20.4 — Command line arguments

👤 **ALEX**[1]   🕐 **JUNE 25, 2024**

The need for command line arguments

As you learned in lesson 0.5 -- Introduction to the compiler, linker, and libraries (https://www.learncpp.com/cpp-tutorial/introduction-to-the-compiler-linker-and-libraries/)[2], 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:

```cpp
// 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:

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

You will sometimes also see it written as:

```
1   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 a C-style 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:

```
1   // Program: MyArgs
2   #include <iostream>
3
4   int main(int argc, char* argv[])
5   {
6       std::cout << "There are " << argc << " arguments:\n";
7
8       // Loop through each argument and print its number and value
9       for (int count{ 0 }; count < argc; ++count)
10      {
11          std::cout << count << ' ' << argv[count] << '\n';
12      }
13
14      return 0;
15  }
```

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.

Note that we cannot use a range-based for-loop to iterate through `argv`, since range-based for-loops don't work on decayed C-style arrays.

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:

```cpp
#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.

3

4

5

6

| B | U | URL | INLINE CODE | C++ CODE BLOCK | HELP! |

```
Leave a comment...
```

👤 Name*

@ Email* ?

Notify me about replies: 🔔

POST COMMENT

🐞 Find a mistake? Leave a comment above! ❓

👤 Avatars from https://gravatar.com/[8] are connected to your provided email address.

218 COMMENTS

Newest ▾

Brayan
🕐 June 23, 2025 3:15 pm PDT

So, I am follwing a SDL tutorial and they use the arg like this:

int main(int argc, char* args[])

What does argS stands for? Only argC and V are mentioned here.

👍 0   ↪ Reply

Brayan

April 20, 2025 3:54 am PDT

I dont understan the Using command line arguments part at the beggining. I dont see any arguments being passed so how does that work?

👍 0   ↪ Reply

Hari

Reply to Brayan [9]   May 19, 2025 5:17 am PDT

It's mentioned that they're optional
Command line arguments are **optional** string arguments that are passed by the operating system to the program when it is launched.

👍 0   ↪ Reply

rasp

February 14, 2025 3:08 am PST

FOR VISUAL STUDIO CODE USERS

go to the `.vscode` folder then open `launch.json` , look for the `args[]` attribute, and simply add your arguments in the [],

for example: `"args": ["arg1", "arg2", "arg3"]`

Then launch your program in Debug Mode (F5 by default) or go to Run>Start Debugging.

test code:

```
1  #include <iostream>
2
3  int main(int argc, char** argv) {
4    for (int i{0}; i < argc; i++) std::cout << i+1 << ". " << argv[i] << '\n';
5  }
```

✎ Last edited 4 months ago by rasp

👍 3   ↪ Reply

Invalid_UserName

February 14, 2025 3:08 am PST

took me a min to get how char** and char * [] are the same

✎ Last edited 4 months ago by Invalid_UserName

👍 4   ↪ Reply

EmtyC

December 9, 2024 2:45 pm PST

`char** argv` lel took me I while to understand that this is the decayed version of (a hypothetical char* argv[\\some number here]) and because the compiler treats argument char* argv[] as char** argv anyways

👍 0   ↪ Reply

Viktor

July 31, 2024 3:55 am PDT

Hi!
There are any downsides of using static_cast?
Thanks!

👍 0   ↪ Reply

> **Alex** `Author`
> 💬 Reply to Viktor [10]   July 31, 2024 4:13 pm PDT
>
> It's verbose (by design). Because the compiler will trust that you know what you're doing, you can perform static_casts that cause data loss (e.g. static casting double value 6.5 into an int).
>
> 👍 0   ↪ Reply

---

xx yy

October 31, 2023 3:11 am PDT

Could you not use `std::stoi()` or is that only for `std::string`?

👍 5   ↪ Reply

> **Alex** `Author`
> 💬 Reply to xx yy [11]   November 1, 2023 7:09 pm PDT
>
> `std::stoi` is for `std::string` arguments. Converting a C-style string into a std::string is comparatively expensive (but probably okay in this case).
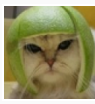>
> 👍 4   ↪ Reply
>
> > **Strain**
> > 💬 Reply to Alex [12]   April 15, 2024 11:43 am PDT
> >
> > What about `std::atoi`? This one works for `char*` s
> >
> > 👍 3   ↪ Reply

Alex `Author`

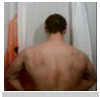💬 Reply to Strain [13] 🕐 April 16, 2024 9:02 pm PDT

Sure, you can use that one if you want.

👍 5   ↪ Reply

Timo

🕐 August 27, 2023 1:32 pm PDT

I don't get why we need to convert to a integer if you previously used an integer command line argument "100". This printed 100 without issue right or am I not getting something? I see that it is expecting char elements but it still printed 100.

✎ Last edited 1 year ago by Timo

👍 0   ↪ Reply

Alex `Author`

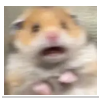💬 Reply to Timo [14] 🕐 August 29, 2023 1:46 pm PDT

Command line arguments are always strings. So if you enter a command line argument of `100`, it's interpreted as string `"100"`, not integer `100`.

👍 5   ↪ Reply

Zoltan

🕐 August 21, 2023 6:20 am PDT

In the for each loop lesson you mentioned that we should call these out, so wouldnt it make sense to replace the for loop with a for-each loop where you first count the number of arguments in the MyArgs app?

👍 0   ↪ Reply

Alex `Author`

💬 Reply to Zoltan [15] 🕐 August 23, 2023 6:14 pm PDT

I don't think so, because we're printing the index of each argument and a range-based for loop (for-each) doesn't give us that.

👍 0   ↪ Reply

Ajit Mali

💬 Reply to Alex [16] 🕐 June 19, 2024 3:54 am PDT

It's not about index printing

cannot build range expression with array function parameter 'argv' since parameter with array type 'char []' is treated as pointer type 'char *'

by the way, Thanks alex for these great lessons
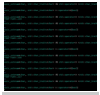
🗷 *Last edited 1 year ago by Ajit Mali*

👍 1   ➴ Reply

Zoltan

💬 Reply to Alex [16]   🕑 August 23, 2023 8:24 pm PDT

Hey yep, I think you're right, sorry about the false alarm :(

👍 0   ➴ Reply

learnccp lesson reviewer

🕑 July 13, 2023 7:00 pm PDT

W lesson

👍 1   ➴ Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-the-compiler-linker-and-libraries/
3. https://www.learncpp.com/cpp-tutorial/ellipsis-and-why-to-avoid-them/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/recursion/
6. https://www.learncpp.com/command-line-arguments/
7. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/command-line-arguments/#comment-609408
10. https://www.learncpp.com/cpp-tutorial/command-line-arguments/#comment-600329
11. https://www.learncpp.com/cpp-tutorial/command-line-arguments/#comment-589260
12. https://www.learncpp.com/cpp-tutorial/command-line-arguments/#comment-589324
13. https://www.learncpp.com/cpp-tutorial/command-line-arguments/#comment-595806
14. https://www.learncpp.Tom/cpp-tutorial/command-line-arguments/#comment-586228
15. https://www.learncpp.com/cpp-tutorial/command-line-arguments/#comment-585860
16. https://www.learncpp.com/cpp-tutorial/command-line-arguments/#comment-586016
17. https://g.ezoic.net/privacy/learncpp.com