COP3503

Command Line Compiling

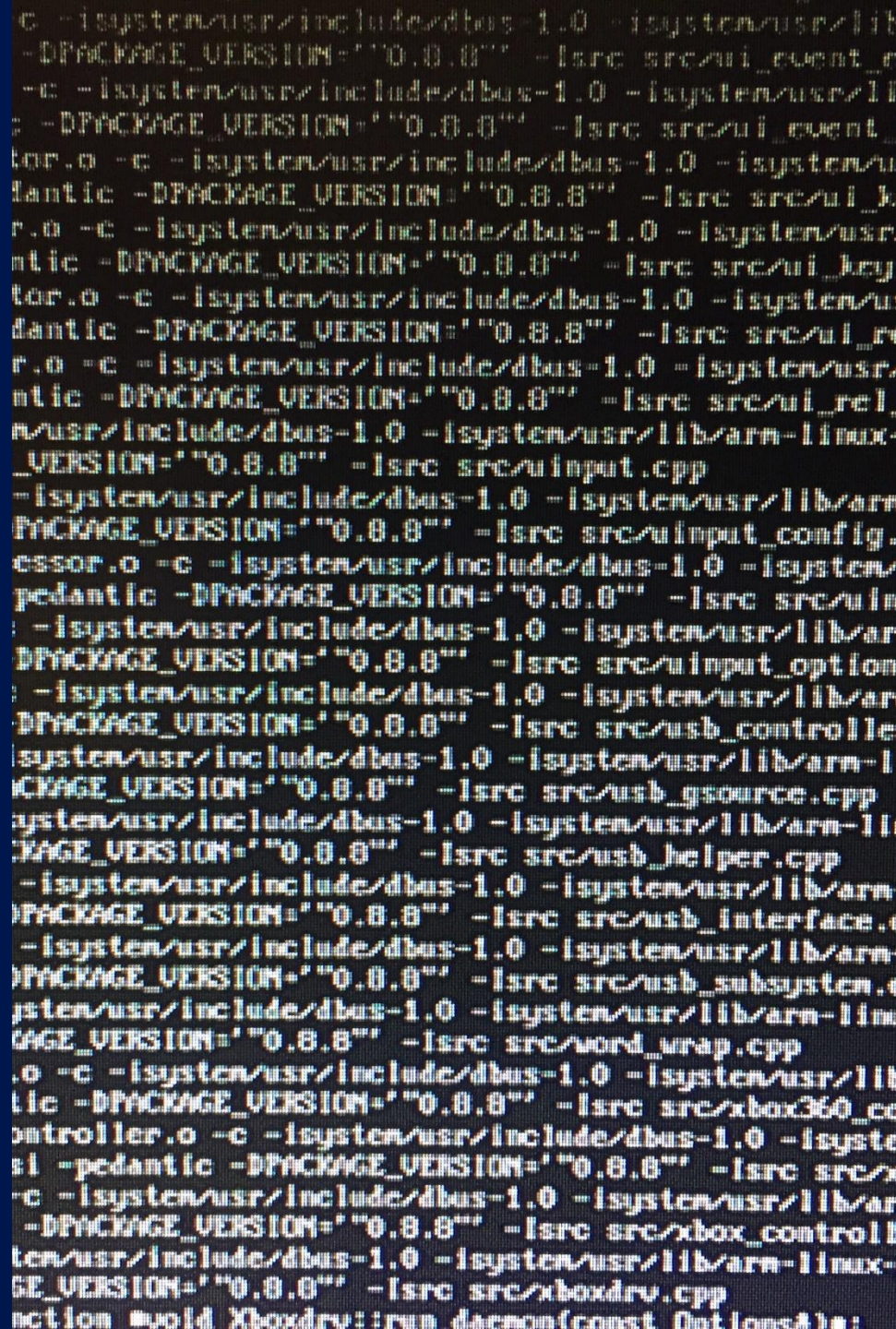
Compiling From the Command Line

- You don't have to write code in an IDE (Integrated Development Environment).
- It makes some aspects of development easier and convenient, but it's not strictly **necessary**.
  - For some tasks, or projects of a certain size, it can be difficult to work effectively without one.
- The **compiler** determines whether or not you've written valid code.
- Behind the scenes, your IDE uses the same compiler.
- The compiling features of your IDE can be executed "by hand" from the command line.

Consider this:
How did people write and compile the code to create an IDE, before the existence of an IDE?

Using the Compiler

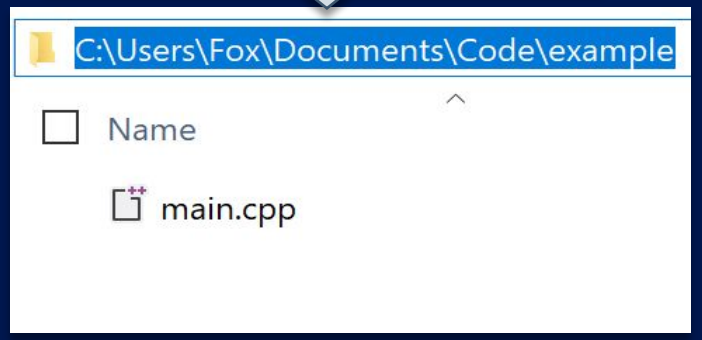
- Fundamentally, your code is compiled by this process:
  - Run `NameOfCompilerProgram`, indicating which files you want to use to build a program
  - Those files are then sent through the stages of the C++ build process (Preprocessor, Compiler, Linker).
  - Assuming no errors, the output is an executable, the program you wrote.



g++: The C++ compiler from GCC (the GNU Compiler Collection)

- **g++**, or **g++.exe** is the name of the compiler program.
- It accepts **arguments** in the form of code files as well as compiler options.
  - We'll talk more about command-line arguments later.

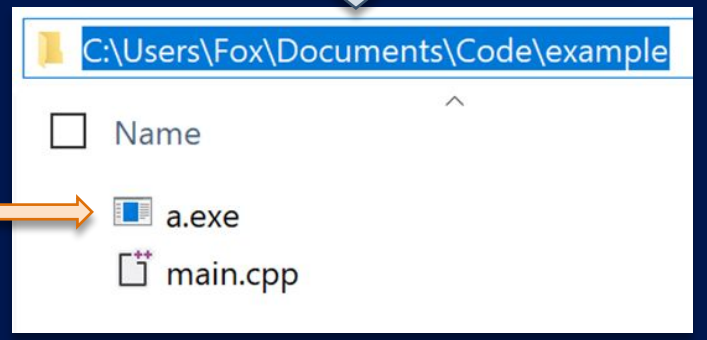
If this is all of our code



We can build it using this command.

```
> g++ main.cpp
```

Which generates this executable



- On a basic level, that's it. Run g++ and tell it what code you want to build. Everything else is just options.

Building With Multiple Files

- In a directory with files:
  main.cpp, **vehicle.cpp** and **vehicle.h**

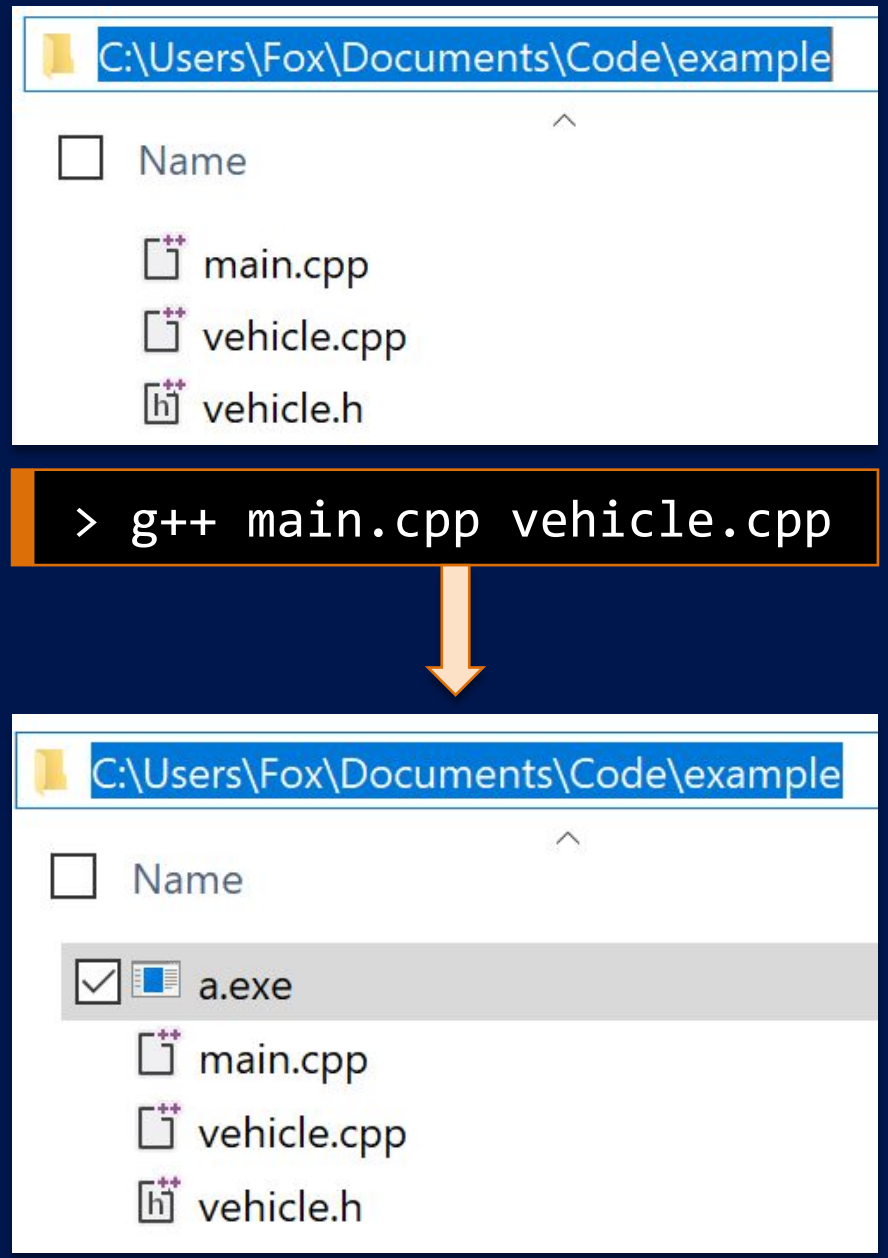
```
> g++ main.cpp vehicle.cpp
```

We only specify source files (.h files are **`#included`** by those source files).

- Or:

```
> g++ *.cpp
```

Compile **all** .cpp files in this directory (the * is called a wildcard).



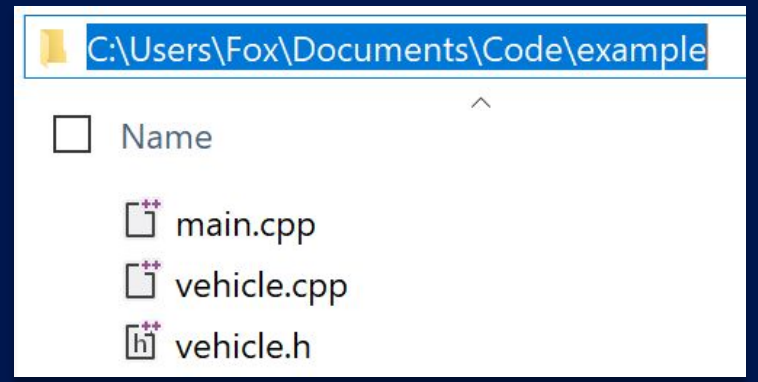
Renaming the Output Executable

- The argument `-o [filename]` lets you specify the name of the executable that gets created.

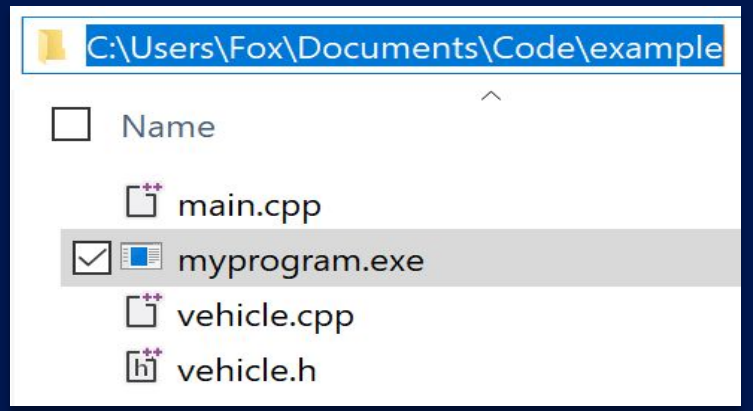
```
> g++ -o myprogram main.cpp vehicle.cpp
```

- Or:

```
> g++ -o myprogram *.cpp
```

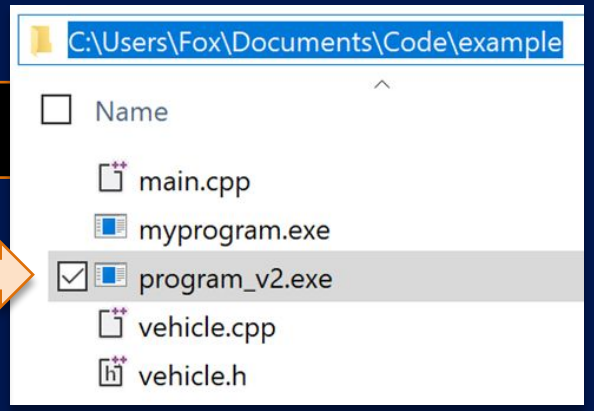



```
> g++ -o myprogram *.cpp
```



Build again, creating a second executable with a different name.

```
> g++ -o program_v2 *.cpp
```



There Are Lots of Arguments and Options You Can Use

- This will do a “full build” and generate an executable.

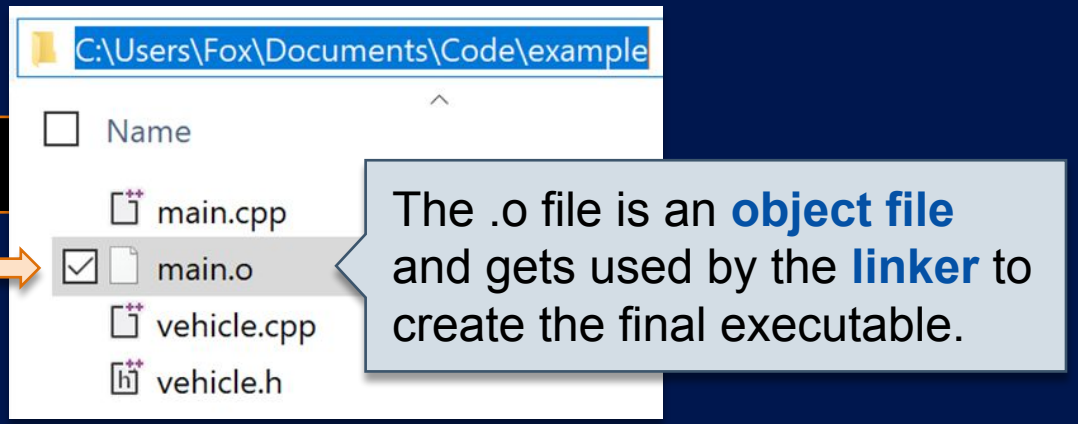
```
> g++ *.cpp
```

- What if you had dozens, hundreds of files?
  Build times can be long (hours!) for large projects.

- What if you wanted to see if a single class (or function) merely compiles?

The `-c option` will **only** compile the specified files.

```
> g++ -c main.cpp
```



The .o file is an **object file** and gets used by the **linker** to create the final executable.

After Compiling Comes Linking

- Build the executable from a single **object file** (**not** a code file)

```
> g++ -o myprogram main.o
```

- Build an executable from **mutliple** object files (assuming they exist)

```
> g++ -o myprogram main.o vehicle.o
```

- Or... use wildcards again

```
> g++ -o myprogram *.o
```

Doing separate compile / link operations is something you might not need to do often.

In this course, we won't have any programs (or project setups) that are so complex that we need to.

In many cases, this is sufficient:
`g++ -o [program_name] cpp_file(s)`

What About Other Compilers?

- Microsoft's C++ compiler (called MSVC, or Microsoft Visual C++) uses an executable called **cl.exe**.
- Using it is similar (but **not** identical) to using g++.

For g++:

```
> g++ [options – file(s), executable name, etc]
```

For MSVC:

```
> cl [options – file(s), executable name, etc]
```

- The overall concept behind these two separate compilers is the same.
- The actual details when it comes to various options can be quite different!

Quick and Easy

- In a directory with a source file called main.cpp:

For GCC:

```
> g++ main.cpp  // output is a.exe by default
```

For MSVC:

```
> cl main.cpp  // output is main.exe
```

Default executable name is the **first** file compiled.

Renaming an Executable

- In a directory with a source file called main.cpp:

For GCC:

```
> g++ -o program main.cpp     // program.exe
```

For MSVC:

```
> cl /Fe: program main.cpp    // program.exe
```

/Fe:
Change the name of the executable

And so on, and so on…

In the end, they're both C++ compilers, and they can more or less do the same thing.

What About Complex Builds?

```
cl /EHsc /Fe: RPG.exe *.cpp /link /LIBPATH:c:/pdcurses/wincon pdcurses.lib
```

- `cl` The base command
- `/EHsc` Turn on exception handling
- `/Fe` Name the executable
- `*.cpp` Wildcard, use any file with .cpp extension
- `/link` Make use of a library, which contains additional code
- `/LIBPATH` Where to find some or all libraries
- `pdcurses.lib` The actual library being linked

At first glance, that might all seem like a bunch of nonsense!

Not at all like this:

```
for (int i = 0; i < 10; i++)
    cout << i * 5;
```

Lots of things are strange at first, especially with programming!

Fortunately, you don't have to learn all of them!

There Are a Lot of Options

From MVSC – Microsoft's C++ compiler

```
@
/?
/AI
/analyze
/arch
/await
/bigobj
/C
/c
/cgthreads
/clr
/constexpr
/D
/diagnostics
/doc
/E
/EH
/EP
/errorReport
/execution-chars
et
/F
/favor
/FA
/Fa
/FC
/Fd
/Fe
/FI
/Fi
/Fm
/Fo
/fp
/Fp
/FR
/Fr
/FS
/FU
/Fx
/GA
/Gd
/Ge
/GF
/GH
/Gh
/GL
/Gm
/GR
/Gr
/GS
/Gs
/GT
/guard:cf
/Gv
/Gw
/GX
/Gy
/GZ
/Gz
/H
/HELP
/homeparams
/hotpatch
/I
/J
/JMC
/kernel
/LD
/LDd
/link
/LN
/MD
/MDd
/MP
/MT
/MTd
/nologo
/O1
/O2
/Ob
/Od
/Og
/Oi
/openmp
/Os
/Ot
/Ox
/Oy
/P
/permissive-
/Qfast_transcendentals
/Qlfist
/QIfist
/Qimprecise_fwaits
/Qpar
/Qsafe_fp_loads
/Qvec-report
/RTC
/sdl
/showIncludes
/source-charset
/std
/Tc
/TC
/Tp
/TP
/U
/u
/utf-8
/V
/validate-charset
/vd
/vmb
/vmg
/vmm
/vms
/vmv
/volatile
/w
/W0, /W1, /W2, /W3, /W4
/w1, /w2, /w3, /w4
/Wall
/wd
/we
/WL
/wo
/Wp64
/Wv
/WX
/X
/Y-
/Yc
/Yd
/Yl
/Yu
/Z7
/Za
/Zc
/Ze
/Zf
/Zg
/ZI
/Zi
/Zl
/Zm
/Zp
/Zs
/ZW
```

There Are a Lot of Options

From MVSC – Microsoft's C++ compiler

```
@                  /FA      /Gm      /link
/?                 /Fa      /GR      /LN
/AI                /FC      /Gr      /MD
/analyze           /Fd      /GS      /MDd
/arch              /Fe      /Gs      /MP
/await             /FI      /GT      /MT
/bigobj            /Fi      /guard:cf /MTd
/C                 /Fm      /Gv      /nologo
/c
/cgthreads
/clr
/constexpr
/D
/diagnostics       /FS      /H       /Oi
/doc               /FU      /HELP    /openmp
/E                 /Fx      /homeparams /Os
/EH                /GA      /hotpatch /Ot
/EP                /Gd      /I       /Ox
/errorReport       /Ge      /J       /Oy
/execution-chars   /GF      /JMC     /P
et                 /GH      /kernel  /permissive-
/F                 /Gh      /LD      /Qfast_transcendentals
/favor             /GL      /LDd     /Qlfist
```

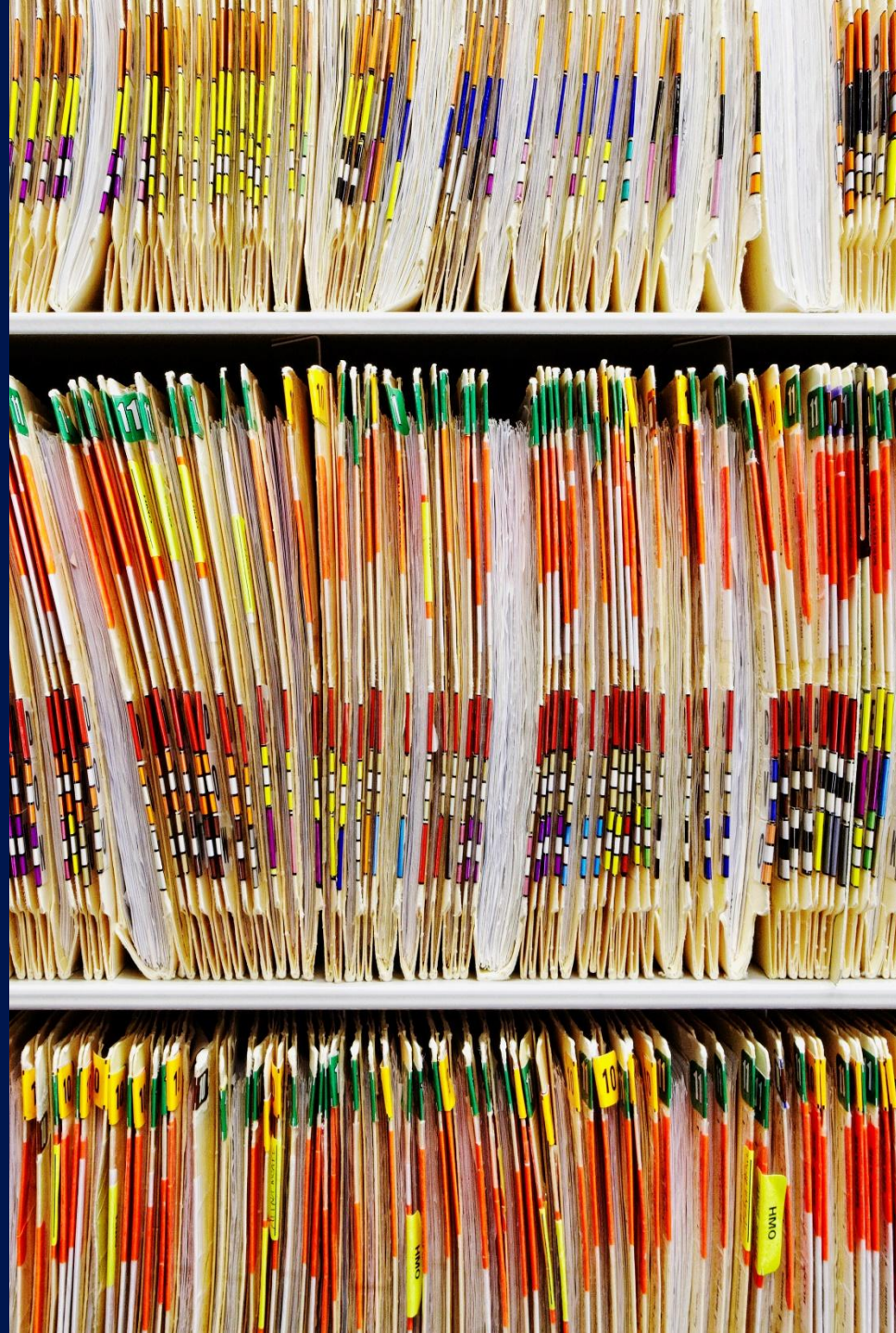
These all serve a purpose (or did, at one point—some may be older and deprecated).

```
/QIfist                    /w1, /w2, /w3, /w4
/Qimprecise_fwaits         /Wall
/Qpar                      /wd
/Qsafe_fp_loads            /we
/Qvec-report               /WL
/RTC                       /wo
/sdl                       /Wp64
/showIncludes              /Wv
/source-charset            /WX
/std                       /X
/Tc                        /Y-
/TC                        /Yc
/Tp                        /Yd
                           /Yl
                           /Yu
                           /Z7
                           /Za
                           /Zc
/validate-charset          /Ze
/vd                        /Zf
/vmb                       /Zg
/vmg                       /ZI
/vmm                       /Zi
/vms                       /Zl
/vmv                       /Zm
/volatile                  /Zp
/w                         /Zs
/W0, /W1, /W2, /W3, /W4    /ZW
```

Depending on what you do, you **may** need to learn **some** of them.

What About Larger Projects? Complex Projects?

- Multiple files, multiple folders, external libraries, different build types (x86 vs x64, Debug vs Release), etc…
- How do you keep track of everything to build all of those?
- Either get **really** good at remembering typing lots of complex commands (was it `/Fo` or `/Fe` that I needed?), **or**…
- Start using a **makefile**.



What's a makefile?

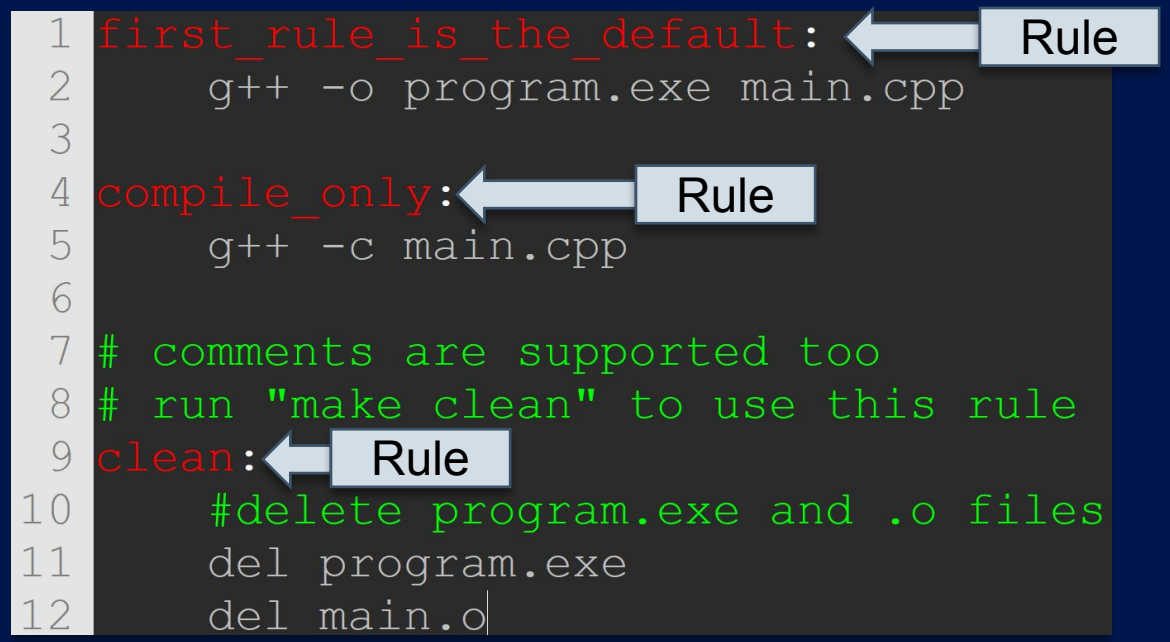
- A **makefile** is a file that contain a list of operations to perform (kind of a "to do" list).
- The makefile itself is just the instructions (the operating system and compiler will do the work).
- Makefiles are often used to help compile code and build large programs.
- A **make** program will read a makefile and pass those instructions to the appropriate program.

Super simple makefile: The command to build a program called program.exe from the file main.cpp

```
1 makefile_rule:
2     g++ -o program.exe main.cpp
```

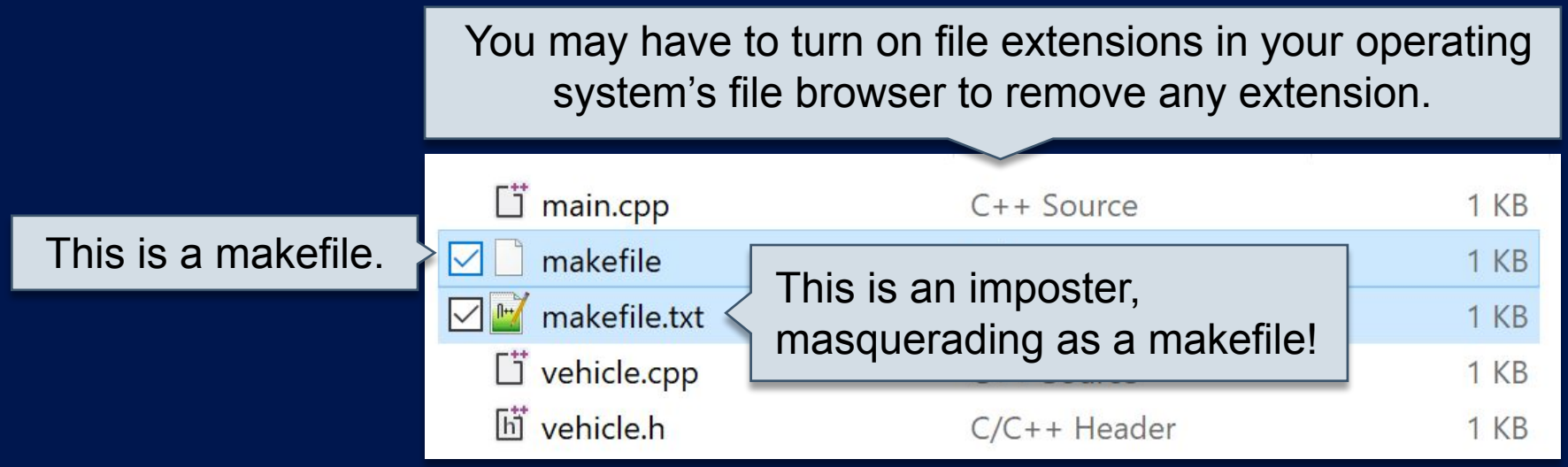
What's Inside a makefile?

- It contains one or more **rules** that each contain a set of instructions.
- Rules can be named anything you want.
- The **first** rule is executed by default, if nothing else is specified.
- Code underneath a rule must be indented with a **TAB** (spaces will generate errors!).



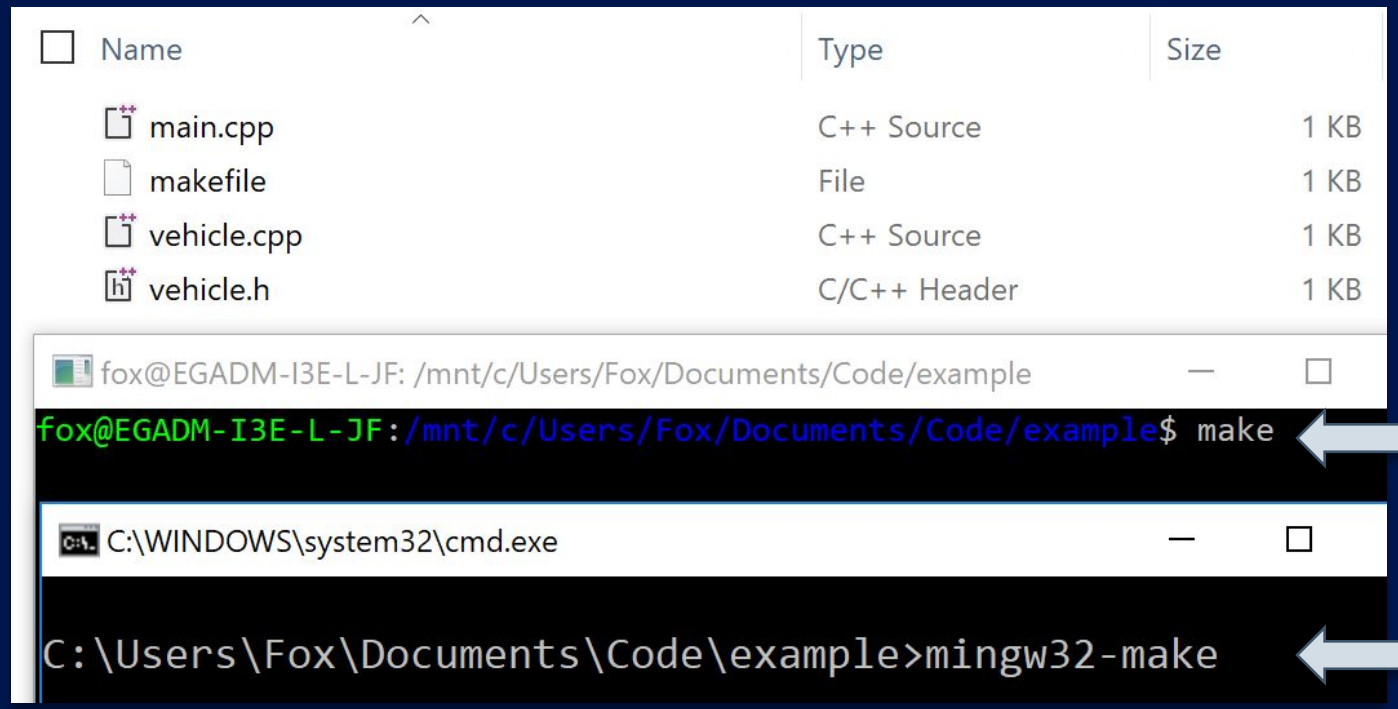
How Do You Create a makefile?

- A makefile is just a plain text file, with these properties:
  - Its name is (usually) just: **makefile**.
  - It has **no extension**—not makefile.txt, not makefile.make, just plain ol' **makefile**.



How Do You “Run” a makefile?

- From a command-line interface, navigate to a folder containing a makefile.
- Run a “make” program -- not all make programs have the same name!



Both of these commands will do the same thing:

1. Search the current directory for a file named “makefile”

2. Open it, read it, execute the instructions

What's a "make" Program?

- A program to open and read a makefile, and then execute the instructions contained within.
- There are many that do similar things (like compilers!).
- It's often called make (generally) but you may encounter:

`make`	The "standard" version, used by g++
`mingw32-make`	Make equivalent for g++ on Windows (if you installed MinGW)
`nmake`	A version for Microsoft's compiler, MSVC
`CMake`	A different, but similar tool (used by Clion, can be used as a standalone tool as well)

Running Different Rules

- Just “make” without a rule will run the first rule, whatever it’s called.
- `make clean` Search for a rule named “clean” and execute its commands
- Replace “make” with the name of your make program (nmake, mingw32-make, etc)

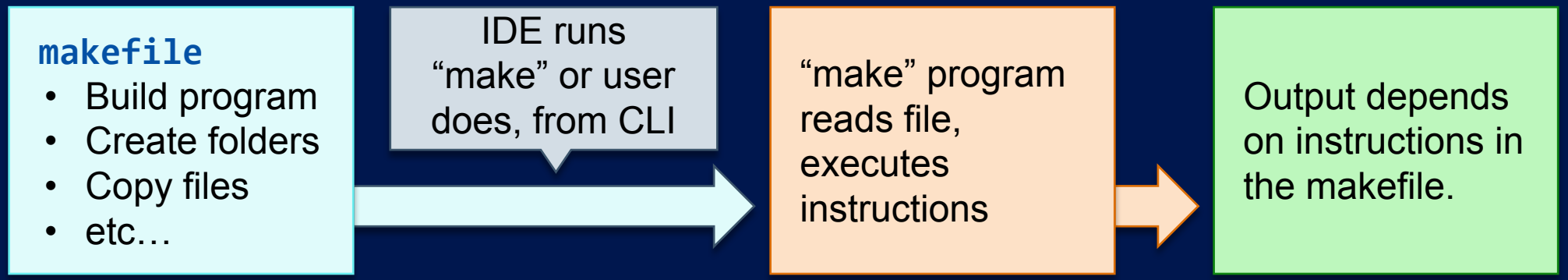
```
1  first_rule_is_the_default:
2      g++ -o program.exe main.cpp
3
4  compile_only:
5      g++ -c main.cpp
6
7  # comments are supported too
8  # run "make clean" to use this rule
9  clean:
10     #delete program.exe and .o files
11     del program.exe
12     del main.o
```

```
C:\Users\Fox\Documents\Code\example>mingw32-make
g++ -o program *.cpp

C:\Users\Fox\Documents\Code\example>mingw32-make compile_only
g++ -c main.cpp

C:\Users\Fox\Documents\Code\example>mingw32-make clean
del program.exe
del *.o
```

Using a make Program



- The process for a user to execute all of the instructions is simple:
  - Just type the name of the make program

Simple makefile, with 1 or 2 lines of commands?

```
> make
```

Enormously complex makefile, with hundreds of commands?

```
> make
```

Makefiles allow for a layer of **abstraction** between the user and the actual commands.

Similar to programming something in a class and hiding it behind a function!

Makefiles Can Be Quite Complex

```
#Replace include and library paths with wherever you have SFML installed
#This assumes you already have a working version of SFML built
include = C:\APIs\SFML-master\include
library = C:\APIs\SFML-master\build\lib

#Shouldn't need winmm or gdi32 if not on windows
LIBRARIES = -lsfml-graphics-s-d \
            -lsfml-window-s-d \
            -lsfml-audio-s-d \
            -lsfml-system-s-d \
            -lopengl32 \
            -lwinmm \
            -lgdi32

# for linking to static versions of the library
CFLAGS = -DSFML_STATIC

# compile all .cpp files in the src directory--assumes all sources files are in that directory
source = $(wildcard src/*.cpp)

#automatically run program.exe (Windows) after building (remove program.exe if not desired)
all:
    g++ -o program $(source) -I$(include) -L$(library) $(LIBRARIES) $(CFLAGS)
    program.exe
program: main.o tile.o
    g++ -o program *.o -L$(library) $(LIBRARIES)
main.o: src/main.cpp
    g++ -std=c++11 -c src/main.cpp -I$(include)
tile.o: src/tile.cpp
    g++ -std=c++11 -c src/tile.cpp -I$(include) $(CFLAGS)
clean:
    del *.o
```

Would you rather type all of this each time you build?

Or just type `make` or (`mingw32-make`)?

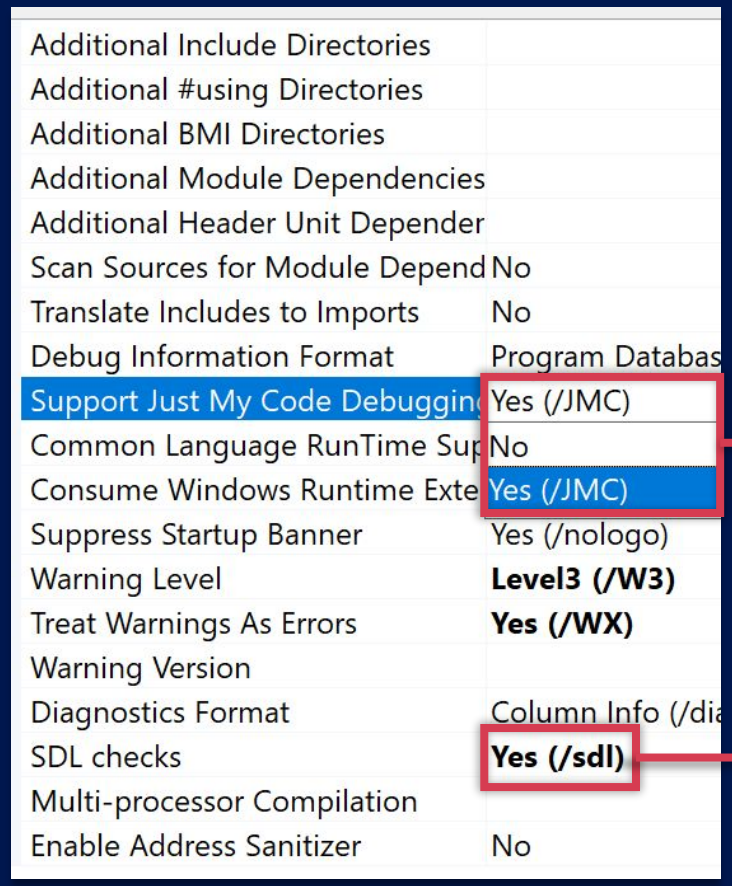
You can do a lot more with this kind of system beyond what we've covered.

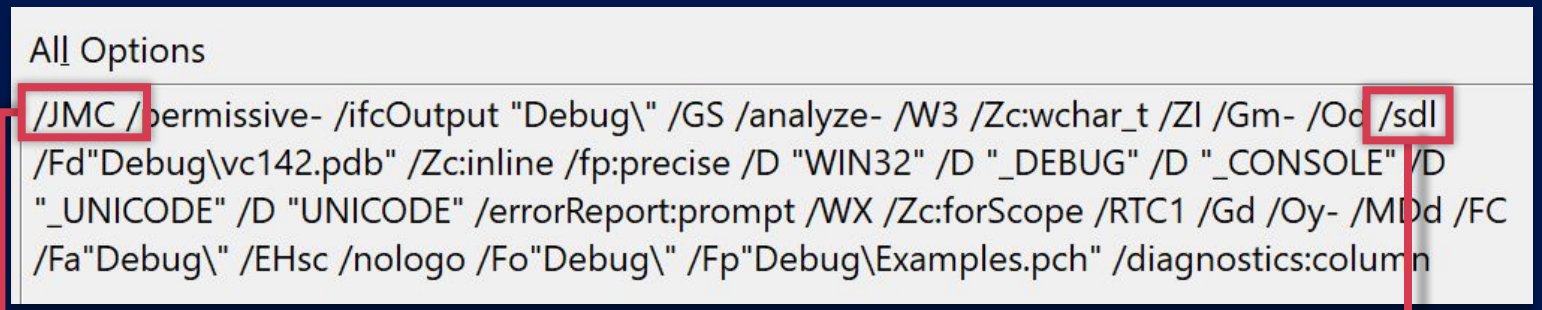
A makefile can have conditions, OS-specific options, directory locations, command-line arguments, etc…

It's a bit overwhelming initially, but very powerful and very versatile.

IDEs Use make Behind the Scences

- Your IDE's interface is just a mask covering up the command-line functionality.
- Most settings and properties you can change have some equivalent in a makefile.





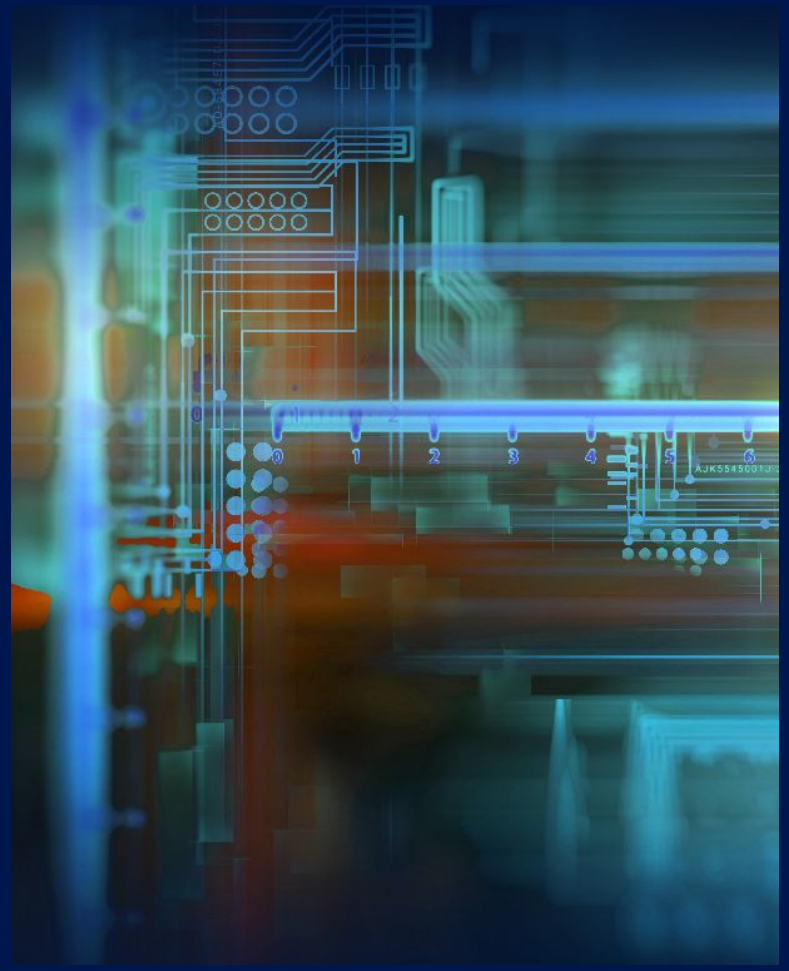
Selecting Yes or No from a list? Easy! Remembering all of the options by name like /Ehsc or /GS? Not as easy!

The IDE helps us manage the build, so we can focus on writing code.

Recap

- Building code with a command-line interface is **just another option**.
- In some cases, it **might** be faster than using an IDE.
- An IDE is really just a shell around a compiler—command-line compiling bypasses the “middleman” that is the IDE.
- You may have to learn a lot of options that are typically “done for you” in your IDE—but this can lead to a lot of **power and flexibility**.
- **makefiles** allow you to simplify the build process by storing complex sequences of commands in a file.
- A **make program** will execute the commands in these files with a simple command, often just “make”.

Conclusion



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

Thank you for watching.