


## 20.4 — Command line arguments

---

 [learncpp.com/cpp-tutorial/command-line-arguments/](http://learncpp.com/cpp-tutorial/command-line-arguments/)

### The need for command line arguments

As you learned in lesson [0.5 -- Introduction to the compiler, linker, and libraries](#), when you compile and link your program, the output is an executable file. When a program is run, execution starts at the top of the function called `main()`. Up to this point, we've declared `main` like this:

```
int main()
```

Notice that this version of `main()` takes no parameters. However, many programs need some kind of input to work with. For example, let's say you were writing a program called Thumbnail that read in an image file, and then produced a thumbnail (a smaller version of the image). How would Thumbnail know which image to read and process? The user has to have some way of telling the program which file to open. To do this, you might take this approach:

```
// Program: Thumbnail
#include <iostream>
#include <string>

int main()
{
    std::cout << "Please enter an image filename to create a thumbnail for: ";
    std::string filename{};
    std::cin >> filename;

    // open image file
    // create thumbnail
    // output thumbnail
}
```

However, there is a potential problem with this approach. Every time the program is run, the program will wait for the user to enter input. This may not be a problem if you're manually running this program once from the command line. But it is problematic in other cases, such as when you want to run this program on many files, or have this program run by another program.

Let's look into these cases further.

Consider the case where you want to create thumbnails for all the image files in a given directory. How would you do that? You could run this program as many times as there are images in the directory, typing out each filename by hand. However, if there were hundreds

of images, this could take all day! A good solution here would be to write a program that loops through each filename in the directory, calling Thumbnail once for each file.

Now consider the case where you're running a website, and you want to have your website create a Thumbnail every time a user uploads an image to your website. This program isn't set up to accept input from the web, so how would the uploader enter a filename in this case? A good solution here would be to have your web server call Thumbnail automatically after upload.

In both of these cases, we really need a way for an external *program* to pass in the filename as input to our Thumbnail program when Thumbnail is launched, rather than having Thumbnail wait for the *user* to enter the filename after it has started.

**Command line arguments** are optional string arguments that are passed by the operating system to the program when it is launched. The program can then use them as input (or ignore them). Much like function parameters provide a way for a function to provide inputs to another function, command line arguments provide a way for people or programs to provide inputs to a *program*.

### Passing command line arguments

Executable programs can be run on the command line by invoking them by name. For example, to run the executable file "WordCount" that is located in the current directory of a Windows machine, you could type:

```
WordCount
```

The equivalent command line on a Unix-based OS would be:

```
./WordCount
```

In order to pass command line arguments to WordCount, we simply list the command line arguments after the executable name:

```
WordCount Myfile.txt
```

Now when WordCount is executed, Myfile.txt will be provided as a command line argument. A program can have multiple command line arguments, separated by spaces:

```
WordCount Myfile.txt Myotherfile.txt
```

If you are running your program from an IDE, the IDE should provide a way to enter command line arguments.

In Microsoft Visual Studio, right click on your project in the solution explorer, then choose properties. Open the “Configuration Properties” tree element, and choose “Debugging”. In the right pane, there is a line called “Command Arguments”. You can enter your command line arguments there for testing, and they will be automatically passed to your program when you run it.

In Code::Blocks, choose “Project -> Set program’s arguments”.

## Using command line arguments

Now that you know how to provide command line arguments to a program, the next step is to access them from within our C++ program. To do that, we use a different form of `main()` than we’ve seen before. This new form of `main()` takes two arguments (named `argc` and `argv` by convention) as follows:

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

You will sometimes also see it written as:

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

Even though these are treated identically, we prefer the first representation because it’s intuitively easier to understand.

**argc** is an integer parameter containing a count of the number of arguments passed to the program (think: `argc` = **argument count**). `argc` will always be at least 1, because the first argument is always the name of the program itself. Each command line argument the user provides will cause `argc` to increase by 1.

**argv** is where the actual argument values are stored (think: `argv` = **argument values**, though the proper name is “argument vectors”). Although the declaration of `argv` looks intimidating, `argv` is really just an array of char pointers (each of which points to a C-style string). The length of this array is `argc`.

Let’s write a short program named “MyArgs” to print the value of all the command line parameters:

```
// Program: MyArgs
#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "There are " << argc << " arguments:\n";

    // Loop through each argument and print its number and value
    for (int count{ 0 }; count < argc; ++count)
    {
        std::cout << count << ' ' << argv[count] << '\n';
    }

    return 0;
}
```

Now, when we invoke this program (MyArgs) with the command line arguments “Myfile.txt” and “100”, the output will be as follows:

```
There are 3 arguments:
0 C:\MyArgs
1 Myfile.txt
2 100
```

Argument 0 is the path and name of the current program being run. Argument 1 and 2 in this case are the two command line parameters we passed in.

## Dealing with numeric arguments

Command line arguments are always passed as strings, even if the value provided is numeric in nature. To use a command line argument as a number, you must convert it from a string to a number. Unfortunately, C++ makes this a little more difficult than it should be.

The C++ way to do this follows:

```

#include <iostream>
#include <sstream> // for std::stringstream
#include <string>

int main(int argc, char* argv[])
{
    if (argc <= 1)
    {
        // On some operating systems, argv[0] can end up as an empty string
        // instead of the program's name.
        // We'll conditionalize our response on whether argv[0] is empty or
        // not.
        if (argv[0])
            std::cout << "Usage: " << argv[0] << " <number>" << '\n';
        else
            std::cout << "Usage: <program name> <number>" << '\n';

        return 1;
    }

    std::stringstream convert{ argv[1] }; // set up a stringstream variable named
    convert, initialized with the input from argv[1]

    int myint{};
    if (!(convert >> myint)) // do the conversion
        myint = 0; // if conversion fails, set myint to a default value

    std::cout << "Got integer: " << myint << '\n';

    return 0;
}

```

When run with input “567”, this program prints:

```
Got integer: 567
```

`std::stringstream` works much like `std::cin`. In this case, we’re initializing it with the value of `argv[1]`, so that we can use operator `>>` to extract the value to an integer variable (the same as we would with `std::cin`).

We’ll talk more about `std::stringstream` in a future chapter.

## The OS parses command line arguments first

When you type something at the command line (or run your program from the IDE), it is the operating system’s responsibility to translate and route that request as appropriate. This not only involves running the executable, it also involves parsing any arguments to determine how they should be handled and passed to the application.

Generally, operating systems have special rules about how special characters like double quotes and backslashes are handled.

For example:

```
MyArgs Hello world!
```

prints:

```
There are 3 arguments:
0 C:\MyArgs
1 Hello
2 world!
```

Typically, strings passed in double quotes are considered to be part of the same string:

```
MyArgs "Hello world!"
```

prints:

```
There are 2 arguments:
0 C:\MyArgs
1 Hello world!
```

Most operating systems will allow you to include a literal double quote by backslashing the double quote:

```
MyArgs \"Hello world!\"
```

prints:

```
There are 3 arguments:
0 C:\MyArgs
1 "Hello
2 world!"
```

Other characters may also require backslashing or escaping depending on how your OS interprets them.

## Conclusion

Command line arguments provide a great way for users or other programs to pass input data into a program at startup. Consider making any input data that a program requires at startup to operate a command line parameter. If the command line isn't passed in, you can always detect that and ask the user for input. That way, your program can operate either way.