Prep C
Command line arguments and its use

Dr. Madhav Rao

July 14, 2015

Introduction

Command line is another way of providing arguments to your developed program. You normally use many options in linux commands such as

```
ls -al
```

Here ls is your executable which accepts *a* and *l* as command line arguments. In this activity, you are to remind yourself how to get data from the command line.

The Command Line

The command line is the line typed in a terminal window that runs a C program (or any other program). Here is a typical command line on a Linux system:

```
madhav@madhav:~/clab$ ./prog3
```

Everything up to and including the dollar sign is the system prompt. As with all prompts, it is used to signify that the system is waiting for input. The user of the system (me) has typed in the command:

```
./prog3
```

in response to the prompt. Suppose *prog3.c* is a file with the following code:

```
#include<stdio.h>

int main(int argc, char *argv[])
    {
    printf("command-line arguments:");
    printf("%s",argv[0]);
    }
```

In this case, the output of this compiled program would be:

```
command-line arguments: ./prog3
```

Command-line arguments

Any whitespace-delimited tokens including and following the program file name are stored in *argv* array. For example, suppose we modify the program prog3 by adding statements to print all the arguments such as argv[0], argv[1], argv[2], argv[3], argv[4], and argv[5]

```
./prog3 123 123.4 True hello, world
```

Then the output would be:

```
command-line arguments:
    "prog3" "123" "123.4" "True" "hello" "world"
```

From this result, we can see that all of the tokens are stored in *list of argv* and that they are stored as strings, regardless of whether they look like some other entity, such as integer, real number, or Boolean.

If we wish for "hello, world" to be a single token, we would need to enclose the tokens in quotes:

```
./prog3 123 123.4 True "hello, world"
```

In this case, the output is:

```
command-line arguments:
    './prog3', '123', '123.4', 'True', 'hello, world', (null)
```

There are certain characters that have special meaning to the system. A couple of these are '*' and ';'. To include these characters in a command-line argument, they need to be *escaped* by inserting a backslash prior to the character. Here is an example:

```
./prog3 \; \* \\
```

To insert a backslash, one escapes it with a backslash. The output from this command is:

```
command-line arguments:
    './prog3', ';', '*', '\'
```

Although it looks as if there are two backslashes in the last token, there is but a single backslash. C Program uses two backslashes to indicate a single backslash.

Counting the command line arguments

The number of command-line arguments (including the program file name) can be found by using the other argument passed in main function:

```
    printf("Length %d",argc)
```

and enter the following command at the system prompt:

```
./prog3 123 123.4 True hello world
```

we get this output:

```
Length 6
```

As expected, we see that there are six command-line arguments, including the program file name.

What command-line arguments are

The command line arguments are stored as strings. Therefore, you must use *atof*, *atoi* functions if you wish to use any of the command line arguments as numbers.

Task: Calculator

Create a directory named *cmdline* that hangs off your *clab* directory. Write a program named *calc.c* that takes three command-line arguments (not including the program file name) and performs a simple calculation. Here are some examples:

```
$ ./calc 2 + 3
5

$ ./calc 5 / 2
2.5

$ ./calc 5 - 6
-1

$ ./calc 4 \* 6
24
```

Remember, you need to place a backslash in front of the asterisk in the last example because an asterisk means something special to Linux.
HINT: To compare arguments with the symbols '+', '-', '', '/', use *argv[2] or argv[2][0]. The argv[2] will contain strings and to compare the starting character we need to specify *argv[2] or argv[2][0]*

Task: Looping over the numbers

Create a program (for.c) which will take three arguments (*intial final step*) as shown below and prints the numbers over step-size. Use For loop in this task. For example:

```
./for 3 10 2
```

should show the following output.

```
3 5 7 9
```

Task: While loop

Create a program (while.c) which does the same task as above, but using while loop.

Task: Arithmatic logic unit

Develop a program (alu.c) which does eight modes of arithmatic logic unit (ALU) operations: multiplexing, demultiplexing, encoding, decoding, direct-memory-access, random-memory-access, updating-cache, and recent-memory-search. All eight modes are controlled by eight bits. Design an ALU such that when user specifies an option in command line argument, the program should indicate the specific operation. The users can specify multiple operations by passing the bits in the command line argument. In the multiple operations scheme, we will assume that sequential operations are performed in the ALU. Use bitwise operations to display the operations selected by the user. Passing a Zero (0) option, should display entire menu of ALU as shown below:

```
./alu 0

Select menu
0x01: multiplexer
0x02: demultiplexer
0x04: encoding
0x08: decoding
0x10: dma
0x20: rma
0x40: updating-cache
0x80: recent-memory-search
```

Passing 1 option, should display "multiplexer" as shown below:

```
./alu 1
Multiplexer selected
```

Passing 2 option, should display ”Demultiplexer” as shown below:

```
./alu 2
Demultiplexer selected
```

Passing 3 option, should display two operations as shown below:

```
./alu 3
Demultiplexer selected
Multiplexer selected
```

Passing 255 option, should display all the operations.

Demonstration

Make sure your are in the *cmdline* directory. Then run the command ’ls’ to check the following source-code files and executables:

```
alu.c
alu
for.c
for
while.c
while
calc.c
calc
```

File input and output

Create a directory named *files* that hangs off your *activities* directory. Write a program named *calc.c* that reads three tokens from a file and performs a simple calculation. Here’s an example:

Suppose the file *simple-sum* contains the line:

```
2 + 3
```

Then, we would expect the following result when running your program:

```
$ ./calc simple-sum
5.00
```

Modify your *simple-sum* file to include four different operations as given below:

```
5 + 2
5 - 2
5 / 2
5 * 5
```

In fact, your program should behave exactly like the program you wrote in the previous activity except the tokens are placed in a file rather than on the command line. Now modify your program *calc.c* such that you include a loop to cover all four operations. You don’t need to use a backslash to protect the operators when reading from a file.

You must use the file operations: *fopen, fclose, fscanf* for this activity. Instead of fscanf, you can use fgetc as well. But make sure you ignore the space after reading the character. For getting operational character, you might have to use *fscanf* or *fgetc* twice as shown below.

```
ch = fgetc(in);
ch = fgetc(in);
```

or

```
fscanf(in,"%c",&ch);
fscanf(in,"%c",&ch);
```

Demonstration

Make sure your are in the *files* directory. Then run the command 'ls' to check all the files.