Computer Systems

Programming Assignments

Deliverables

All labs must be accompanied by a compressed (.zip) file with the following documents/files:

1- The zip file must be named: YourName_labX.zip

2- Source C code named YourName_labX.c

3- A README.txt file which contains the following lines & information:

NAME: Your Name
CSC-2X1 - Lab X
A brief explanation on the purpose of your lab
Compile & execution instructions
Answers to the lab questions, if applicable, must be typed.

Unidentifiable programs will not be graded.

4- Programs will be tested on the VM assigned for the course.

Program 1 Command Line Arguments

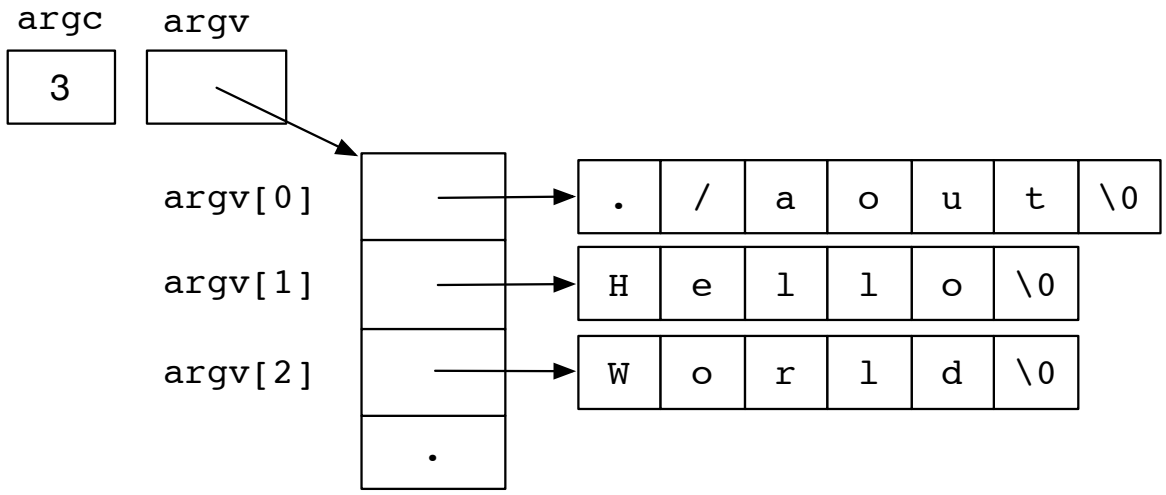
Command line arguments

When a C program is called, the arguments on the command line *are made available* to the main program as an argument count argc and an array of character strings argv containing the arguments. Manipulating these arguments is one of the most common uses of multiple levels of pointers ("pointer to pointer to ..."). By convention, argc is greater than zero; the first argument (in argv[0]) is the command name itself.

Command line data structure

The following figure shows the structure of the command line arguments when you type:

```
% ./a.out Hello World
```



Notice the each argument is terminated by the `\0` character. All strings must be terminated by a `\0`.

The following program prints the command line arguments.

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

int main ( int argc, char *argv[]) {  // could also use char ** argv
int i;
for ( i =0; i < argc; i++ )
 printf("\n",i, argv[i]);
printf("\n");
return 0;
}
```

When this program is executed we get the following output:

```
% cc arg.c
```

```
% ./a.out Hello World
argv[0] ./a.out
argv[1] Hello
argv[2] World
```

The following program will print the length of the first command line argument and its length.

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

int main ( int argc, char *argv[]) {  // could also use char ** argv
int i;

for (i =0;i<strlen(argv[0]);i++) {
  printf("%c\n",argv[0][i]);
}
printf("Length: %lu\n",strlen(argv[0]));
return 0;
}
```

When this program is run we get the following output:

```
% cc arg.c
$./a.out
.
/
a
.
o
u
t
Length: 7
```

Creating a Command Line Data Structure

A different situation arises when a program reads input from `stdin`. A typical example is the shell. The shell reads a command form `stdin`, decodes it, creates a command line data structure and finally executes the command. The structure of the shell can be viewed as follows:

```
initialize

while(!exit){
  read a command from stdin
  decode the arguments
  execute the command
}
```

The shell will read the command from the command line and create a command line data structure as explained above.

Program guidelines.

Write a C program that will read a line from `stdin`, decodes the input, and creates a command line data structure.
You must consider the following situations:

- An empty line. The user hit the return key without having typed any input.
- The program should terminate when the user types `exit` token in the command line.

A skeleton of your program will look like this:

```
initialize

while(!exit){
  prompt the user for data
  read a command from stdin
  decode and tokenize the argument list
  create the command line data structure
  print the command line data structure
}
```

A sample execution would look like this:

```
%csc: ./prog1
%enter data: ./a.out Hello World
argv[0] ./a.out
argv[1] Hello
argv[2] World
%enter data: this is my command
argv[0] this
argv[1] is
argv[2] my
argv[3] command
%enter data: exit
OK close shop and go home
%csc:
```