Lab: Command-Line Arguments
CSC 161, “Imperative Problem Solving and Data Structures”
Department of Computer Science · Grinnell College
March 11, 2016

In this lab, we’ll learn another way to provide input to a C program: having the
program read arguments from the command line that launches it.

Pairs

- Tyler Williams and Mattori Birnbaum
- Ying Zhang and Ella Nicolson
- Colin Greenman and Tanner Tufto
- Dennis Chan and Saung Thuya
- Josh Lavin and Jong Hoon Bae
- Lex Martin and Erhaan Ahmad
- Anna Blindermann and Lilly Webster
- Eli Salm and Gemma Nash
- Yuyin Sun and Jae Hong Shin
- Faizaan Ali and Cory McCartan
- Sanjay Sudhir and Zachary Susag

Sophie Gaschott will fill in for anyone who is absent.

Exercises

Exercise 000: Write and test a C program called command-line-analysis that
recovers from argv the constituents of the command line that launched it and outputs them,
in order, one to a line, each labelled with its index in argv and enclosed in vertical bars.
A test run of your program might look like this:

```
$ command-line-analysis -b -k arrangement -s 32
0: |command-line-analysis|
1: |-b|
2: |-k|
3: |arrangement|
4: |-s|
5: |32|
```

Exercise 001: The shell can deal with a command-line argument that contains a space
character if the user “escapes” it by placing a backslash in front of the space to indicate
that it is not to be interpreted as a separator, or if the user encloses the entire argument
in double quotation marks. Using command-line-analysis, determine whether the shell
applies this same convention when setting up the argv array.

Exercise 010: If the wildcard character * is used as a command-line argument, the
shell “expands” it, replacing it with the names of all of the files in the current working
directory. Using command-line-analysis, determine whether the shell applies this same
convention when setting up the argv array.

The argc/argv facility provides all of the command-line arguments as strings, even
if they are numerals. Fortunately, the standard library stdlib provides seven func-
tions (strtod, strtof, strtold, strtol, strtoll, strtoul, and strtoull) for recov-
ering the numerical values of string numerals, returning values of types double, float,
long double, long, long long, unsigned long, and unsigned long long, respectively.
(If you want a value of some smaller signed integral type, call strtol and cast the result
to the desired type; similarly, if you want a value of some smaller unsigned integral type, call `strtoul` and cast the result.)

A typical call to convert a string `foo` to a `double` value looks like this:

```c
strtod(foo, NULL)
```

The second argument is used when the string `foo` might contain additional characters after the numeral. In that case, the second argument should be the address of a string variable, and `strtod` will store a pointer to the character immediately following the numeral into that string variable:

```c
char foo[] = "-33.9 or below";
char *bar;
double bound = strtod(foo, &bar);
```

After these statements, `bound` will have the value $-33.9$ and `bar` will point to the space character immediately following the digit 9 inside the string `foo`.

The functions that return integral values, such as `strtol`, all take a third argument, which must be an `int` value that is either 0 or in the range from 2 up to and including 36. If it is 0, the functions treat the numeral as if it were a numeric literal in a C program: a base-sixteen numeral if it begins with `0x` or `0X` (optionally preceded by a sign), or a base-eight numeral if it begins with 0, or a base-ten numeral otherwise. If the third argument is non-zero, it specifies the base of numeration, and letters of the alphabet are pressed into service as digits having values from ten up to, but not including, the specified base. (This is why the maximum base is 36 – once all the letters from `a` to `z` acquire digit values from ten to thirty-five, we run out of letters to serve as digits in higher bases.)

For example, the call

```c
strtoul(numer, NULL, 16)
```

will treat the string `numer` as a base-sixteen numeral (even without an explicit `0x` at the front) and return its numerical value as an `unsigned long`.

Exercise 011: Write and test a C program that takes two command-line arguments, both of which must be numerals for positive integers, and finds and outputs their greatest common divisor, using a function similar to the `gcd` function from the “Testing and Verification” lab, but adapted to take `unsigned long` values as arguments and to return an `unsigned long` result.

Exercise 100: Write and test a C program that takes one or more command-line arguments, each of which must be a numeral for a real number, and finds and outputs the sum of those real numbers.

Exercise 101: Extend the C program that you wrote for the preceding exercise so that, before starting into the addition loop, it checks (using `strcmp`) whether `argv[1]` is equal to the string "-p". If it is, then the extended program should compute and output the product of the real numbers expressed by the remaining command-line arguments rather than their sum.

Exercise 110: Adapt your solution to the first “robot siren” exercise (exercise 100 from the “Loops” lab) so that the user can specify on the command line the number of seconds that the siren should take to ascend one full octave.

Exercise 111: Write and test a C program that takes one or more command-line arguments, each of which should be an integer specifying the frequency (in hertz) of a tone that the Scribbler 2 robot can play, and directs the robot to play those tones, in the specified order, for an eighth of a second each. (You can get the effect of longer tones by
writing the same frequency as two or more successive command-line arguments.) Run the
program at least twice (without recompiling!), giving it different command-line arguments
to demonstrate that the command-line arguments control the tune.