C.1 — The end?

Congratulations! You made it all the way through the tutorials! Take a moment and give yourself a well-deserved (insert something you enjoy here).

Now, after breathing a long sigh of relief, you're probably asking the question, "What next?".

What next?

By this point, you should have a solid understanding of the core C++ language. This sets you up well to continue your learning journey into other adjacent areas. So if there's something you're really interested in learning about, now's a good time to see whether you have enough knowledge to jump into that.

However, for most users, I think there are a few natural next steps.

Data structures, algorithms, and design patterns

If you haven't already learned about these, this is my strongest recommendation.

A **data structure** is a collection of data and a well defined set of methods to access or manipulate that data. The most common data structure used in programming is the array, which holds a number of elements of the same type in sequential memory. You can manipulate the data inside an array by using array indexing to directly access (or modify) the elements inside the array. In the lessons, we also covered the stack data structure, which provide push, pop, and top functions to access the data on the stack.

An **algorithm** is a self-contained set of operations that typically manipulate or calculate outputs from the data in a data structure. For example, when you look through an array to find the median value, you're executing an algorithm. Binary search is an algorithm to determine if a given value exists in a sorted array. Sorting routines (such as selection sort and bubble sort) are algorithms that sort data sets.

Over the years, mathematicians and computer scientists have come up with a fairly standard set of reusable data structures and algorithms that are useful for constructing more complex programs. These all have various tradeoffs. For example, arrays are fast to access data and sort, but slow to add or remove elements. Linked lists, on the other hand, are slow to access data and sort, but very fast to add or remove elements (if you already know where those elements are).

Why does it matter? Let's use an analogy. If you were going to build a house, you could build all of your tools from scratch if you wanted. But it would take a long time, and you'd probably mess quite a few things up and have to start over (ever created a hammer? Me neither). Also, if you use the wrong tool for the job, your quality would suffer (try nailing in nails with a wrench).

More likely, you'd go to the local hardware store and buy a few tools: a hammer, a level, a carpenter's square, etc… and then read some internet tutorials on how to use them properly. These would vastly accelerate your house construction efforts.

Data structures and algorithms serve the same purpose in programming: they are tools that, if you know how to use them, can vastly accelerate how quickly you can get things done at quality.

The good news is that many of these data structures and algorithms have already been implemented in the standard library. You've already encountered some of these in the preceding tutorials: std::array, std::vector, std::stack, std::string, and std::sort, to name a few. Learning to use these effectively and appropriately is a great use of your time.

If you're short on time (or patience), learning how to use the most common data structures and algorithms is the minimum you should do. But if you have the inclination, try recreating those data structures yourself, from scratch. It's really good practice on writing reusable code, and will help you down the road when something you need isn't in the standard library. But then throw them out, and use the ones in the standard library. :)

Data structures and algorithms give us good tools for storing and manipulating data. However, there is one more tool that we can add to our toolkit that can help us write better programs. A **design pattern** is a reusable solution to a commonly occurring software design problem.

For example, we often need to traverse through the elements of some aggregate data structure (like an array or linked list), e.g. to find the largest or smallest value. But having to understand how an aggregate type is implemented to know how to traverse it adds complexity, especially if we have to write separate traversal code for each aggregate type ourselves. Iterators (which we covered earlier in this tutorial series) are a design pattern that provides an interface for traversing different aggregate types in a consistent way, and without having to know how those aggregate types are implemented. And code that is more consistent is easier to understand and less likely to have bugs.

Here's another example. Let's say you are writing an online game, and your program needs to maintain an open connection with a server to send and receive game state updates. Because opening new connections is expensive, you likely will want to ensure your program only has a single global connection object, and prevents the creation of additional server connections (so you do not accidentally create lots of connection objects and overload your

server). If you were to look through a reference of common design patterns, you'd discover that there is already a design pattern for ensuring that only a single, global instance of an object can be created (called a singleton). So instead of creating your own (possibly flawed) interface for this, you can implement a battle-tested singleton design pattern that other programmers will likely already be familiar with.

The C++ standard library

The bulk of the C++ standard library is data structures and algorithms. However, the standard library contains other things too, and another next step could be to explore those. Among other things, there are numerics (math) libraries, input/output routines, functions to handle localization and regionalization, regular expressions, threading, and file access. Every new release of C++ (which has been happening about every 3 years now) adds a batch of new functionality into the standard library. It isn't critical that you know how everything in there works, but it's worth at least being aware of what exists, so that if you happen upon the need for it, you can go learn more as needed. https://cppreference.com/w/cpp is my go-to reference for discovering what exists.

Graphical applications

In our tutorial series, we developed console applications, because they're easy, cross-platform, and don't require installing additional software. Unlike many modern programming languages, C++ does not come with functionality to create application windows, or to populate those windows with graphical elements or graphical user interface widgets (checkboxes, sliders, etc…). To do those things in C++, you'll either need to enlist the help of a 3rd party library, or learn to use the native APIs of your OS/platform.

Getting a graphical application up and running requires a few additional steps over console apps. First, you'll need to actually install the 3rd party library or OS SDK and connect it to your IDE, so you can compile it into your program. Most graphical libraries should come with instructions on how to do this for the most popular IDEs. Next, you need to instantiate an OS window, which requires calling certain function from the toolkit. Most, if not all, of the libraries should have sample programs that you can compile and dissect if you're not sure how to do something basic.

There are a lot of libraries out there, and which one you should use depends on your requirements (you'll have to do your own research to determine which one is right for you). Popular choices include Qt, WxWidgets, SDL, and SFML. If you want to do 3d graphics, all of these frameworks support OpenGL, and there are great OpenGL tutorials on the internet.

Graphical applications typically run differently than console applications. With a console application, the program starts executing at the top of main() and then runs sequentially, usually stopping only for user input. Graphical applications also start executing at the top of main(), typically spawn a window, populate it with graphics or widgets, and then go into an

infinite loop waiting for the user to interact with the window (via mouse click or keyboard). This infinite loop is called an event loop, and when a click or keypress happens, that event is routed to the function(s) that handle that type of event. This is called event handling. Once the event is handled, the event loop continues to run, waiting for the next bit of user input.

TCP/IP / Network programming (aka. the internets)

These days, it's pretty rare to find programs that don't connect to the internet, a back-end server/service, or leverage the cloud in some way. Any program that requires you to have an account and log in is connecting to a server and authenticating a user. Many programs connect to some service to check whether an update is available. Social applications maintain a persistent connection to a social infrastructure, to allow users to communicate with each other on demand. These are examples of networking.

Networking (broadly) is the concept of having your program connect to other programs, either on your machine, or on network-connected machines, to exchange information. Networking is a powerful tool -- in the past, if you wanted to change the behavior of your application, you had to release an application update. Now, with some good program design, you can simply update information on a server somewhere, and all instances of the program can leverage that change.

As with many things C++, there are libraries out there to help make your C++ programs network capable. The Asio C++ library is a commonly used one (there are two variants -- a standalone version, and a version that integrates with Boost, which is a library that provides a lot of different functions, much like the standard library).

Multithreading

All of the programs we've seen in this tutorial series run sequentially. One task is completed, then the next one starts. If a task gets stuck (e.g. you're asking the user for input and they haven't entered any yet), the whole program pauses. This is fine for simple academic programs, but not so great for actual applications. Imagine if your program couldn't handle the user clicking on something because it was busy drawing something on the screen, or if the whole program paused/froze when a network call was happening. The program would feel unresponsive.

Fortunately, a method exists to allow programs to execute multiple tasks at the same time. This is called threading. Much like how (most of) you can walk and chew bubble gum at the same time, threading allows a program to "split" its attention and do multiple things in parallel.

For example, some graphical applications (such as web browsers) put the rendering (drawing graphics) portions of the applications on a separate thread, so that updating the screen doesn't block other things (like accepting user input) while the drawing is happening.

Network calls are often done on separate threads, so that if the network call takes a while to resolve, the application doesn't grind to a halt while it's waiting.

Threading is powerful, but it introduces additional complexity, and a lot of room for additional errors. Therefore, I wouldn't recommend starting here -- but it is a good area to learn about eventually, especially if you want to do complex graphical applications or network programming.

Improve your fundamentals

Another option is to spend time improving your understanding of best practices. For this, I highly recommend having a read-through of the CPP Core Guidelines, with an optional delving into the GSL library.

Keep practicing!

The best way to get better at programming is to do more programming!

Developing a larger project (e.g. 1000+ lines of code) can be fun and challenging. Try implementing a simple game or simulation of some kind! Card games are often a good choice, as they tend to have fairly straightforward rules, and can be implemented using console output. One suggestion: Crazy Eights.

Now would also be a good time to join a website that offers various programming exercises to test your skills. We highly recommend Codewars, as it is free and has a ton of creative programming challenges!

Don't stop when you've got something working. Spend some time refining your code (e.g. refactoring, removing redundancy). Measure the speed of your solution (see 18.4 -- Timing your code) and then see if you can find ways to optimize your code so it performs faster!

A good bye!

At this point, I'd like to take a moment to thank you for stopping by and reading this tutorial series. I hope you enjoyed your time here and have found this site useful. Special thanks to those of you who have helped keep this website a free resource available to everyone by viewing ads that interest you. Please drop by again!

Good luck (and skill) in your future endeavors, and happy programming! And remember, old programmers never die -- they just go out of scope.

-Alex

PS: If you have any feedback or other suggestions for things to explore next, please mention them in the comment section below.