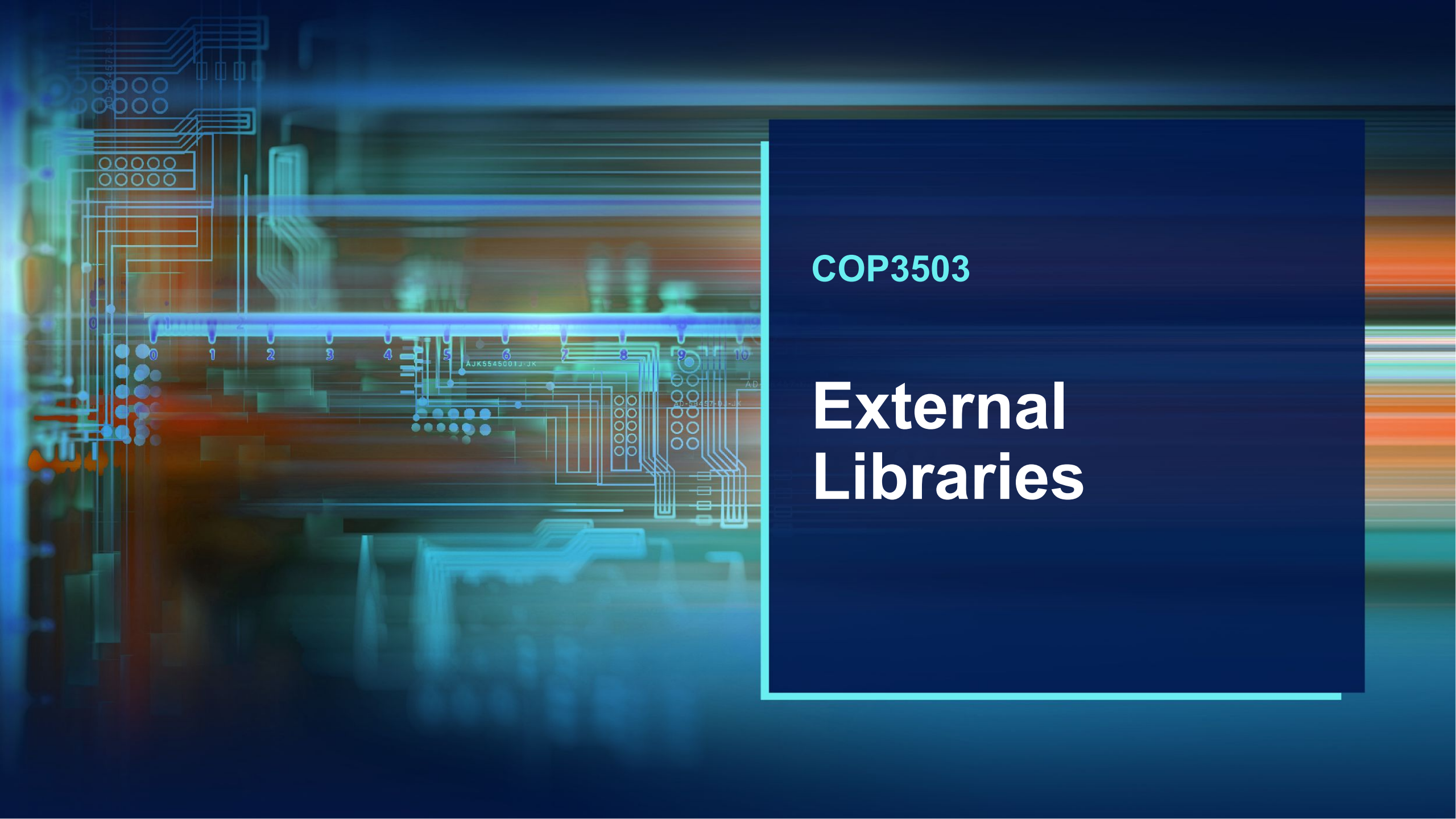
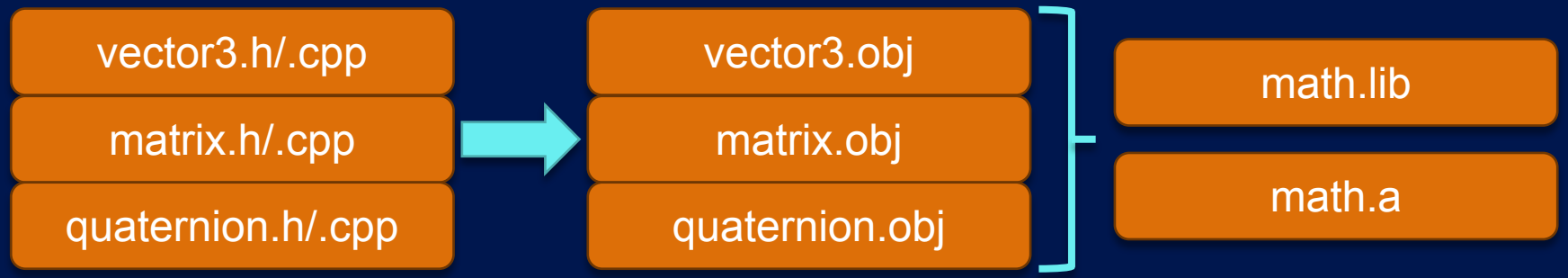


COP3503

External Libraries

What Are Libraries?

- **Definition 1**: Libraries are code solutions to some type of problem
  - Graphics functionality, networking functionality, support for external hardware devices
- **Definition 2**: Reusable code (i.e., classes and functions) compiled into a single file—**a library file** (sometimes called an **archive**)
- Remember compiling code into object files, and then an executable?
- We can compile to a library instead of an executable, and import that library into other projects.



.lib == MSVC file
.a == g++ file

Both serve the same purpose for their respective compilers.

Libraries are distributed with **header files** you need to #include in order to use the library in your code.

Why Use Libraries?

- C++ (especially `namespace std`) has lot of functionality, but it's "core" computational processes are:
  - Primitive types, storage containers
  - Data structures like vectors and maps
  - File I/O
  - Iterators over containers
- These are solid components of any application.
- Applications often need functionality beyond this.

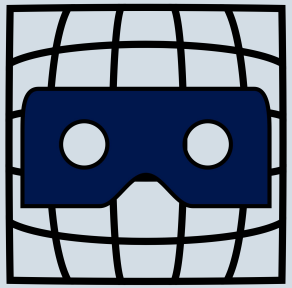
Languages grow over time, and may include "native" solutions that were previously only accessible by using an external library.

Example: A library called boost was almost a necessity, but modern C++ now contains much of the same functionality.

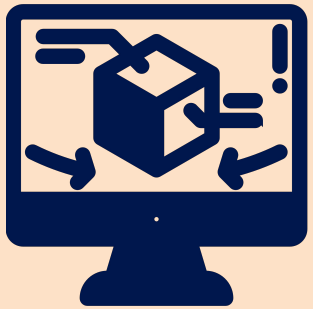
Examples Going Beyond the Core

Communicating with a website

Drawing 2D or 3D graphics



Building a user interface: Needs a graphics library plus the UI-specific functionality



Working with audio or video files

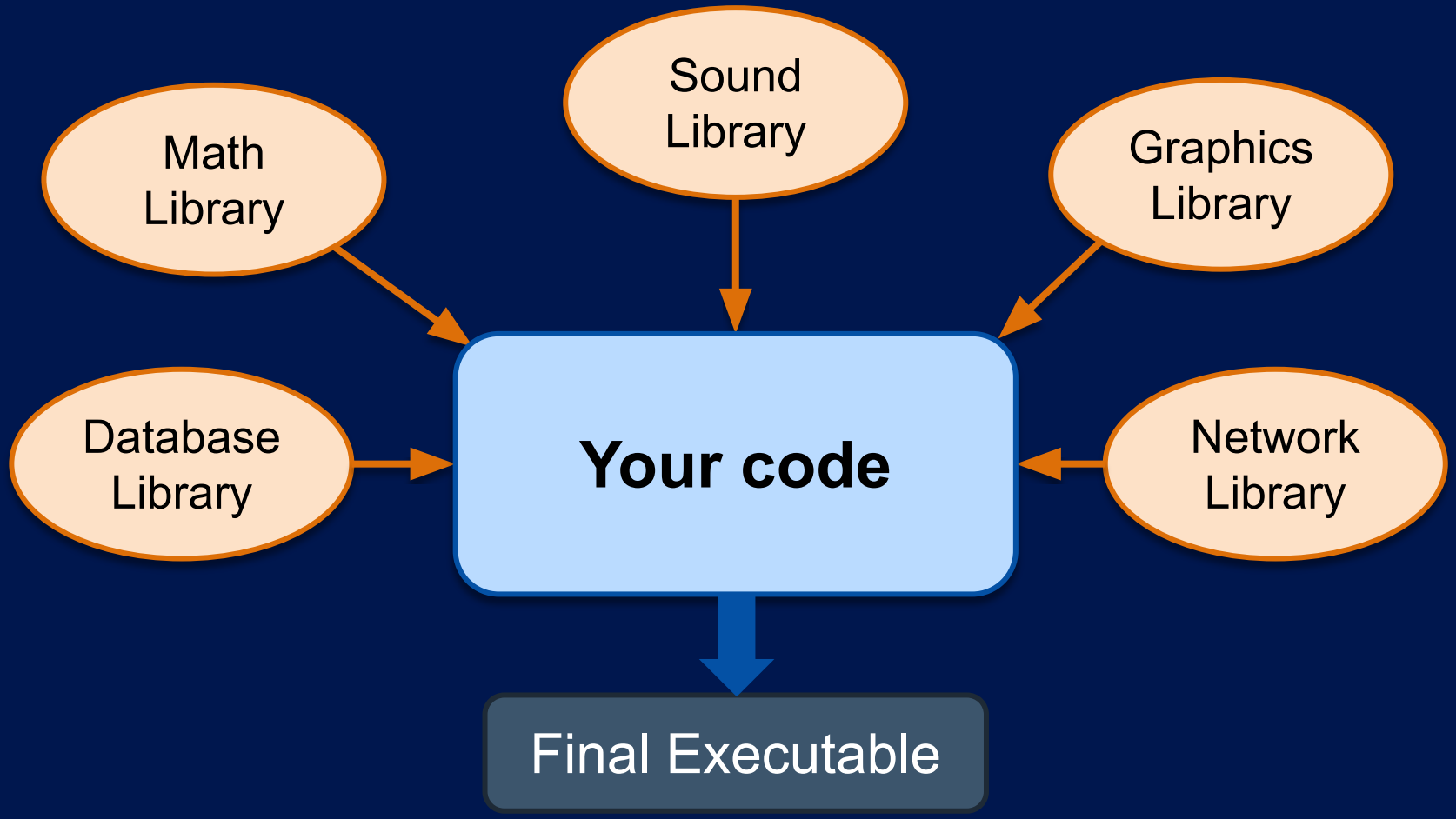
- Some larger “libraries” are made of multiple smaller library files.

SFML (Simple and Fast Multimedia Library) has library files for graphics, networking, audio, and more.

The Windows SDK contains numerous libraries for building Windows-specific programs.

SDK stands for **Software Development Kit**—a large-scale “library” that could contain multiple library files, helper applications, debuggers and more.

Applications May Use Many Libraries



Using an external library just gives you **access** to a tool – you still have to use it.

It's similar in ways to writing `#include <vector>` in your code

Once you have access, it's up to **you** to utilize that functionality.

External Code Is More Code for You to Learn

```
glGenBuffers(1, &renderInfo.vertexBuffer);
glGenBuffers(1, &renderInfo.indexBuffer);
glGenVertexArrays(1, &renderInfo.vertexArrayObject);

glBindVertexArray(renderInfo.vertexArrayObject);

glBindBuffer(GL_ARRAY_BUFFER, renderInfo.vertexBuffer);
glBufferData(GL_ARRAY_BUFFER, vertSize * vertexElementSize,

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, renderInfo.indexBuffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indexSize * sizeof(unsigned int), indexData, GL_STATIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (void *)0);
glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (void *)(3 * sizeof(float)));
glEnableVertexAttribArray(1);

glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(GLfloat), (GLvoid*)(6 * sizeof(GLfloat)));
glEnableVertexAttribArray(2);
```

gl<something>
These are all part of the OpenGL graphics library.
New functions, new data types, lots more to learn!

Learning how to use someone else's library can take a considerable amount of time, depending on the size/complexity of the library.

Steps for Using External Libraries

1. Install the thing.
2. Integrate it into a project.
3. Write code to utilize it.

Steps 1 and 2 can be some of the most difficult, time-consuming, and frustrating parts of a project.

It can be a lot of work to prepare to do the **real** work (i.e., step 3).

Installation: Easy / Preferred Mode

What you want to encounter:	What's often the case:
1. Download an installer program. 2. Run, wait a little bit, then it's done automagically (maybe you have to click "Next" a few times).	1. Download this .zip file. 2. Extract to some folder. 3. Tell your operating system where to find that folder (or reference that location when building a project).

Linux version:
Install the right package, often easy as: `apt install <packageName>`

Installation: Developer Mode

(A.K.A., Why developers get paid the Big Bucks!)

1. Figure out which version to download.
   - Which operating system?
   - Which compiler?
   - Pre-compiled library or the source code?
   - Choose wisely, or it'll break later!
2. Download and extract .zip file to some location.
3. Run an installer using Python via a command-line interface.
4. Realize you don't have Python.
5. Install Python.
6. Run original installation command again.
7. Realize you didn't install the **correct** version of Python.
   - You downloaded the newest version? Well, there's your mistake right there! (LOL, silly developer)

Life lesson:
New != Better
New != Best
New == New

Installation: Developer Mode

(A.K.A., Why developers get paid the Big Bucks!)

8. Go back to the documentation and learn which version of Python you need.
9. Install that.
10. Realize you can't have the old version **and** the new version installed without some issues (you can overcome this, but still… one more thing…).
11. Uninstall the new one.
12. Go back to step <whatever> and resume setup of the original tool.
13. Success! Maybe, hopefully…

+ Replace **Python** with any other **dependency** (or any **NUMBER** of dependencies) and the rest of this still applies.

A library you are trying to use may itself depend on another library, which depends on… etc.

Dependency: Just something your current program or library needs to function properly.

Integrating Into a Project

- This is where it can get a little ugly…
- Every library is different, you just have to sort it out.

Easy Mode

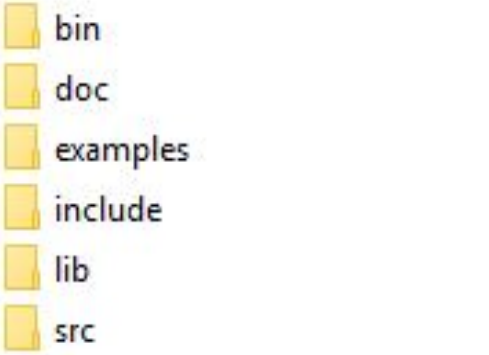
- Installation included a sample project or some default settings you can use.
- There could be a **makefile**, or some program you run, and assuming Part 1 was done correctly, it “just works”.
- Open up that project and start writing your own code!

Ideally, the developers of this thing worked diligently to ensure this was the case.

Building a Project: Developer Edition

- First, what files (library files and headers) do you need?
- Set your project to reference those files.
- Set your project to reference the location of those files.
- Your IDE may have settings for all of these, or you can learn the different compiler options for a command-line build.

C:/APIs/SomeLibrary



Project 1
Location: C:/Code/Projects/Proj1

```
library_path = C:/APIs/SomeLibrary/lib
include_path = C:/APIs/SomeLibrary/include
libraries = graphics.lib, math.lib, network.lib
```

Project 2
Location: E:/Dev/Code/Proj/Proj2

```
library_path = C:/SomeLibrary/lib
include_path = C:/APIs/SomeLibrary/include
source_path = C:/APIs/SomeLibrary/src
libraries = graphics.lib, database.lib
```

There Isn’t a Universal Standard

- Not all libraries will work* with all compilers.
  - They may not work **easily**.
- There is often a path of least resistance:
  - “For best results, use **<compiler>** on **<Operating System>**”
- If you can’t or don’t want to follow the suggestions?
  - (i.e., You’re of the “Notepad or bust!” mindset)?
- Sometimes there are viable alternatives… and sometimes there aren’t.

- Some developers may provide options, different versions of a library.
- Others may just provide source code, and then you build that code into the .lib files you need

Download SFML 2.5.1

On Windows, choosing 32 or 64-bit libraries should be based on which platform you want to compile for, not which OS you have. Indeed, you can perfectly compile and run a 32-bit program on a 64-bit Windows. So you'll most likely want to target 32-bit platforms, to have the largest possible audience. Choose 64-bit packages only if you have good reasons.

The compiler versions have to match 100%!
Here are links to the specific MinGW compiler versions used to build the provided packages:
TDM 5.1.0 (32-bit), MinGW Builds 7.3.0 (32-bit), MinGW Builds 7.3.0 (64-bit)

Visual C++ 15 (2017) - 32-bit	Download | 16.3 MB	Visual C++ 15 (2017) - 64-bit	Download | 18.0 MB
Visual C++ 14 (2015) - 32-bit	Download | 18.0 MB	Visual C++ 14 (2015) - 64-bit	Download | 19.9 MB
Visual C++ 12 (2013) - 32-bit	Download | 18.3 MB	Visual C++ 12 (2013) - 64-bit	Download | 20.3 MB
GCC 5.1.0 TDM (SJLJ) - Code::Blocks - 32-bit	Download | 14.1 MB		
GCC 7.3.0 MinGW (DW2) - 32-bit	Download | 15.5 MB	GCC 7.3.0 MinGW (SEH) - 64-bit	Download | 16.5 MB

On Linux, if you have a 64-bit OS then you have the 64-bit toolchain installed by default. Compiling for 32-bit is possible, but you have to install specific packages and/or use specific compiler options to do so. So downloading the 64-bit libraries is the easiest solution if you're on a 64-bit Linux.
If you require a 32-bit build of SFML you'll have to build it yourself.

It's recommended to use the SFML version from your package manager (if recent enough) or build from source to prevent incompatibilities.

Linux	GCC - 64-bit	Download | 2.21 MB

macOS	Clang - 64-bit (OS X 10.7+, compatible with C++11 and libc++)	Download | 5.50 MB
	macOS libraries are only compatible with 64-bit systems.	

API (Application Programming Interface)

- The structure of the code you have access to is its API.
- What **header files** do you need to #include?
  - These are a “table of contents” to indicate what is in those libraries you referenced.
  - (Surprise! Header files aren’t just for irritating students!)
- What **namespaces** are in those headers?
- What **classes/functions** are in those namespaces? That’s what the library can **do**.
- Not all APIs are created equally… some are easy to use and understand, have good documentation.
- Others? Not so much! Bad documentation, confusing naming conventions, “unclean” code, etc.

How Do You Know You're Doing it Right?

- Does it compile? Your compiler will tell you if it's broken.
  - Actually **fixing** that problem could be complicated.
  - Missing a path? Might be a simple "file not found" error. Didn't set a property? Code you didn't write generates errors—how do you even begin to fix that? (Hint: Set the property!)
- Until you **do it** (and possibly fall flat on your face), you don't know.
- Read the documentation to see what you may have missed.
- Sometimes the error isn't apparent—the fastest solution is to start from the beginning and recheck everything.

Is Working With a Library Always This Bad?

- Yes and no. Some languages are better than others.
- The nature of C++ makes it a more “hands on” process.
  - In C++ you have more control (and power) but more responsibility.
- A language like Python is “easier” to use in many ways.
  - There’s a standardized system of importing packages and modules (“library” isn’t used as often)

(Pyramid International, 2022)

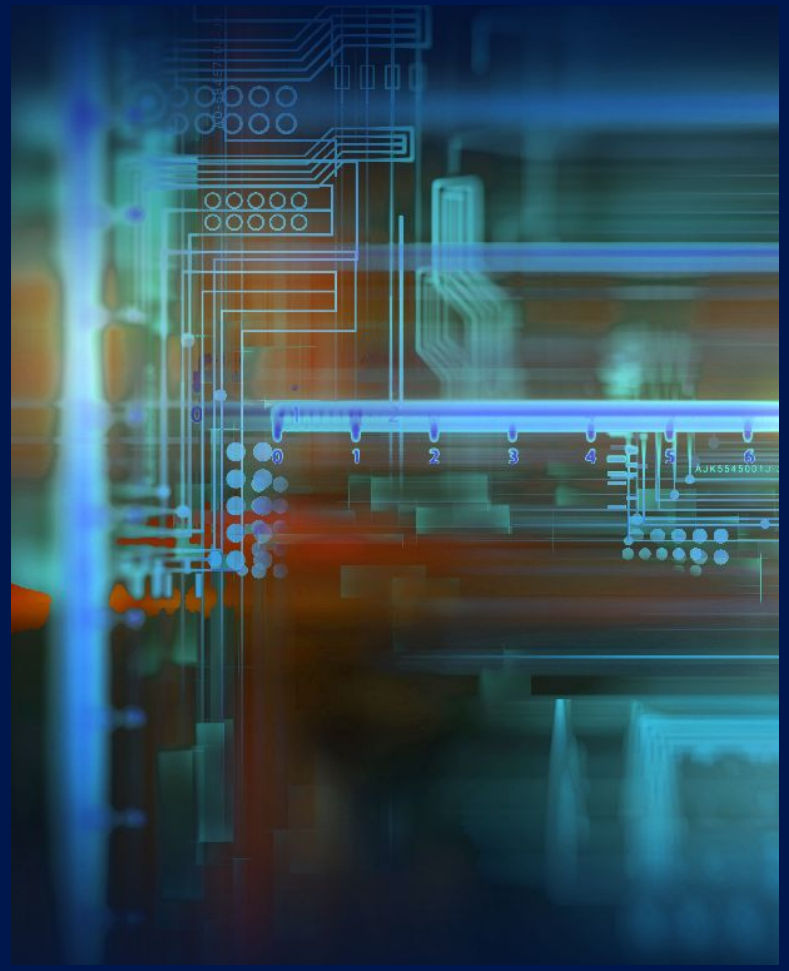
A Necessary Evil

- The initial set up of a project, or a tool chain, is nearly always a pain.
- **THIS IS UNAVOIDABLE**.
- You **CANNOT** run away from this.
- Sometimes you get to use tools that “just work” and it’s a wonderful, glorious thing!
- For all those other times… you have to live with it, and just deal.
  - Download yet another update, another driver, install another version of another library, etc.
- You will get more comfortable navigating this space as you do it more frequently.
- Every new tool has a new set of quirks to sort out.

Recap

- Libraries are collections of code written to solve a particular problem (i.e., provide some functionality).
- They can be imported into other programs and reused.
  - Most large-scale programs use numerous libraries.
- Integrating libraries can be a complex process.
  - We get better at it over time!
- This process is unavoidable, and potentially different for every library.
- After integration, we have to actually use it!
  - The overall structure/organization of a library is called an API (Application Programming Interface).

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.

References

Pyramid International. (2022). Image of Spiderman [Online Image]. Amazon. https://www.amazon.com/Pyramid-International-Spider-Man-Plastic-Multi-Colour/dp/B00AWX2BFW