Processing Command-Line Arguments

In a UNIX environment, when you type a command such as

```
g++ main.cpp utils.cpp fileio.cpp
```

or

```
rm file1 file2 file3 file4
```

and press the `Enter` key, the shell program parses this command-line into words separated by whitespace characters. A ***word*** is usually any sequence of non-whitespace characters[1]. The first word on the command line, except in certain unusual commands, is the name of a program or a shell built-in command to be run. In the above examples, it is `g++` and `rm` respectively. The words that follow the program name are called ***command-line arguments***. In the first example above, the command-line arguments are `main.cpp`, `utils.cpp`, and `fileio.cpp`. In the second example, they are `file1`, `file2`, `file3`, and `file4`.

In UNIX and in other POSIX-compliant operating systems, the operating system arranges for the program name and the command-line arguments to be made available to the program itself via parameters to the `main()` function. Programs can ignore this information by writing the main function as

```
int main () { /* program here ... */ }
```

However, the C and C++ standards require compliant implementations of C and C++ to accept a `main()` with two parameters as follows:

```
int main ( int argc,  char * argv[]  ) { /* program here ... */ }
```

The first parameter is an integer specifying the number of words on the command-line, including the name of the program, so `argc` is always at least one. The second parameter is an array of C strings that stores all of the words from the command-line, including the name of the program, which is always in `argv[0]`. The command-line arguments, if they exist, are stored in `argv[1]`, ..., up to `argv[argc-1]`. If you have not seen the `char*` type, refer to the notes on C strings and pointers on this website.

Also note that there is nothing special about the names of two parameters `argc` and `argv`; they can be whatever names you want them to be. It is a convention to use the names `argc` and `argv`, although you will often find programs that use `ac` and `av` instead. You can name them `foo` and `bar` but that would be pretty bad programming style.

A simple C++ example that illustrates how a program can access the command-line arguments is below. This simple program does nothing more than display the name that the user typed to execute the program, followed by the command-line arguments that it received from the shell.

```
#include <iostream>
using namespace std;

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

[1] Certain symbols such as the shell redirection operators, semicolons, quotes, and so on, are not considered words in this sense.

```
    cout << argv[0] << ": ";
    for ( int i = 1; i < argc ; i++ ){
         cout << argv[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Whenever you write a program that expects command-line arguments you must check whether the expected number of arguments was provided by the user. Otherwise, the program will attempt to access locations in the array of arguments that do not exist.

For example, suppose that you write a program that expects the names of two files on the command-line, the first being the name of a file to open for reading and the second being the name of a file to open for writing. Suppose that the name you give to the program executable is `myprog`. Then proper use of `myprog` would be something like

```
myprog inputfile outputfile
```

There have to be at least three words on the command-line for your program to run properly. There might be more, but it can ignore those words. Therefore, the program should only run if the first parameter to `main()` is at least 3. The program must therefore begin with

```
int main ( int argc,  char * argv[]  )
{
    /* declarations here */

    if ( argc < 3 ) {
        /* handle the incorrect usage here */
        cerr << usage:  << argv[0] << " inputFileName   outputFileName \n";
        exit(1);
    }
    /* rest of program */
}
```

If the user did supply the correct number of arguments, then it is safe for the program to access the strings from the second parameter. The program might look something like

```
int main ( int argc,  char * argv[]  )
{
    ifstream fin;
    ofstream fout;

    if ( argc < 3 ) {
        /* handle the incorrect usage here */
        cerr << usage:  << argv[0] << " inputFileName   outputFileName \n";
        exit(1);
    }

    fin.open(argv[1]);
    if ( fin.fail() ) /* handle the error here */

    fout.open(argv[2]);
    if ( fout.fail() ) /* handle the error here */
    /* rest of program */
}
```

A Refinement

If the user types a command such as

```
../../proj1/testcode/myprog infile
```

and forgot to include the output file, the above code would produce output such as

```
usage:  ../../proj1/testcode/myprog inputFileName   outputFileName
```

If you do not want to display the entire path name of the program, but prefer that it only displays

```
usage:  myprog inputFileName   outputFileName
```

then you have to strip off the leading part of the `argv[0]` string so that the only thing left is what comes after the final '/' character. The C string library has a function that will make this easy, provided you are familiar with pointers.

You can use the `strrchr()` function, whose prototype is

```
char *strrchr(const char *source, int ch);
```

This function finds the last occurrence in the string `source` of the character `ch`. If `ch` is not in `source`, then it returns a `NULL` pointer. Therefore a strategy for displaying the characters of the program name after the final '/' is to check whether it has a slash, and if it does, display the part after it. The simplest way to do this is to take advantage of the fact that the program is allowed to modify the `argv[]` array. In particular, it can change what `argv[0]` points to. The following C++ program does just this.

```
#include <iostream>
#include <cstring>
using namespace std;

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

    forwardslashptr = strrchr( argv[0], '/');
    if ( forwardslashptr != NULL )
        /* argv[0] does contain the '/', so reset it so it points to
           the character just past the '/' character. Nothing needs
           to be done if it has no slash.
        */
        argv[0] = forwardslashptr+1;

    cout << argv[0] << ": ";
    for ( int i = 1; i < argc ; i++ ){
         cout << argv[i] << " ";
    }
    cout << endl;
    return 0;

}
```

This program uses "pointer arithmetic". The line

```
argv[0] = forwardslashptr+1;
```

sets the `argv[0]` string, which is, remember, a pointer to an array of characters, to the value obtained by adding one to the address stored in `forwardslashptr`. The compiler translates pointer addition to add however many bytes are needed by the type of thing that the pointer points to. In other words, if a pointer points to a `char` and a `char` takes up one byte, it adds 1. Therefore the above instruction causes `argv[0]` to point to the first character after the slash.