Command Line Arguments

coursera.org/learn/interacting-system-managing-memory/supplement/Dmbkq/command-line-arguments

One form of input that our programs can take is *command line arguments*. As you have seen from many programs that you have used by now, when you run a command from the command prompt, you can specify arguments on the command line, by writing them after the name of the program. For example, when you run

gcc -o hello hello.c

you are passing three arguments to gcc (-o, hello, and hello.c). These arguments tell gcc—which is basically the implementation of a big algorithm to compile a C program—what specific instance of the problem we want it to solve (in this case, what C source code file to read, and where to put the resulting program).

We can write our programs so that they can examine their command line arguments. To do so, we must write **main**with a slightly different signature:

```
}
int main(int argc, char ** argv) {
  //...whatever code you want goes here...
```

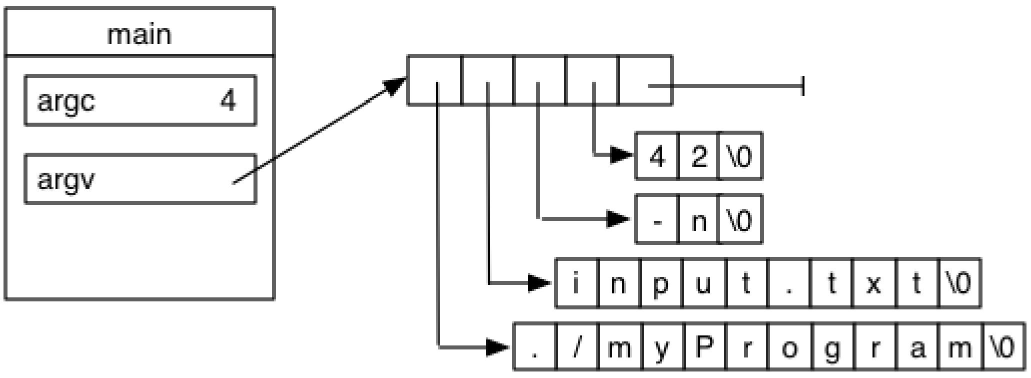
Here we see that **main**now takes two arguments. The first, **int argc** is the count of how many command line arguments were passed in (it stands for **arg**ument **c**ount). The second is an array of strings, which contains the arguments that were passed in (it stands for **arg**ument **v**ector). The 0th element of **argv** (that is, argv[0]) contains the name of the program as it was invoked on the command line (so if you wrote **./a.out**, then argv[0] is the sequence of characters "./a.out\0"). We can see this behavior in a toy example:

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

int main(int argc, char ** argv) {
  printf("Hello, my name is %s\n", argv[0]);
  return EXIT_SUCCESS;
}
```

This program prints out the name it was invoked with, and then exits.

argv[0] is not typically useful, but we must remember it is there if we access the other arguments so that we look for them at the correct indices. Whatever other arguments were passed to our program appear in argv[1] through argv[argc – 1]. The arguments appear in the array in the order that they were written on the command line (so the first one is in argv[1], the second is argv[2], and so on). The exact division of the text of the command line into discrete arguments is up to the command shell, but is typically done by separating the text at whitespace (note that the program could be invoked by something other than the shell, in which case, the invoking program can pass it whatever arguments it wants). The arguments are passed in as text, so if the program intends to interpret one of its arguments numerically, it must convert that text into a number.



The figure above depicts the frame at the start of main for a program which is run with the command line

```
./myProgram input.txt -n 42
```

We access the elements of **argv**as we would any other array of strings. We will note that for programs which expect a particular number of arguments, they should check **argc**first to make sure that the user actually supplied the right number of arguments before accessing the elements of **argv**—failure to do so can result in the program segfaulting when the user does not provide enough arguments.

The simplest access patterns to the command line arguments are iterating over the elements of **argv**(for programs that do the same thing to all arguments), extracting particular information from specific indices (for example, a program may always expect an input file name as argv[1] and an output file name as argv[2]), or a mix of the two (reading specific information from the first elements, then iterating over the rest).

✓

Completed