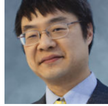
A graphical depiction of the steps in building a C++ executable, basics

May 29, 2024



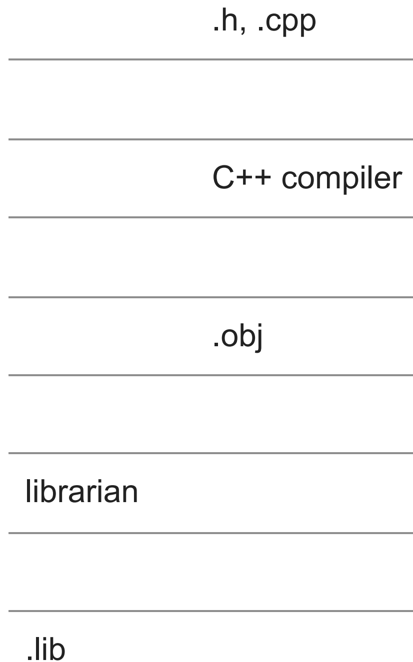
Raymond Chen

One of the things you have to know when trying to diagnose a build failure is understanding what each step of the build accomplishes, so that you can fix the problem at the correct step.

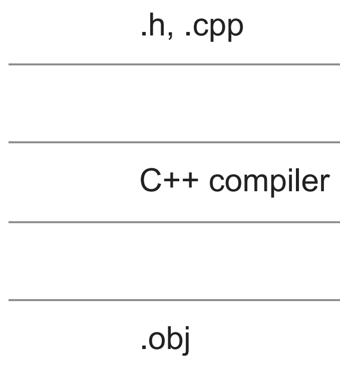
Here's the basic idea.

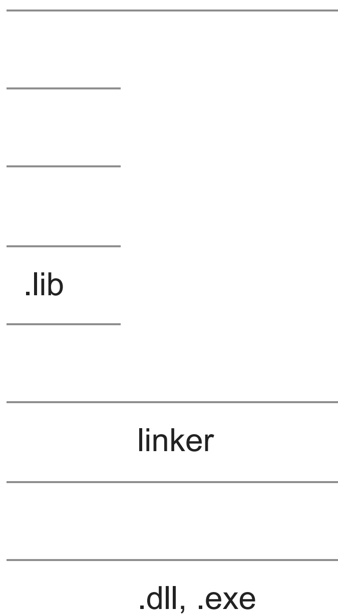
	.h, .cpp
	↓
	C++ compiler
	↓
	.obj
↓	
librarian	
↓	
.lib	
	↓
	linker
	↓
	.dll, .exe

Given a bunch of header files (.h) and C++ source files (.cpp), the C++ compiler produces a corresponding set of object files (.obj).[1] If your project builds a library, then the object files are given to the librarian to produce a library file, and that's your project. In other words, your library project uses only the top part of the diagram, through to building the .lib file, but doesn't go down the other branch that leads to the linker.



If your project builds an executable module (a .dll or .exe, for example), then the object files and any input libraries are given to the linker, which then generates the desired module file. In that case, you're using the other branch of the diagram that leads to the linker:





Already you know enough to solve this problem:

> My solution consists of three projects: A core library, a program that consumes the library, and a unit test that consumes that same library. I added some code to the core library, and now the program and unit test are generating linker errors due to a missing import library used by the code I added. I added that library to the core library's `AdditionalDependencies`, but that didn't fix it.

Next time, we'll expand this diagram to include additional tools you may encounter in Windows projects.

Answer to exercise: The error complaining about the unresolved external symbol is coming from the linker, so you need to provide the library with that symbol to the linker. The core library project doesn't run the linker: It runs the librarian to produce `corelibrary.lib`. It's the program and unit test that consume the `corelibrary.lib` and produce `contoso.exe` and `contoso_unittest.exe`, respectively. Those are the projects that need the new library listed in the `AdditionalDependencies`.

To avoid having to update both the program and unit test each time the build requirements of the core library change, you might put those special configuration settings in a separate file that is included by the program and unit test projects, so that any changes need to be made in only one place.

[1] For unix, the same principles apply, but the file extensions are different. The extension for object files is .o, the extensions for library files are .a and .so, depending on what type of library they are. And unix executables traditionally have no extension.