---
basato su:
http://crasseux.com/books/ctutorial/argc-and-argv.html

So far, all the programs we have written can be run with a simple command. For example, if we compile a C program to an executable called myprog, we can run it from within the same directory with the following command at the GNU/Linux command line:

```
./myprog
```

However, what if you want to pass information from the command line to the program you are running? Consider a more complex program like GCC. To compile the hypothetical myprog executable, we type something like the following at the command line:

```
gcc -o myprog myprog.c
```

The character strings -o, myprog, and myprog.c are all arguments to the gcc command. (Technically gcc is an argument as well, as we shall see.)
Arguments are separated by (one or more) blank spaces.

Command-line arguments are very useful. After all, C functions wouldn't be very useful if you couldn't ever pass arguments to them -- adding the ability to pass arguments to programs makes them that much more useful. In fact, all the arguments you pass on the command line end up as arguments to function "main" in your program.

Up until now, the skeletons we have used for our C programs have looked something like this:

```
int main()
{

  return 0;
}
```

From now on, our examples may look a bit more like this:

```
int main (int argc, char *argv[])
{
  return 0;
}
```

As you can see, main now has arguments. The name of the variable argc stands for "argument count"; argc contains the number of arguments passed to the program. The name of the variable argv stands for "argument vector". A vector is a one-

dimensional array, and argv is a one-dimensional array of strings: i.e. an array which has elements that are C-strings (a C-string is a character array where useful elements are followed by the '\0' character). Each string corresponds to one of the arguments that was passed to the program.

Array argv comprises a number of elements equal to argc: argv[0], argv[1], ..., argv[argc-1].

Another way to see argv is: argv is an array of pointers to char, each of which points to the first character of a C-string containing one of the arguments passed to the program via the command line. In fact, in C the identifier of a char array c (in this case, one of the elements of argv) corresponds to a pointer to the first element of the array (in this case, the first character of one of the command line arguments).
For example, the command line

```
  gcc -o myprog myprog.c
```

would result in the following values internal to GCC:

```
  argc
      4
  argv[0]
      "gcc"
  argv[1]
      "-o"
  argv[2]
      "myprog"
  argv[3]
      "myprog.c"
```

As you can see, the first argument (argv[0]) is the name by which the program was called, in this case gcc. Thus, there will always be at least one argument to a program (i.e., argv[0]), and argc will always be at least 1.

Note that a fully equivalent definition of function main is

```
  int main (int argc, char **argv)
```

In fact, both

```
  char *argv[]
```

and

```
  char **argv
```

correspond to an array whose elements are pointers to char (char*). In C, an array variable corresponds to a pointer to the first element of the array; therefore, argv (being an array of char*) has type char**.

The following program accepts any number of command-line arguments and prints them out:

```
  #include <stdio.h>

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

    printf ("This program was called with \"%s\".\n", argv[0]);

    if (argc > 1)
    {
      for (count = 1; count < argc; count++)
      {
        printf("argv[%d] = \"%s\"\n", count, argv[count]);
      }
    }
    else
    {
      printf("The command had no other arguments.\n");
    }

    return 0;
  }
```

If you name your executable fubar, and call it with the command

```
  ./fubar a b c
```

it will print out the following text:

```
This program was called with "./fubar".
argv[1] = "a"
argv[2] = "b"
argv[3] = "c"
```

If, instead, you call it with the command

```
  ./fubar

it will print out

This program was called with "./fubar".
The command had no other arguments.

Note that if you need to pass to a program an arguments which includes spaces, it should be enclosed in double quotes ("").
Otherwise the space would be interpreted as a separator between two different arguments. For example:

  ./myprog two args

has two arguments:

  argc = 2
  argv[1] = "two"
  argv[2] = "args"

while

  ./myprog "two args"

has only one:

  argc = 1
  argv[1] = "two args"
```