Command Line C++

Meeting C++ 2025

About Me

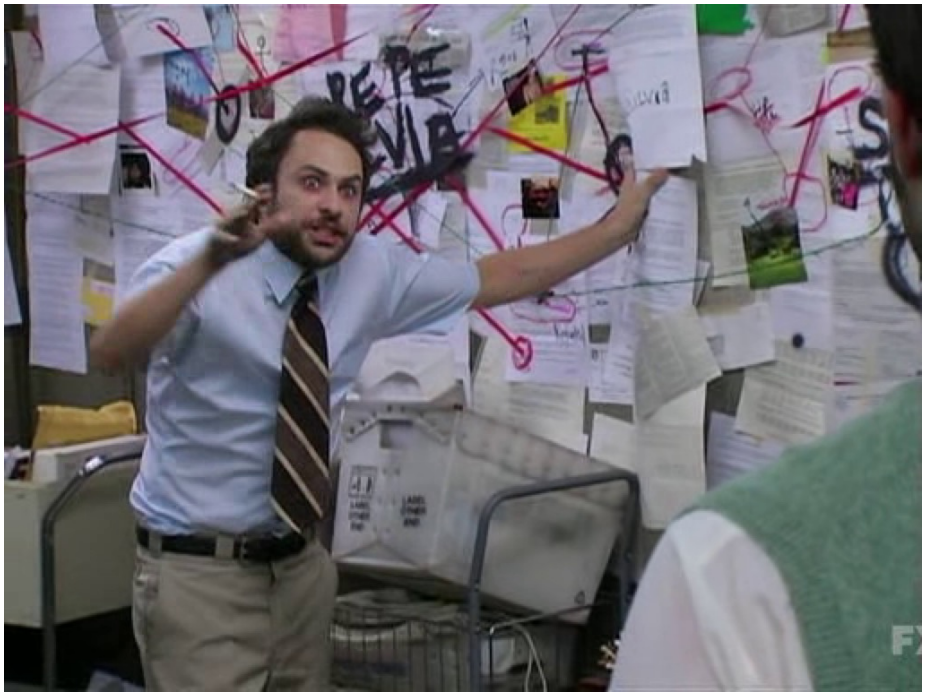
- I am a Graduate Computer Scientist
- I am an independent contractor.
- I like C++
- I also like the command line.
- I am passionate about energy efficient computing.

- Hello, My name is Mathew Benson and I like C++
- Welcome to my talk on command line C++.

Contacts

- Twitter : @bensonorina
- mastodon : @mathewbenson.hachyderm.io
- email : mathew@benson.co.ke
- Github : github.com/mathewbensoncode
- Discord : #CppAfrica -> mathewbensonke

Plan for the Talk



- Rationale
- Command Line Overview
- Command Line C++
- Build Systems & Tooling

- Here is an overview of what I would like to cover in this session.
- I will start with an explanation of why I believe that using the command line in developing C++ is important.
- I will dive into a few definitions to try to explain better what is a superior way to learn and/or improve skill in development in C++
- I will dive into a simplified getting started workflow to illustrate the command line underpinnings of our development environments

Rationale?

What is my goal

- This is a deeper dive into the concepts I discussed in last years talk "C++ in the Developing World, Why it matters"
- An experiment at creating an environment where the C++ language can be learnt, through "Play".

- To empower better learning and/or code understanding. Raising the bar or code knowledge.
- The goal of this talk is to create a roadmap to empower any individual, anywhere to get started with learning about computing and even improve their current skills
- I believe that C++(and of course C) is an important language to learn because its the "primary" language to build the performant programs that can run the world today.
- This includes the main programs used by many people, the main access to operating system internal systems, browsers, code editors, graphic programs, etc.
- This talk may be a little introductory, but the intention is to create an accessible starting point to answer some questions that a beginner may not find all too easily.
- Some crucial concepts and ideas have been overshadowed by the graphical interfaces that have been used for some time now
- I will attempt to show how to use the command line and free resources like cppreference and other free resources as a learning resource as we advance through this talk.

Who This talk is for

- Beginners
- Intermediate Developers, who may have been programming in the language syntax, but rely heavily on the IDEs and tools to deal with the underlying cruft.
- Experts to spark the conversation on how we can do better in passing the knowledge along
- The main aim is mainly to give some guides that I believe to be important for spreading the joy that is developing in C++
- This will not be comprehensive, but is meant to provide a roadmap for things to look into as one does study

Analogy to a movie

- If you ever had the experience of joining a movie halfway and happened to have missed some key points to the plot that were established at the beginning, you kind of "can" enjoy the action and drama of the movie climax, the big last fight, But it really isn't the same without context. Sometimes we need to go back to the beginning, to get a sense of what is going on.

- C, C++ and computing has a long history
- There is a lot of important history and concepts that are easy to overlook
- It may also be harder to enjoy when missing some plot points
- One key factor is the relationship with our operating systems.

- I really like this picture as an illustration of the foundations of our computing today
- Dennis Ritchie and Ken Thompson, who both worked at Bell Labs.
- One is credited with the development of the Unix operating system and the other with the development of the C programming language

- These two programs have been and still are very foundational to our systems today
- Fun fact, Bjarne, the founder of C++ was also at Bell labs and worked just a few doors away.

How do we get started with C++ today?

- The startup experience for C++ is quite confusing and muddied.
- Generally new users start using tools that are too complicated for their use case
- It is my opinion that starting out with an IDE like visual studio is too complicated for new users.

Windows

- Visual Studio with the C++ native package
- Visual Studio Build Tools(These are terminal Tools that are a subset of the full version). Suitable for Resource constrained systems as it only includes the command line tools)
- In windows the compiler does not work out of the box in the regular terminal, instead we usually have to get a specially configured terminal, which is part of the visual studio install
- These days Windows does have an SSH server built in and it is possible to access these terminal tools remotely. This is far better than having to use Remote Desktop which can be problematic over the internet
- I personally currently just download the build tools. for those who like to use VS-Code
- The build tools are needed in addition to the visual studio code

Powershell vs CmD

- The shells available on windows, though similar with regards to how command processing works on the surface, do operate differently because of their slightly different origins
- Powershell is not purely text based, instead its underlying technology is dotnet object, though it is accessed through a text-like api
- cmd is based on DOS, which was a shell that was light weight as opposed to the UNIX shells

MacOS

- XCode
- I am not on the mac platform, but as it is Unix based, generally It should be pretty similar to linux

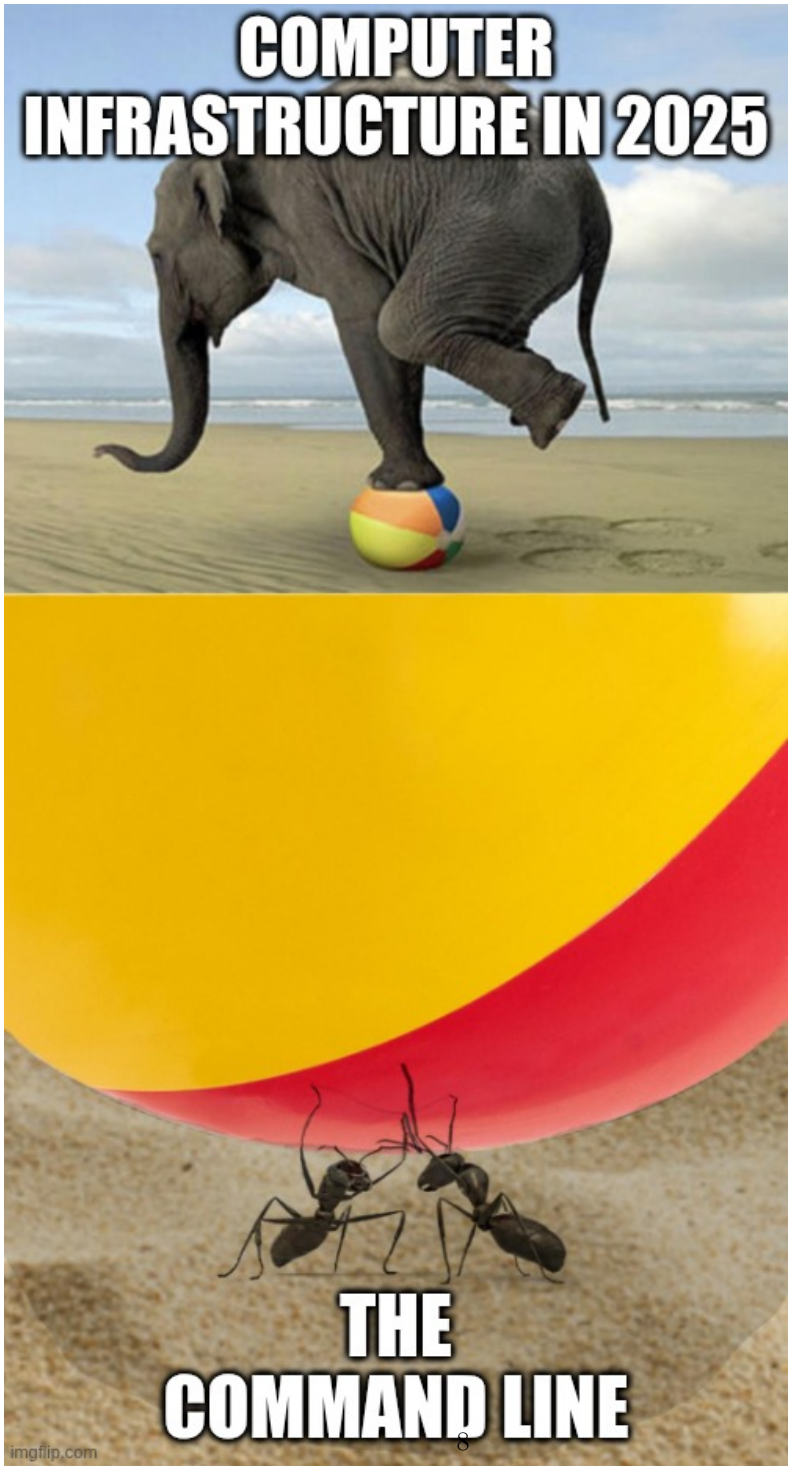
Linux

- Generally access to the compilers is easier here as the tools are available as packages, which in some distros comes preinstalled
- Qt-Creator
- Emacs, Neovim... etc.

- I am primarily a linux user, though I am quite familiar with windows and a bit with unix. The concepts that I am trying to get across today should be universal, but I am mentioning these just FYI, and as a starting point for self-study later.

Why The Command Line?

TEXT



- Text is the foundational interface to the heart of our computers.
- Because we code in text, our compilers and other tools work with text.
- Our Operating Systems with graphics as well as Integrated Development Environments try to shield us from The Text interfaces
- This may be ideal for Speed and project management in certain Professional Development Environments
- However, It is not ideal for Learning Environments.

Access

- Text being easy to transport, gives opportunities for remote access, work and collaboration

Playground

- We learn language... and other stuff by getting to play.
- Before a child starts to speak, they "play" with the sounds they hear.
- So to learn a language, you need to have a safe "playground"
- When playing, we need to slow down and allow our brains to play with concepts and ideas
- I would go so far as to argue that Using a Command Line should be part of the foundational education experience. It is as important as Math.

Cost

- It costs less to run than GUIs.

- This matters when access to power is an issue like remote areas and places where access to power is challenging
- With SSH and the lightweight nature of the terminal, It is possible to significantly reduce travel costs as you don't have to travel all over the place just to access a computer.
- This has been quite a gamechanger for me personnally. I am able to work remotely, and can easily access work environments from virtually anywhere.

Power

- The command line is the realm of experts.
- However, to become an expert, you need to play
- GUI based IDEs are indeed convenient, however, the convenience requires knowledge of what they do for you to make the most out of them.

Why not IDES

- IDE -> INTEGRATED DEVELOPMENT ENVIRONMENT.
- It is a integration of distinct components
- This integration is great for speeding up development for those already familiar with the individual components
- For learning, we need to dis-integrate in order to play with the various components

But... why C++?

- Historically, C, which is the foundation of C++ was very tightly coupled to the concepts of the unix operating system.
- C++ is the pathway to better constructs of managing software complexity while still being backward compatible with C.
- Because of the ability to link to system software.
- Many APIs that are on our systems only provide a C++ interface and others only have a C API
- Is open source really open without literacy of the language?
- Its really about access. If I go to another country, China for example, I need to have some knowledge of how to speak Chinese to survive.
- Code literacy is also an important factor to better code use and applications

About The Command Line

- Lets go through some very important terms in relation to the command line and programming in general

Operating System

- When we start up our computer systems, there exist special programs whose main purpose is to start up other programs.
- These are our operating systems like windows, linux, macOS etc.
- They have a very similar background and have several similarities

The Terminal

- The terminal historically was a printer and a keyboard. The printer being a “printable” surface

- This graduated to a screen and a keyboard which could be connected over long distances to a mainframe or mini-computer

- The keyboard and the screen are the most prominent and consistent components of our computer systems.
- The terminal as it exists today, is generally in programs that "Emulate" the behaviour of the traditional terminals but run in todays graphical environments.
- We can't talk about the command line without first talking about the terminal.

Command Line/SHELL

- This is a program that is used to input keystrokes as text and output text to the terminal
- Text Based user interface where Commands are typed in, interpreted by the "operating system", processes created, leading to Joy or agony
- The command line is a text based way interface to accessing the computer.
- It was the primary way of accessing the computer resources, like programs, files actions, etc before the advent of the graphical usre interface

How it works

- You generally start with a prompt, which is the systems way of showing you that its waiting for input

  $

- The first word you enter is the name of a program "known" by the operating system
- Any words after the first word are "Arguments" to the program
- The output from the program is displayed after running the program
- The prompt is displayed again after the previous program is done

General Shell Types

- I will try to classify the "Shell" types available on different platforms

Unix Based(POSIX-Compatible)

- These are available on Unix like systems, which includes macs

DOS Based(CMD)

- This is generally only available on windows and was its primary shell

Powershell

- The powershell is a newer type of Shell, primarily available on windows, but also available on other platforms
- This one is kind of different from the other types, it may use text, but the text refers to "DotNet" like constructs

How to Get Help

- For Unix(POSIX) based shells, the convention for most programs is to use the Argument "-h" or "–help" after the program name
- For DOS based shells, the convention is the "/?" Argument after the program name
- Powershell based shells, the convention is the "Help" cmdlet followed by the name of the cmdlet

Text

But what is text?

- Computers, are basically just big calculators, Calculate being a synonym for Compute.
- Its all just ONEs and ZEROs

- We create groups of these ones and zeros(bits) into groups, the most common being a byte which is 8 bits

ASCII

- A Standard called ASCII(American Standard Code for Information Interchange), was developed to symbolize how to represent letters, numbers and characters.
- This standard was actually initially designed for Transmiting telegrams over large distances.
- ASCII, consists of 128 values for representing a set of visible characters and other values that are used for communicating different values in numerical form.
- ASCII, represents each character as a 7 bit value

Unicode

- As ASCII was developed in "English" speaking nations, its character set was sufficient for that language
- However, in order to accomodate other writing systems, character types, different approaches were used
- These all did retain the idea of ASCII
- It got really messy... but that isn't the focus of this talk
- In the internet world of today we use "UTF-8" for representing text in our terminal emulators and many other applications

Text in C++?

- According to cppreference.com, A C++ program is a sequence of text files(typically header and source files)that contain declarations
- These undergo, translation to become and executable program which is executed when the C++ implementation calls its main function.
- The smallest data type in C and C++ is the char, which is an 8 bit value also known as a byte.
- The char is able to handle the ASCII character codes and is quite ubiquitous in alot of "programs" in the wild.

What is a Program?

- Our computers, don't really run on text, the CPUs understand a fixed set of instructions, represented in Ones and Zeros, that control and coordinate these instructions many times a second, to produce the wonderful computing experience we know and love.
- When we write our code in text, in the C++ language(there are other languages, but we don't talk about those in a C++ conference), it gets stored in a file(which is just a sequence of ones and zeroes stored on disk).

- This file is "passed" through the command line to another program called the compiler, which does the translation into the kind of instructions understood by the computer system.

Compiled vs Translated

- There are many languages in existence, but they generally fall into two broad categories, Compiled and Translated.
- C++ is Compiled language, it has to be full translated and put together before being run on the computer.
- There exist languages that do this translation in "real" time. These are called translated languages. This translation happens each time the program is run.

Command Line C++

The C++ compiler

- The Compiler is a text based program that reads in "plain text files" with C++ in them
- It creates output files which are runnable programs
- If unsuccessful in compilation it gives output explaining what happened

Getting to know your compiler

- The compiler has many options that control the process of taking your text program to machine code
- Here is a start to exploring what is possible

GCC and/or clang

```
$ c++ --help
$ info g++
$ clang++ --help
```

MSVC(cl.exe)

```
$ cl.exe /?
```

Your First C++ program

Hello World using GCC

```
1    // File => helloworld.cpp
2    int main(){
3        std::cout<<"Hello World!\n";
4    }
```

```
$ c++ helloworld.cpp
```

```
helloworld.cpp: In function ‘int main()’:
helloworld.cpp:3:10: error: ‘cout’ is not a member of ‘std’
    3 |     std::cout << "Hello World\n";
      |          ^~~~
helloworld.cpp:1:1: note: ‘std::cout’ is defined in header ‘<iostream>’; this is probably fixable by adding ‘#include <iostream>’
  +++ |+#include <iostream>
    1 |
```

Figure 1: We Get a Compiler Error

- The compiler will give an explanation as to why it didn't succeed in producing a valid program
- This is in text form... (which is important for later)
- Also Note the File name and Position information of the error
- IDEs and other tools use this information to aid in development

Hello Again World

```cpp
// File => helloworld.cpp
#include <iostream>

int main(){
    std::cout<<"Hello World!\n";
}
```

```
$ c++ helloworld.cpp
```

- The program now compiles.
- Because we didn't specify the output name, a default name "a.out" is given to the resulting executable

What about Windows?

- Microsoft has its own implementation of the C++ compiler.
- To achieve the same as above we use the the "cl.exe" program

Hello World on Windows

```cpp
// File => helloworld.cpp
#include <iostream>

int main(){
    std::cout<<"Hello World!\n";
}
```

```
$ cl.exe helloworld.cpp
```

- The MSVC compiler cl.exe does not follow the a.out convention
- It creates an executable with the same name as the source file

Stages of Compilation

The Compiler is actually does its work in a few different stages

1. Pre-Process
2. Assemble
3. Compile
4. Link

Controlling the Compiler Process

- Until now, we have been passing the name of a text file with the extension .cpp to the compiler
- The compiler assumes this to be an executable and tries to do the full process
- With Some compiler flags it is possible to state where you want the process to stop

Pre-Process

- This is usually(by convention) done with the 'E' argument

```
$ c++ -E helloworld.cpp # gcc and clang
```

```
$ cl.exe /E helloworld.cpp # msvc
```

- It produces alot of text output to the console.
- This is because it copies any "included" file combines it with our source file to create the actual file to be compiled
- The `#include <iostream>` file is really big

- We can save the result of the PreProcessing stage to a file with Shell Redirection

```
`$ c++ -E helloworld.cpp > helloworld_preprocessed.cpp # gcc and clang`
```

- With `cl.exe` we actually have a specific flag to output to a file

```
`$ cl.exe /E helloworld.cpp  # msvc`
```

- We could even use the "PreProcessed" files as input for the next stage as though they were regular source files

Assemble

- This source file and stops the processing at the Assembly Stage converting it into Assembly Code, which is a human readable form of machine code

```
$ c++ -S libgreet.cpp
```

```
$ cl.exe /Fa helloworld.cpp
```

- This produces a file with the `.s` suffix and the same name as the source file which contains the assembly code of the translation unit
- I don't know if there is a way to stop at this stage with the MSVC compiler.

Compile

- This translates the translation unit into machine code
- By convention this uses the /c flag
- The output of this stage is an "Object" file by convention with the suffix `.o`

```
$ c++ -c helloworld.cpp
```

```
$ cl.exe /c helloworld.cpp
```

- Note that the `libgreet.cpp` file compiles, i.e. there is no output after compilation

Link

- Linking is actually done by a separate program, that is invoked internally by our compiler.
- In this stage, one or more object files are put together, "linked" to form a program in this case.
- The output of this stage may also be an "LIBRARY" archive containing the combination of several object files which can be linked to in the link process of another executable.
- This is one of the "Super Powers" of C++. The ability to combine with "Other" peoples object files and Libraries

Libraries

What are they?

- When we compile, the compiler produces an object file for each translation unit.
- A program can be formed when there exists a `main` function in one of the compiled objects.
- However, we can have translation units that just have functions and other data.
- The object files can be "put" together in a library of object files that a program with a main function can link to to produce a working program.
- This is a foundation for code sharing, which is one of the strong points of C++, especially with the backward compatibility with C

- A lot of operating system facilities are available as C APIs, but can be used by C++ programs with higher level constructs for better code organization.

Headers

- The functionality in object files is incomplete without the use of headers.
- Headers are generally used to store "Declarations", which means pieces of code that just introduce an entity.
- The Header files are included into several translation units with the expectation that somewhere else there is a translation unit that contains the actual definition of the declared things.

Example

- Lets Refactor our hello world program, we'll move the functionality into a function called greet in another source file(with a .cpp extension)and try compiling

```cpp
// File => libgreet.cpp
#include <iostream>

void greet(){
    std::cout<<"Hello World From Greet!\n"
}
```

```
$ c++ libgreet.cpp
```

```
/bin/ld: /lib/../lib64/crt1.o: in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

Figure 2: No main Function

- Runnable programs require a main function
- When we try to compile without any arguments, the compiler tries to create an executable

Back to Refactoring the Greeting program

hellogreet.cpp

```cpp
// file => hellogreet.cpp
int main(){
    greet();
}
```

```
$ c++ hellogreet.cpp librarygreet.cpp
```

- We can compile more than one translation unit at a time and the "Compiler" infrastructure should link them together
- note that this time, our "main" program does not #include the header.
- This is done in the library.

Figure 3: We get a compiler error!

- This is because the `hellogreet.cpp` has no idea about the greet function.
- `.cpp` files are Translation units and are "translated" independently

Using A Header file

- Lets Create a Header file, with the extension `.hpp` that contains a "Declaration" of the function that is shared between the two translation units

```cpp
// file => librarygreet.hpp

void greet();
```

- We will include the line `#include "librarygreet.hpp"` into both the `librarygreet.hpp` and the `helloworldgreet.cpp` files

```cpp
//file => librarygreet.cpp
#include <iostream>
#include "librarygreet.hpp"
 ...
```

```cpp
//file => helloworldgreet.cpp
#include "librarygreet.hpp"
 ...
```

```
$ c++ helloworldgreet.cpp librarygreet.cpp
```

Standard Library

- The Concept of linking various compiled object files together applies with the standard library as well.

```cpp
//hello_world.cpp
#include <iostream>

int main(){
    std::cout<<"Hello standard library world\n";
}
```

- In this case the `#include <iostream>` pre-processor directive pulls in the `iostream` header into the translation unit
- The iostream header includes declarations for including names like `std::cout` into various programs.
- The definition of what exactly `std::cout` means is known by the "compiler" and is linked in automatically during the compilation process.

External Library

- Lets try and use an external library for formatting
- We will use the `fmt` library to Format our greeting

```cpp
// file => librarygreetfmt.cpp
#include <fmt/format.h>
#include "librarygreet.hpp"
void greet(){
 fmt::print("Hello Format World\n");
}
```

- To do this, we `#include <fmt/format.h>`, which is one of the Headers for the fmtlib API

- Lets now compile the greet application as before, but this time with the `librarygreetfmt.cpp` file

`` `$ c++ librarygreetfmt.cpp helloworldgreet.cpp` ``

Figure 4: Linker Error

- To resolve this, we need to pass into the the compiler an argument that tells us about the Library Binary to include in the linking process to enable us to get the object files for the library to be joined with our object files

```
`$ c++ -lfmt librarygreetfmt.cpp helloworldgreet.cpp`
```

- This assumes that the 'fmt' library is located in the standard location where the compiler would be able to find it.
- If it is not available in the system, it would have to be installed for this to work.

Real World Programming

- In the real world, we find ourselves dealing with quite a number of source files
- We need other tools to manage the process of co-ordinating the "finding" of headers and "libraries"

Build Systems

- These are programs that take as input "Instructions" on what to Compile and Link in order to produce working programs
- Examples of these are:
  - Makefiles
  - MsBuild
  - Ninja
  - XCode
  - etc.
- These tend to be system specific, but in some cases can be cross-platform
- In general they do serve the same purpose.
- They are however, usually command line tools, which are used by IDEs.
- Just like I am recommending getting to know the compiler, It would be good to also take a swing at getting to know these build systems

CMake

- C++ is available in many platforms.
- One way that we have for making our code cross-platform is to use a common program for generating the build-systems for different systems is to take what is common and describe it in a platform indendent way, like a formula
- This formula can then be used to generate the "recipes" for compiling and distributing software in various other systems.

- CMake is also used in several IDEs in the background but is also a command line program

Overview of Using CMake

Create a CMake Project

- To use cmake you need to create a directory for your project, we can use the current one, and in that directory create file with the name `CMakeLists.txt`
- We can put in the examples that we have done so far in a cmake file to see how it comes together

```
1
2  CMAKE_MINIMUM_REQUIRED(VERSION 3.28)
3
4  project(mycppproject LANGUAGES CXX)
5
6  add_executable(helloworld)
7  target_sources(helloworld PRIVATE helloworld.cpp)
8
9  add_library(librarygreet)
10 target_sources(librarygreet PRIVATE librarygreet.cpp)
11
12 add_executable(helloworldgreet)
13 target_sources(helloworldgreet PRIVATE helloworldgreet.cpp)
14 target_link_libraries(helloworldgreet PRIVATE librarygreet)
15
16 find_package(fmt REQUIRED CONFIG)
17
18 add_executable(helloworldgreetfmt)
19 target_sources(helloworldgreetfmt PRIVATE helloworldgreet.cpp librarygreetfmt.cpp)
20 target_link_libraries(helloworldgreetfmt PRIVATE fmt::fmt)
```

Configuring a CMake Project

- Using the "recipe" above we can now tell cmake which build system we would like to use to do the compilation and linking for us
- It is generally considered good practice to do the "building" in a separate directory to keep thing organized. We can tell cmake where we would like the "object" files to go.
- So far we have been doing everything in the same directory and it can get messy for bigger projects
- We need to first create the directory.

```
$ mkdir build_directory
```

- We then tell cmake to build the project there and tell it our selected Buildsystem depending on the desired system.
- I will use Ninja in this case, as it is cross plaform. *(It needs to be installed on the system)

```
$ cmake -B build_directory -G Ninja
```

Building the project

- Building refers to executing the compiler commands required to produce the programs and/or libraries we specified in the CMakeLists.txt file
- In the build_directory we can usually execute the `Build System` commands and the Build system will execute the recipe created by CMake to do the compilation.
- However, we can also use CMake to do this for us, with the following command.

```
$ cmake --build build_directory
```

- After executing this, we should find the three programs ready in the build_directory

Compiler Explorer

- Compiler Explorer, as many of you are aware is a web based platform for accessing different C++ compilers and tools
- It is a web API, that provides access to compilers working on "out there in the cloud"
- If you do not have good internet access, however, it is possible to install locally
- However, even if you do not have good internet access, It should be possible to program using the terminal only tools to achieve the same result

Next Steps and Further Resources

- C++Weekly - list of videos for learning
- CMake
- New windows Shell
- Neovim for C++

Conclusion

- The slides above were really just an example of how to play on the command line using command line tools instead of IDEs.
- There are many more areas to explore, This was just a roadmap of how to get to play with some of the core flags of the command line

- The command line is an accessible way to learn to program with C++. Take advantage of it
- It is also an important skill that should be taught in more places, particularly schools and early learning.