Command Line Arguments in C

CSE 2421

Recommended Reading: *C Primer Plus*, Chapter 11, Section on Command Line Arguments

How arguments are passed to main()

- We know that, when we call a function in a C program, arguments can be passed to the function.
- When we execute a C program from the command line, we can also pass arguments to main().
- Let’s see how this is done.

Passing arguments from the command line

- main() is invoked when we run a command on the command line to execute a C program. For example, for a program called myProg:

```
$myProg
```

- If we want to pass arguments, or parameters, say p1, p2, and p3, to main() in myProg, we enter them on the command line after the name of the program:

```
$myProg p1 p2 p3
```

Passing arguments from the command line cont.

- Any arguments passed from the command line will be placed in read-only memory.
- The arguments can be accessed by main() by declaring two parameters for main() (See the following slide).

Options for main() with arguments

- If we want to pass arguments to main() from the command line, here are two appropriate declarations of the parameters for main() inside the program:

  int main(*int argc, char **argv*) { … }

  int main(*int argc, char *argv[]*) { … }

- One of these approaches uses a pointer to a pointer to char, and the other uses an array of char pointers to pass the arguments.
- These two declarations are actually equivalent (they are just different ways of looking at the same thing – more explanation below).

Parameters on the command line

- The parameters given on the command line are passed to a C program using *two variables,* which have the following names, by convention:

  1. *argc* contains the count of the command line arguments. Think "argcount"

  2. *argv* contains pointers to the individual arguments as character strings (*argv[]* is an array of char *). Think "argvector".

-In the declaration of main(), the names of argc and argv may be ***any valid identifiers*** in C, but it is a common convention to use these names.

-**BUT WAIT**: There is no guarantee that the strings pointed to by the pointers in argv are stored in *contiguous locations in memory*, as they would be in a normal array, because argv is *an array of pointers* to these strings!

Example

- Again, suppose that, at the command line prompt, we enter:

```
$myProg p1 p2 p3
```

- This results in the following strings to which the elements of argv point, because argv[] is an array of char * (or a char **):

```
argv[0] points to “myProg\0” /*string with program name */
argv[1] points to “p1\0”
argv[2] points to “p2\0”
argv[3] points to “p3\0”
argv[4] is a NULL pointer
```

- Notice carefully that, because these strings are of type char *, they are stored in *read only memory*, and any attempt to write to them will result in a segmentation fault!
- Also notice that the strings are all terminated by a null byte (‘\0’).

More on argv

- Recall that any array may be viewed as a (constant) pointer to the first element of the array.
- For this reason, argv can be declared in one of two ways:

```
    int main(int argc, char *argv[]);
    int main(int argc, char **argv);
```

- Thus, argv may be viewed as an array of char *, or as char ** (a pointer to a pointer to char).

int main(int argc, char *argv[])

- After the arguments are passed to main(), we can access them, just as any other function parameters:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
        int i;
        printf("argc =\t%d\n", argc);
        for (i = 0; i < argc; i++)
                printf("argv[%i]\t= %s\n", i, argv[i]);
        return 0;
}
```

What is printed?

- The program on the previous slide prints:

```
argc =     4     /* argc is 4 – the number of parameters */
argv[0] = myProg
argv[1] = p1
argv[2] = p2
argv[3] = p3
```

How can we use these parameters?

- The name of the program, argv[0], may be useful when printing diagnostic or error messages. Usually, multiple processes (or programs) are running on the system, so it is useful to be able to identify which process caused an error.
- The values of the other parameters can be accessed and used just as any other function parameters.
- In Lab 4, you’ll need to use the value of two different parameters to determine the input filename and the output filename

More on argc and argv

- It is guaranteed that argc is non-negative, and that argv[argc] is **a null pointer**.
- By convention, the command line arguments specified by argc and argv include the program name as the string that is pointed to by argv[0].
- For example, if a user types the command “rm file”, the shell will initialize the rm process with argc = 2, argv[0] as “rm”, and argv[1] as “file”, and argv[2] as “0” [the value of a null pointer].
- Remember that the main() function is special. Every C program must define it exactly once.

Command Line Arguments – Example

```
/* sum.c – This program sums the integer values that correspond to the strings from argv[1] to
   argv[argc – 1] */
#include <stdio.h>
int main(int argc, char *argv[]){
   int i;
   int sum = 0;
   for (i = 1; i < argc; i++)
          sum = sum + ( atoi (argv[i]) );     /* atoi converts a string to an int */
   printf("The sum of the command line arguments after the first is %i\n", sum);
   return 0;
}
```

Command Line Arguments – Example cont.

- Now, after compiling, on the command line, suppose we enter:

```
% sum 100 50 25
```

- The program prints out:

```
The sum of the command line arguments after the first is 175
```

Command Line Arguments – Example cont.

- Now, after compiling, on the command line, suppose we enter:

```
% sum 100 50 25 10 > results
```

- The program prints out nothing but the file results contains:

```
The sum of the command line arguments after the first is 185
```

- The shell (bash) grabbed the > results redirection and processed it. It is not part of the command line arguments given to sum