand3 UMass RPI SUNY Harvard  
  
argumentArray[0]: MyCommand3  
argumentArray[1]: UMass  
argumentArray[2]: RPI  
argumentArray[3]: SUNY  
argumentArray[4]: Harvard  
  
$MyCommand3 UMass RPI SUNY Harvard Rochester MIT  
  
argumentArray[0]: MyCommand3  
argumentArray[1]: UMass  
argumentArray[2]: RPI  
argumentArray[3]: SUNY  
argumentArray[4]: Harvard  
argumentArray[5]: Rochester  
argumentArray[6]: MIT
```

Figure 5- Output of MyCommand3

Important: make sure to free the space you allocate before your program exits.