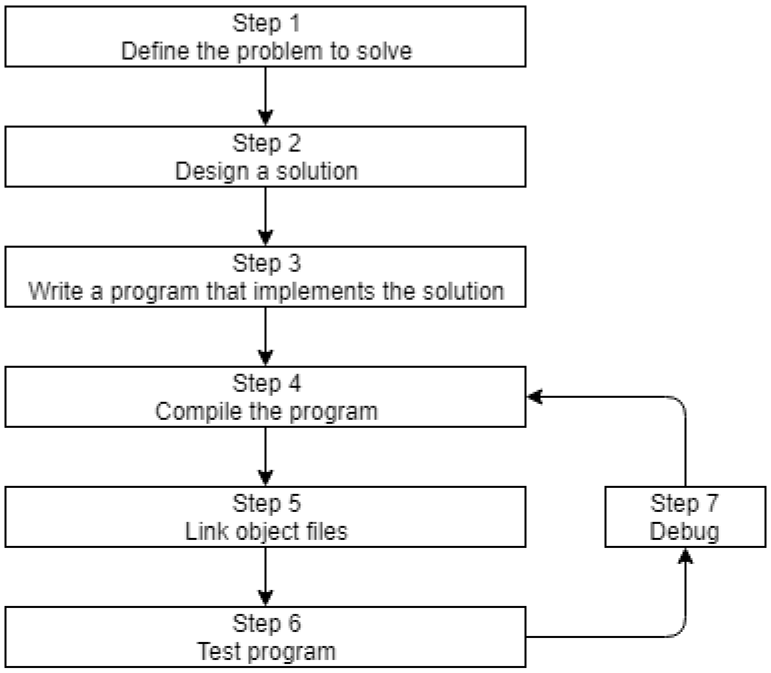
0.5 — Introduction to the compiler, linker, and libraries

learncpp.com/cpp-tutorial/introduction-to-the-compiler-linker-and-libraries/

Continuing our discussion of this diagram from the previous lesson (0.4 -- Introduction to C++ development):



Let's discuss steps 4-7.

Step 4: Compiling your source code

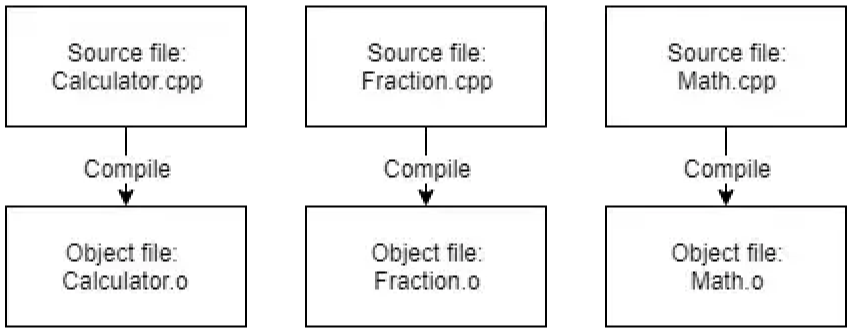
In order to compile C++ source code files, we use a C++ compiler. The C++ compiler sequentially goes through each source code (.cpp) file in your program and does two important tasks:

First, the compiler checks your C++ code to make sure it follows the rules of the C++ language. If it does not, the compiler will give you an error (and the corresponding line number) to help pinpoint what needs fixing. The compilation process will also be aborted until the error is fixed.

Second, the compiler translates your C++ code into machine language instructions. These instructions are stored in an intermediate file called an **object file**. The object file also contains metadata that is required or useful in subsequent steps.

Object files are typically named *name.o* or *name.obj*, where *name* is the same name as the .cpp file it was produced from.

For example, if your program had 3 .cpp files, the compiler would generate 3 object files:



C++ compilers are available for many different operating systems. We will discuss installing a compiler shortly, so there is no need to do so now.

Step 5: Linking object files and libraries

After the compiler has successfully finished, another program called the **linker** kicks in. The linker's job is to combine all of the object files and produce the desired output file (typically an executable file). This process is called **linking**.

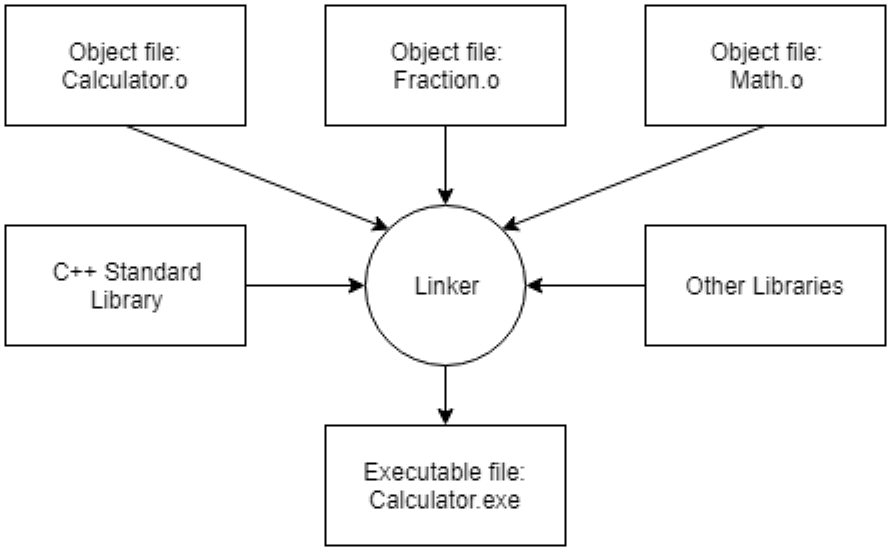
First, the linker reads in each of the object files generated by the compiler and makes sure they are valid.

Second, the linker ensures all cross-file dependencies are resolved properly. For example, if you define something in one .cpp file, and then use it in a different .cpp file, the linker connects the two together. If the linker is unable to connect a reference to something with its definition, you'll get a linker error, and the linking process will abort.

Third, the linker also is capable of linking library files. A **library file** is a collection of precompiled code that has been "packaged up" for reuse in other programs.

C++ comes with an extensive library called the **C++ Standard Library** (usually shortened to *standard library*) that provides a set of useful capabilities for use in your programs. One of the most commonly used parts of the C++ standard library is the *iostream library*, which contains functionality for printing text on a monitor and getting keyboard input from a user. Almost every C++ program written utilizes the standard library in some form, so it's very common for the standard library to get linked into your programs. Most linkers will automatically link in the standard library as soon as you use any part of it, so this generally isn't something you need to worry about.

You can also optionally link other libraries. For example, if you were going to write a program that played sounds, you probably would not want to write your own code to read in the sound files from disk, check to ensure they were valid, or figure out how to route the sound data to the operating system or hardware to play through the speaker -- that would be a lot of work! Instead, you’d probably download a library that already knew how to do those things, and use that. We’ll talk about how to link in libraries (and create your own!) in the appendix.



Once the linker has finished linking all the object files and libraries, then (assuming all goes well) you will have an executable file that you can run.

Building

Because there are multiple steps involved, the term **building** is often used to refer to the full process of converting source code files into an executable that can be run. A specific executable produced as the result of building is sometimes called a **build**.

For advanced readers

For complex projects, build automation tools (such as **make** or **build2**) are often used to help automate the process of building programs and running automated tests. While such tools are powerful and flexible, because they are not part of the C++ core language, nor do you need to use them to proceed, we’ll not discuss them as part of this tutorial series.

Steps 6 & 7: Testing and Debugging

This is the fun part (hopefully)! You are able to run your executable and see whether it produces the output you were expecting!

If your program runs but doesn’t work correctly, then it’s time for some debugging to figure out what’s wrong. We will discuss how to test your programs and how to debug them in more detail soon.

Integrated development environments (IDEs)

Note that steps 3, 4, 5, and 7 all involve software programs that must be installed (editor, compiler, linker, debugger). While you can use separate programs for each of these activities, a software package known as an integrated development environment (IDE) bundles and integrates all of these features together. We’ll discuss IDEs, and install one, in the next section.