

Attribution/License

- Original Materials developed by Mike Shah, Ph.D. (www.mshah.io)
- This slideset and associated source code may not be distributed without prior written notice

Please do not redistribute slides/source without prior written permission.



CppIndiaCon 2024
The C++ festival of India.

Aug
23 & 24



Getting Started with **Modern C++**

-- A Tour of Features

-- in C++

with Mike Shah

Gold Sponsors

think-cell

10:00 - 11:00 IST Sat, August 24, 2024
(12:30 AM - 1:30 AM EDT - Sun. August 25, 2024)

60 minutes with Q&A
Introductory/Intermediate Audience

Social: [@MichaelShah](https://twitter.com/MichaelShah)

Web: mshah.io

Courses: courses.mshah.io

 **YouTube**

www.youtube.com/c/MikeShah

<http://tinyurl.com/mike-talks>

Your Tour Guide for Today

Mike Shah

- **Current Role:** Teaching Faculty at **Yale University** (Previously Teaching Faculty at Northeastern University)
 - **Teach/Research:** computer systems, graphics, geometry, game engine development, and software engineering.
- **Available for:**
 - **Contract work** in Gaming/Graphics Domains
 - e.g. tool building, plugins, code review
 - **Technical training** (virtual or onsite) in Modern C++, D Language, and topics in Software Design, Performance, or Graphics APIs
- **Fun:**
 - Guitar, running/weights, traveling, video games, and cooking are fun to talk to me about!



Web

www.mshah.io



[YouTube
https://www.youtube.com/c/MikeShah](https://www.youtube.com/c/MikeShah)

Non-Academic Courses

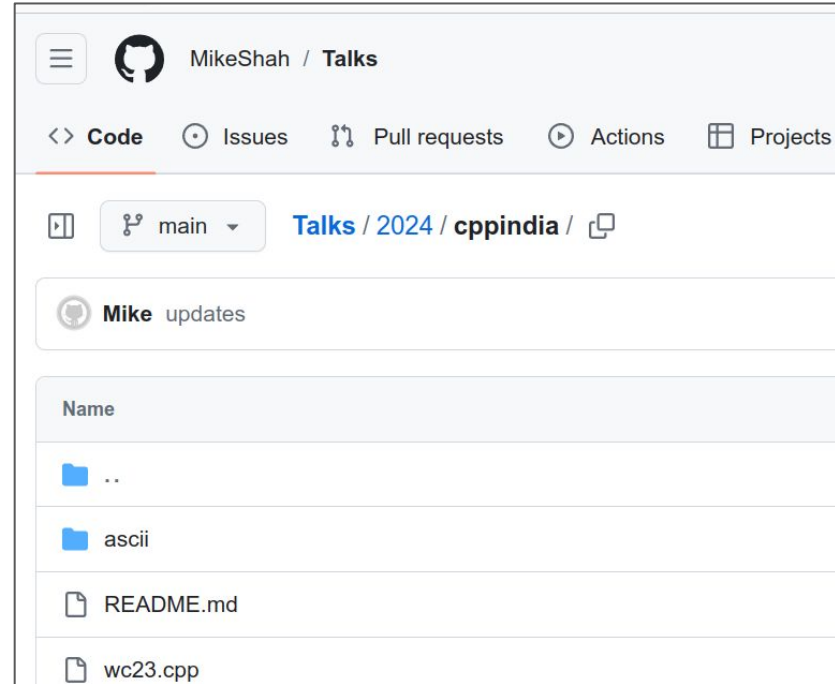
courses.mshah.io

Conference Talks

<http://tinyurl.com/mike-talks>

Code and Slides for the talk

- Code Located here:
<https://github.com/MikeShah/Talks/tree/main/2024/cppindia>
- Slides posted after conference at:
 - www.mshah.io
- Live coding the examples from this (if any) posted at:
 - www.youtube.com/c/MikeShah





Talk Abstract: Modern C++ is a powerful and expressive language used by millions of programmers. The large C++ ecosystem of libraries and tooling allows C++ developers to build scalable and fast systems on multiple platforms utilizing techniques from multiple programming paradigms to conquer many domains in the software industry. That's the elevator pitch at the least -- so how does one get started in Modern C++ and utilize all these features of the language?

In this talk, I take audience members on a step-by-step journey to understand the fundamental pieces of the C++ ecosystem focusing primarily on new features of the Modern C++ language. In this talk we will focus on new language features and the STL and pointing out key parts (array, span, smart_pointers, ranges, concepts, and thread) and how these features improve upon legacy C++ code. Audience members will leave this talk excited about using a modern version of the language, and what features and libraries can enhance their experience with C++ going forward.

Prerequisites - I assume some C++ experience for this talk ^(1/2)

- If you have not programmed C++ you may benefit from this free Quick Start on YouTube
 - <https://www.youtube.com/playlist?list=PLvv0ScY6vfd-R9N-vIDXdd4HO9IYATixJ>
- If you have not programmed C++, you may consider my course longer course on C++
 - <https://courses.mshah.io/courses/quick-start-introduction-to-modern-c-image-loader>

[C++ Quick Start Part 1/4] Quick First Time C++ Introduction to iostream and vector in 23 minutes
Mike Shah • 3.2K views • 2 years ago

[C++ Quick Start Part 2/4] Classes and Compiling multiple files (program structure) in 25 minutes
Mike Shah • 1.6K views • 2 years ago

[C++ Quick Start Part 3/4] Read, write, and parse files(fstream, string, & stringstream) in 31 min.
Mike Shah • 6.9K views • 2 years ago

[C++ Quick Start Part 4/4] References, Pointers, and Dynamic Memory Allocation in 30 minutes
Mike Shah • 1.4K views • 2 years ago

Free Playlist on Getting Started with C++ in about 2 hours

<https://www.youtube.com/playlist?list=PLvv0ScY6vfd-R9N-vIDXdd4HO9IYATixJ>

Prerequisites - I assume some C++ experience for this talk (2/2)

- If you have not programmed C++

you may benefit from my

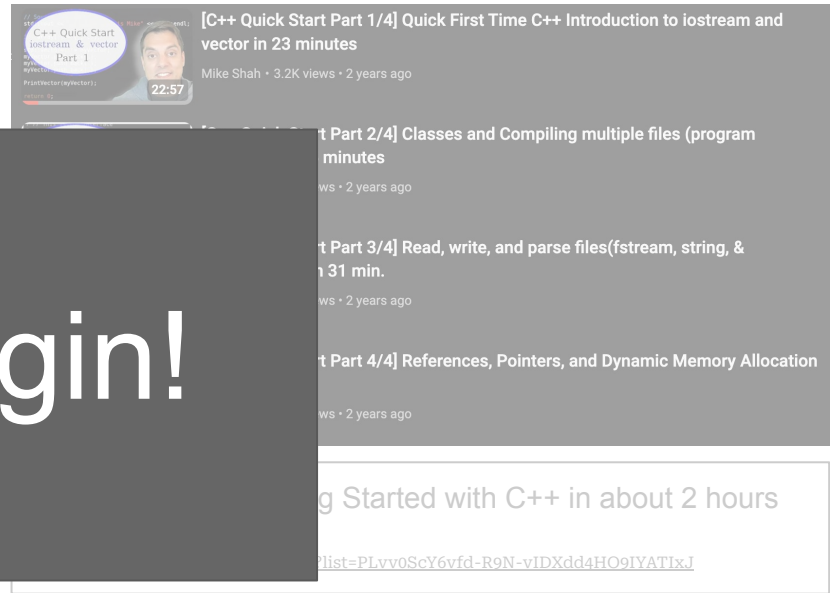
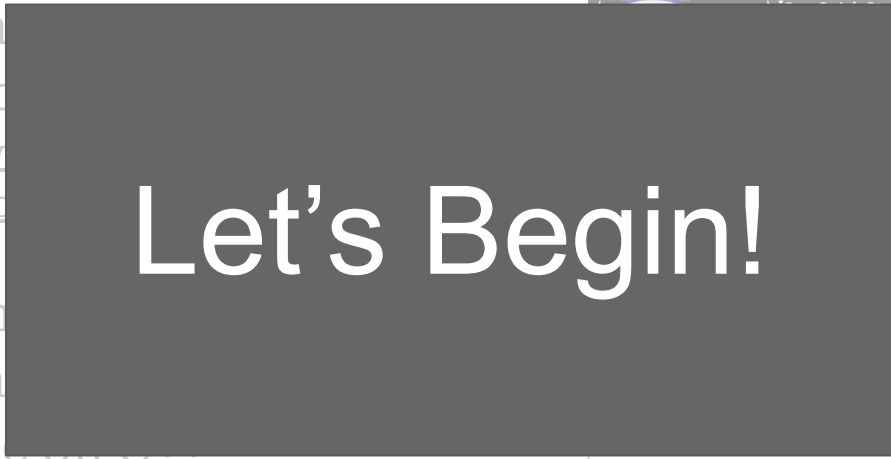
Quick Start course

- <https://www.youtube.com/watch?v=PLvv0ScY6vfd-R9N-viDXdd4HO9iYATixJ>

- If you have more experience

you may consider my longer course on C++

- <https://courses.mshah.io/courses/quick-start-introduction-to-modern-c-image-loader>



An Evolving Language (1/3)

- The C++ Language and standard library is evolving every 3 years
- As an example of the evolution -- observe some different ways we can iterate through a collection over time.
 - These are just examples using various 'for-loop' constructs
 - Whether a simple for-loop
 - Using an algorithm
 - Using a for-ranged loop
 - Or using other abstractions
- All of these techniques can be used and abstracted upon -- this is a brief example of language evolution

```
1 #include <iostream>
2 #include <vector>
3 #include <ranges>
4 #include <algorithm>
5
6 int main(){
7     std::vector c{1,2,3,4,5};
8
9     // C-Style loop
10    // Nothing wrong with this...
11    std::cout << "C-Style Loop" << std::endl;
12    for(size_t i=0; i < c.capacity(); ++i){
13        std::cout << c[i] << std::endl;
14    }
15
16    // C++98 style loop with iterators
17    std::cout << "C++98 Style iterator Loop" << std::endl;
18    for(auto it = c.begin();
19         it != c.end();
20         ++it){
21        std::cout << *it << std::endl;
22    }
23
24    // Iterators also will give us access to std::algorithm
25    // std::for_each available since at least c++98.
26    std::cout << "std::algorithm Loop" << std::endl;
27    std::for_each(c.begin(), c.end(), [](auto &e){
28        std::cout << e << std::endl;
29    });
30
31    // C++11 provides 'ranged-for' loop
32    std::cout << "ranged-for Loop" << std::endl;
33    for(const auto& e : c){
34        std::cout << e << std::endl;
35    }
36
37    // Range(s) views provided in C++20
38    std::cout << "ranges view" << std::endl;
39    for(const auto& e: std::views::all(c)){
40        std::cout << e << std::endl;
41    }
42
43 }
```


An Evolving Language (2/3)

- The C++ Language and standard library is evolving every 3 years
- As an example observe some ways to iterate through time.
 - These are just different ways to write a 'for-loop' construct
 - Whether using a range
 - Using a range
 - Using a range
 - Or using other abstractions
- All of these techniques can be used and abstracted upon -- this is a brief example of language evolution

So where does that leave you as a beginner getting started with **modern C++**?

```
1 #include <iostream>
2 #include <vector>
3 #include <ranges>
4 #include <algorithm>
5
6 int main(){
7     std::vector c{1,2,3,4,5};
8
9     // C-Style loop
10    // Nothing wrong with this...
11    std::cout << "C-Style Loop" << std::endl;
12    for(size_t i=0; i < c.capacity(); ++i){
13        std::cout << c[i] << std::endl;
14    }
15
16    // loop with iterators
17    std::cout << "C++98 Style iterator Loop" << std::endl;
18    it = c.begin();
19    it != c.end();
20    ++it){
21        << *it << std::endl;
22    }
23
24    // This also will give us access to std::algorithm
25    // methods available since at least c++98.
26    std::cout << "std::algorithm Loop" << std::endl;
27    std::algorithm(c.begin(), c.end(), [](auto &e){
28        << e << std::endl;
29    });
30
31    // This describes 'ranged-for' loop
32    std::cout << "std::ranges::ranged-for Loop" << std::endl;
33    for(const auto& e : c){
34        std::cout << e << std::endl;
35    }
36
37    // Range(s) views provided in C++20
38    std::cout << "ranges view" << std::endl;
39    for(const auto& e: std::views::all(c)){
40        std::cout << e << std::endl;
41    }
42
43 }
```

An Evolving Language (3/3)

- The C++ Language and standard library is evolving every 3 years
- As an example observe some time to iterate through
 - These are just 'for-loop' comparisons
 - Whether
 - Using a
 - Using a
 - Or using other abstractions
- All of these techniques can be used and abstracted upon -- this is a brief example of language evolution

So where does that leave you as a beginner getting started with **modern C++**?

The goal of this talk is to help you navigate what is important, and what to focus on when starting C++

```
1 #include <iostream>
2 #include <vector>
3 #include <ranges>
4 #include <algorithm>
5
6 int main(){
7     std::vector c{1,2,3,4,5};
8
9     // C-Style loop
10    // Nothing wrong with this...
11    std::cout << "C-Style Loop" << std::endl;
12    for(size_t i=0; i < c.capacity(); ++i){
13        std::cout << c[i] << std::endl;
14    }
15
16    // loop with iterators
17    std::cout << "C++98 Style iterator Loop" << std::endl;
18    it = c.begin();
19    it != c.end();
20    ++it){
21        << *it << std::endl;
22    }
23
24    // Also will give us access to std::algorithm
25    // functions available since at least c++98.
26    std::cout << "std::algorithm Loop" << std::endl;
27    std::algorithm(c.begin(), c.end(), [](auto &e){
28        out << e << std::endl;
29    });
30
31    // describes 'ranged-for' loop
32    std::cout << "std::ranged-for Loop" << std::endl;
33    for(const auto& e : c){
34        std::cout << e << std::endl;
35    }
36
37    // Range(s) views provided in C++20
38    std::cout << "ranges view" << std::endl;
39    for(const auto& e: std::views::all(c)){
40        std::cout << e << std::endl;
41    }
42
43 }
```

5 Questions

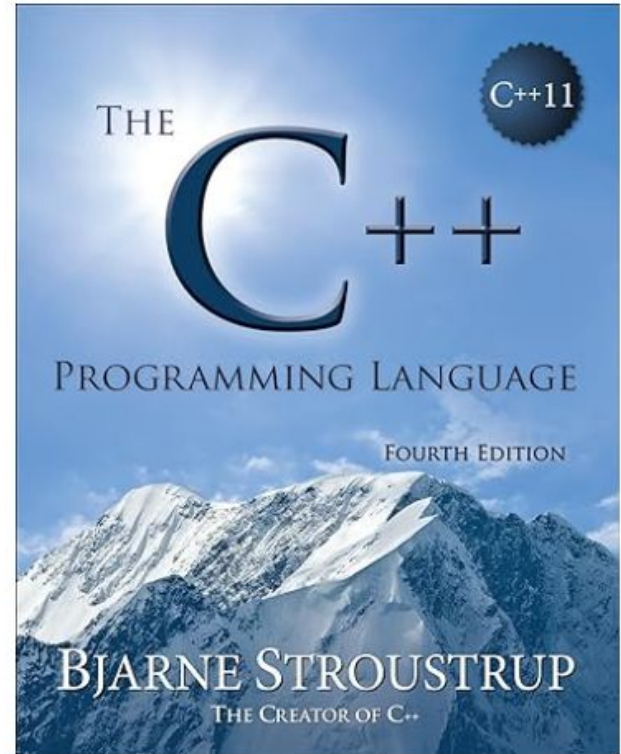
- This talk is titled: “Getting Started with Modern C++”
 - The landscape of computer programming has changed a great deal since I was a beginner
 - So today I really want to think hard about what it means to get started and be a beginner, and what questions I might ask.
 - So today’s talk will try to answer 5 questions
1. Why the C++ Programming Language?
 2. What is the C++ Ecosystem?
 3. Is C++ evolving for the future, and is it worth it if I invest time now?
 4. Can you show me some Modern C++?
 5. What Modern C++ features should I focus on?

Question #1: Why the C++ Programming Language?



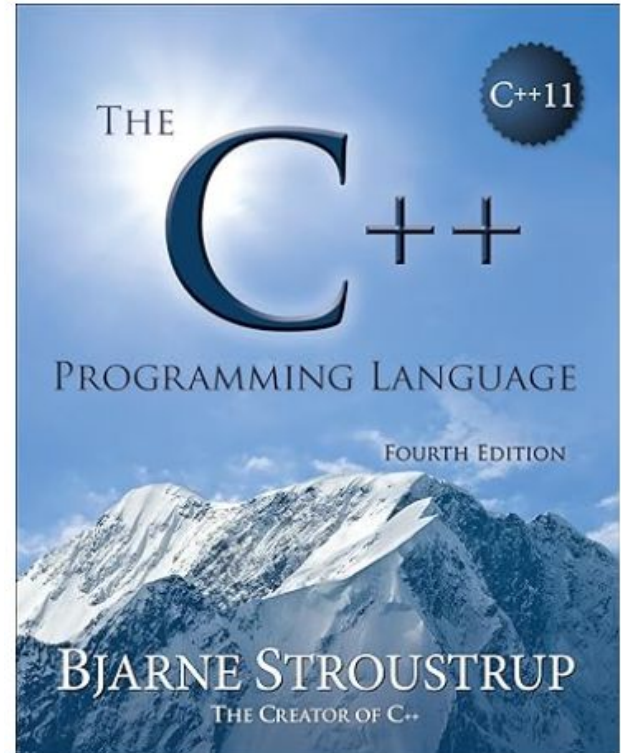
Why the C++ Programming Language? (1/2)

- C++ is a powerful, versatile, and expressive language
 - Powerful:
 - This means in terms of speed, compiled C++
 - Versatile:
 - C++ can be found with effective use cases in nearly every domain: finance, games, business, web, networks, automotive, robotics, etc.
 - Expressive:
 - You choose the best abstraction to use in the language: procedural, object-oriented, data-oriented, generic, functional, etc.
 - C++ attempts to provide zero-cost abstractions (or [tries to provide](#) tools to create your own)



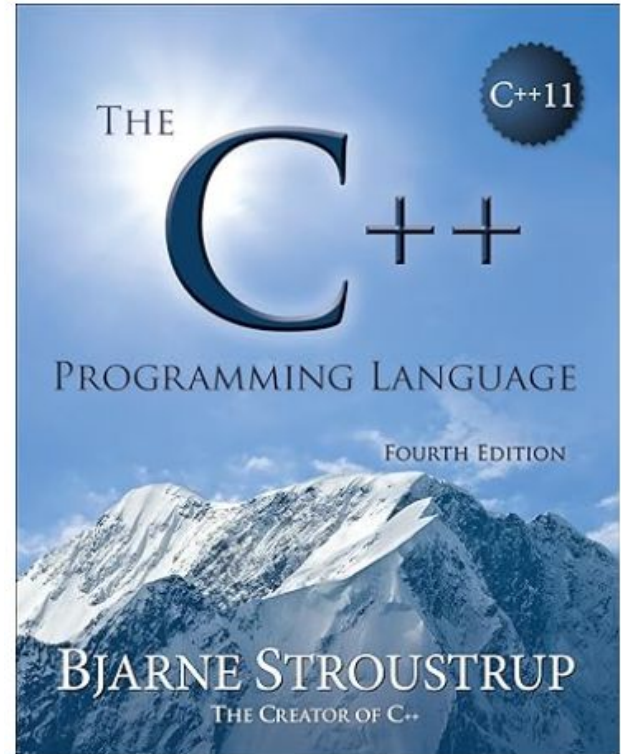
Why the C++ Programming Language? (2/2)

- The language has also been proven the last 40+ years -- and continues to power the software world.
 - Whether directly (i.e. applications written in C++) or otherwise using libraries created in C++ (e.g. Python bindings or wrappers of C++ code)



Why choose C++ for a new project in 2024?

- Performance
 - Zero-overhead Principle [[cppreference](#)]
 - i.e. “Don’t pay for what you do not use” and the language “values efficiency”
- Ecosystem
 - Tooling (IDEs, static and dynamic analysis, linters, build systems, etc.) is widely invested in.
- Portability
 - C++ can be run on the web (wasm or emscripten), embedded (small devices, game consoles, robotics), and desktop devices.
- Certification
 - Various C++ compilers are certified for domains in safety critical domains (e.g. automotive and aircraft)



C++ Superpowers [from <https://www.stroustrup.com/C++.html>]

C++ is a general-purpose programming language with a bias towards systems programming that

- is [a better C](#)
 - a. It had time to learn!
- supports [data abstraction](#)
 - a. Information hiding, etc.
- supports [object-oriented programming](#)
 - a. From its Simula roots
- supports [generic programming](#).
 - a. For reusable algorithms, data structures, and even compile-time programming



<https://www.stroustrup.com/>

C++ Superpowers [from Me]

I'll also go further to add that C++:

- is a productive language
 - a. lots of library support beyond the standard library
- is evolving to support safer data abstractions
 - a. smart pointers, `std::optional`, ranges, etc.
- supports multiple programming paradigms including:
 - a. generic programming / Object-Oriented / Procedural
 - b. functional-style programming
 - i. lambda's, `const`, `std::function`, `map (std::transform)`,
`reduce (std::accumulate)`, `filter (std::remove/copy/find_if)`
- supports compile-time execution
 - a. `constexpr`, meta-programming through templates



What is **hard** about learning the C++ Language?

From my experience teaching and training, there are two things:

1. Understanding the ‘ecosystem’
 - a. This includes:
 - i. compilation process, static and dynamic linking
 - ii. How to structure large-scale and long-lived projects [[Lakos](#)]
2. Being disciplined about what features you use
 - a. When you are first learning the language, it can be difficult to try to do things ‘the right way’
 - i. ‘The right way’ will depend on your domain.
 - ii. C++ does not have the same guardrails as languages like DLang, Rust, Ada, Swift, etc. for safety.
 1. This is a double-edged sword

Question #2: What is the
C++ Ecosystem?



C++ Ecosystem of Tools

- Compiler
 - g++, clang++, msvc, etc.
- Linker
 - ld
- IDE / Text Editor
 - IDE's: Visual Studio, CLion, XCode, etc.
 - Text Editor: [VIM](#), VSCode, Sublime, etc.
- Build Systems
 - Make, Ninja, etc.
 - Cmake - meta-build tool
- Tooling
 - Static Analysis (Linters like cppcheck, Misra Check, etc.)
 - Dynamic Analysis (Asan, tsan, UBSan)
 - Debugging Tools (GDB, LLDB, [UDB](#) (Time Travel Debugging), etc.)
 - Profilers (e.g. perf)
- Key Libraries to your project
 - Testing frameworks (Catch, Google Test)
 - Domain specific frameworks (e.g. GUI library, graphics API, sound API, threading library, etc.)

C++ Ecosystem of Tools

- **Compiler**
 - g++, clang++, msvc, etc.
- **Linker**
 - ld
- **IDE / Text Editor**
 - IDE's: Visual Studio, CLion, XCode, etc.
 - Text Editor: [VIM](#), VSCode, Sublime, etc.
- **Build Systems**
 - Make, Ninja, etc.
 - Cmake - meta-build tool
- **Tooling**
 - Static Analysis (Linters like cppcheck, Misra Check, etc.)
 - Dynamic Analysis (Asan, tsan, UBSan)
 - Debugging Tools (GDB, LLDB, UDB (Time Travel Debugging), etc.)
 - Profilers (e.g. perf)
- **Key Libraries to your project**
 - Testing frameworks (Catch, Google Test)
 - Domain specific frameworks (e.g. GUI library, graphics API, sound API, threading library, etc.)

At a minimum this is all you need

1. A compiler
2. A linker
 - a. (usually 'hidden' when you're first learning)
3. A text editor

No need to complicate things and think you need more when you're starting out :)

C++ Ecosystem of Tools

- **Compiler**
 - g++, clang++, msvc, etc.
- **Linker**
 - ld
- **IDE / Text Editor**
 - IDE's: Visual Studio, CLion, XCode, etc.
 - Text Editor: VIM, VSCode, Sublime, etc.
- **Build Systems**
 - Make, Ninja, etc.
 - Cmake - meta-build tool
- **Tooling**
 - Static Analysis (Linters like cppcheck, Misra Check, etc.)
 - Dynamic Analysis (**Asan**, **tsan**, **UBsan**)
 - Debugging Tools (GDB, LLDB, UDB (Time Travel Debugging), etc.)
 - Profilers (e.g. perf)
- **Key Libraries to your project**
 - Testing frameworks (Catch, Google Test)
 - Domain specific frameworks (e.g. GUI library, graphics API, sound API, threading library, etc.)

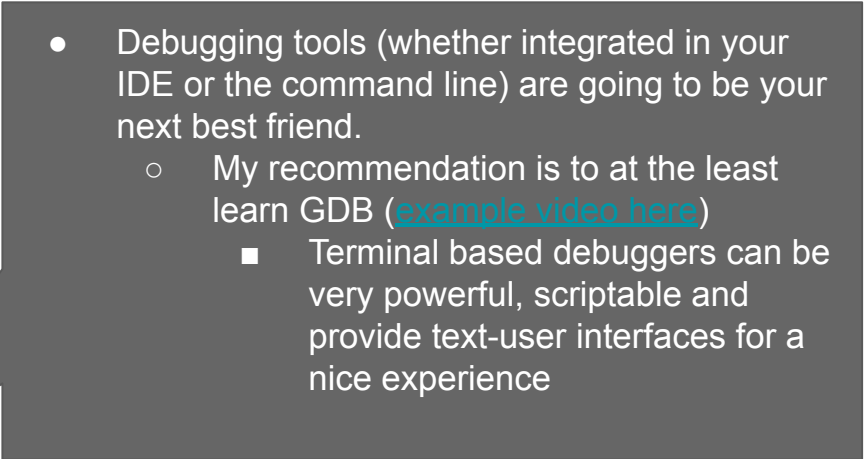
Some tips on using your compiler to make your life easier:

- Compile with:
 - `-Wall -Werror -g`
 - Use the sanitizers
 - e.g. [-fsanitize=undefined](#)
- There are nice lesser known tools (at least with g++) such as `-Weffc++` as well
 - [See video here](#)

C++ Ecosystem of Tools

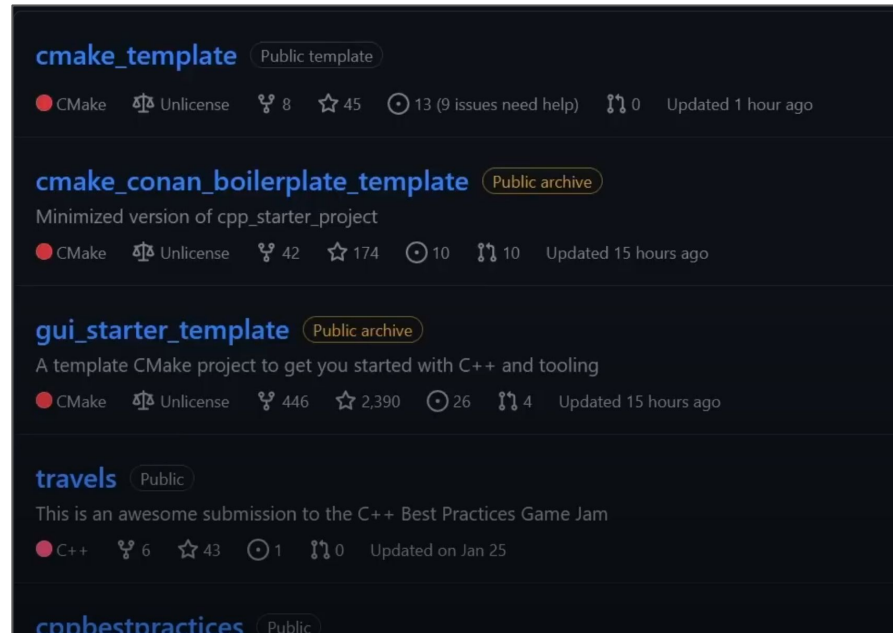
- Compiler
 - g++, clang++, msvc, etc.
 - Linker
 - ld
 - IDE / Text Editor
 - IDE's: Visual Studio, CLion, XCode, etc.
 - Text Editor: VIM, VSCode, Sublime, etc.
 - Build Systems
 - Make, Ninja, etc.
 - Cmake - meta-build tool
 - Tooling
 - Static Analysis (Linters like `cppcheck`, Misra Check, etc.)
 - Dynamic Analysis (Asan, tsan, UBSan)
 - Debugging Tools (GDB, LLDB, UDB (Time Travel Debugging), etc.)
 - Profilers (e.g. perf)
 - Key Libraries to your project
 - Testing frameworks (Catch, Google Test)
 - Domain specific frameworks (e.g. GUI library, graphics API, sound API, threading library, etc.)
- 
- For folks just getting started, I like [cppcheck](#).
 - `cppcheck` will help detect bugs and help encourage you to use more modern constructs in your code.
 - Linux users can immediately use:
 - `sudo apt-get install cppcheck`
 - `cppcheck --enable=all *.cpp`

C++ Ecosystem of Tools

- Compiler
 - g++, clang++, msvc, etc.
 - Linker
 - ld
 - IDE / Text Editor
 - IDE's: Visual Studio, CLion, XCode, etc.
 - Text Editor: VIM, VSCode, Sublime, etc.
 - Build Systems
 - Make, Ninja, etc.
 - Cmake - meta-build tool
 - Tooling
 - Static Analysis (Linters like cppcheck, Misra Check, etc.)
 - Dynamic Analysis (Asan, tsan, UBSan)
 - **Debugging Tools (GDB, LLDB, UDB (Time Travel Debugging), etc.)**
 - Profilers (e.g. perf)
 - Key Libraries to your project
 - Testing frameworks (Catch, Google Test)
 - Domain specific frameworks (e.g. GUI library, graphics API, sound API, threading library, etc.)
- 
- Debugging tools (whether integrated in your IDE or the command line) are going to be your next best friend.
 - My recommendation is to at the least learn GDB ([example video here](#))
 - Terminal based debuggers can be very powerful, scriptable and provide text-user interfaces for a nice experience

C++ Project Template

- There exist nice templates for exercising a good chunk of a modern C++ ecosystem
 - Here is one from Jason Turner for creating a CMake project
 - The latest repository is here:
 - https://github.com/cpp-best-practices/cmake_template
- Use these as inspiration for creating your own
 - It may also be wise for you to create your own 'template' as you use C++ in your domain
 - Most existing templates will probably be good resources for discovering useful tools and libraries -- enough so that expert C++ programmers consider them as *essential* enough to put in their default templates.



C++ Weekly - Ep 376 - Ultimate CMake C++ Starter Template (2023 Updates)

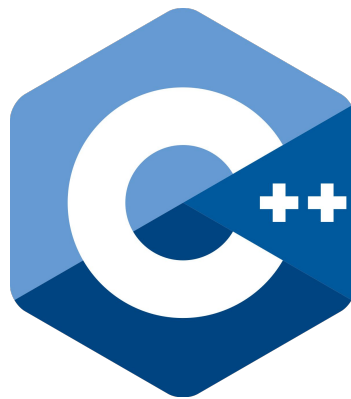
<https://www.youtube.com/watch?v=ucl0cw9X3e8>

Question #3: Is C++
evolving for the future, and
is it worth it if I invest time
now?



What's *really* Important?

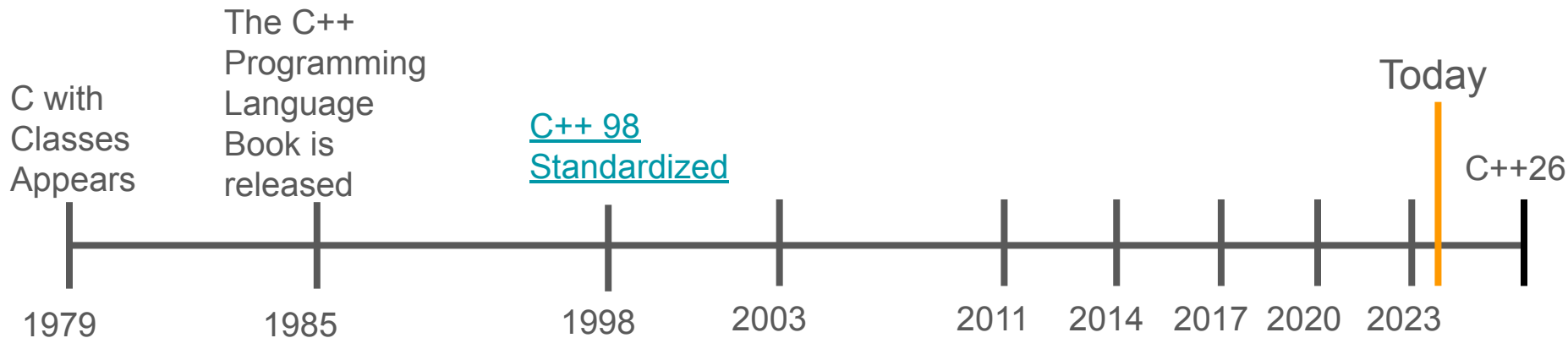
- What's really important from this timeline is that the language is evolving every 3 years
- The language (and compiler vendors) make great efforts to support backward compatibility
 - i.e. Code that you wrote in 1998 often still compiles on compilers in 2024.
 - This can be extremely valuable when you have long lived and core infrastructure!
- Is it worth the time and effort to learn C++ and the ecosystem?
 - Yes -- I do not see C++ moving away anytime soon in performance critical domains.
 - Time spent in C++ or alongside other languages will only help you become a better programmer.



C++ History and Evolution

From C++98 to C++23 and Beyond

Where the language was and where the language is going



C++ Modern History (1998-Present Day)

- 1998: -- Original ISO C++ Standard officially adopted (“C++98”).
 - 776 pages.
- 2003: TC1 (“Technical Corrigendum 1”) published (“C++03”).
 - A minor revision in 2003, primarily bug fixes for C++98.
- 2005: TR1 (Library “Technical Report 1”) published.
 - 14 likely new components for the standard library.
- 2011: “C++0x” ratified ⇒ “C++11”.
 - This was a **major update** that modernized C++ to its current form in September of 2011
 - 1353 pages.
 - C++ now evolves on an ambitious 3-year schedule.
- 2014: C++14 ratified.
 - 1372 pages -- largely minor improvements to 2011 features
- 2017: (Will visit shortly)
- 2020: (Will visit shortly)
- 2023: (Will visit shortly)
- 2026:
 - Next C++ Standard currently in the works
- Note:
 - Around 2022, what are dubbed as successor languages (cpp2, Carbon, etc.) also started exploring the evolution of the language

C++ standards		
Year	ISO/IEC Standard	Informal name
1998	14882:1998 ^[34]	C++98
2003	14882:2003 ^[35]	C++03
2011	14882:2011 ^[36]	C++11, C++0x
2014	14882:2014 ^[37]	C++14, C++1y
2017	14882:2017 ^[38]	C++17, C++1z
2020	14882:2020 ^[17]	C++20, C++2a
TBA	14882:2024	C++23, C++2b
TBA		C++26, C++2c

Question #4: Can you show me some Modern C++?

(i.e. 2003 C++ versus Modern C++)



A Classic Example - wc (word count)

- A good way to learn or practice your C++ is to implement various command line programs
 - If you're not familiar with the 'wc' program, it is a helpful utility for counting words, lines, and bytes in a file
- Let's look at a simple program and compare C++98 versus C++20 (and beyond) code
- Note:
 - If you're more senior and listening to this talk, perhaps this is an interesting 'take home' projects for interviewing candidates -- though I have many more thoughts on tech interviews :)

```
WC(1) BSD General Commands Manual
      WC(1)

NAME
  wc -- word, line, character, and byte count

SYNOPSIS
  wc [-c|lmw] [file ...]

DESCRIPTION
  The wc utility displays the number of lines, words, and bytes contained in each input file, or standard input (if no file is specified) to the standard output. A line is defined as a string of characters delimited by a <newline> character. Characters beyond the final <newline> character will not be included in the line count.

  A word is defined as a string of characters delimited by white space characters. White space characters are the set of characters for which the iswspace(3) function returns true. If more than one input file is specified, a line of cumulative counts for all the files is displayed on a separate line after the output for the last file.
```

wc98 (word count, C++ 98 standard)

- (wc98.cpp)
 - `g++ -g -Wall -std=c++98 wc98.cpp -o wc98`
- Here's the core implementation of a word count (wc) program.
 - It works, and is written in pure C++98 code.

```
12 struct wcInfo{
13     size_t bytes;
14     size_t lines;
15     size_t words;
16     // 0-initialize everything in constructor
17     wcInfo(){
18         bytes=0;
19         lines=0;
20         words=0;
21     }
22 };
```

```
24 /// Helper function for returning file size
25 size_t GetFileSize(const char* filename){
26     std::ifstream myFile(filename, std::ios::ate | std::ios::in | std::ios::binary);
27     return myFile.tellg();
28 }
```

```
30 /// Main function for retrieving the file size
31 wcInfo wc(const char* filename){
32     wcInfo result;
33
34     // Open file for input
35     std::ifstream myFile(filename, std::ios::in);
36
37     // Iterate through each line and each word
38     // using stringstream to parse 'whitespace'
39     if(myFile.is_open()){
40         result.bytes = GetFileSize(filename);
41         std::string line;
42         while(std::getline(myFile, line)){
43             ++result.lines;
44             std::stringstream s(line);
45             std::string word;
46             while(s >> word){
47                 ++result.words;
48             }
49         }
50     }else{
51         std::cerr << "File could not be read: " << filename << std::endl;
52     }
53
54     return result;
55 }
```


wc23 (word count, C++ 23 standard)

- (wc23.cpp)
 - `g++ -g -Wall -std=c++23 wc23.cpp -o wc23`
- Same program as before, but several improvements using Modern C++ 11 through C++23 features
- I'll highlight a few for comparison

```
14 struct wcInfo{
15     size_t lines{0};
16     size_t words{0};
17     size_t bytes{0};
18 };
19
20 // Main function for retrieving the file size
21 wcInfo wc(const char* filename){
22     wcInfo result{};
23     // Open file for input
24     std::ifstream myFile(filename, std::ios::in);
25
26     // Helper lambda function for counting words in line
27     auto wordsInLine = [](auto line){
28         size_t count{};
29         std::stringstream s(line);
30         std::string word{};
31         while(s >> word){
32             ++count;
33         }
34         return count;
35     };
36
37     // Iterate through each line and each word
38     // using stringstream to parse 'whitespace'
39     if(myFile.is_open()){
40         result.bytes = std::filesystem::file_size(filename);
41         std::string line{};
42         while(std::getline(myFile, line)){
43             ++result.lines;
44             result.words += wordsInLine(line);
45         }
46     }else{
47         std::cerr << "File could not be read: " << filename << std::endl;
48     }
49
50     return result;
51 }
```

```
60     // structured binding
61     auto [lines, words, bytes] = wc(argv[1]);
62     std::println("\t{}\t{}\t{}", lines, words, bytes);
```

```

12 struct wcInfo{
13     size_t bytes;
14     size_t lines;
15     size_t words;
16     // 0-initialize everything in constructor
17     wcInfo(){
18         bytes=0;
19         lines=0;
20         words=0;
21     }
22 };

```

Observe on the the right side we can more concisely default initialize members. This is useful for Plain Old Data (POD) data types

features

- I'll highlight a few for comparison

```

14 struct wcInfo{
15     size_t lines{0};
16     size_t words{0};
17     size_t bytes{0};
18 };
19
20 // Main function for retrieving the file size
21 wcInfo wc(const char* filename){
22     wcInfo result{};
23     // Open file for input
24     std::ifstream myFile(filename, std::ios::in);
25
26     // Helper lambda function for counting words in line
27     auto wordsInLine = [](auto line){
28         size_t count{};
29         std::stringstream s(line);
30         std::string word{};
31         while(s >> word){
32             ++count;
33         }
34         return count;
35     };
36
37     // Iterate through each line and each word
38     // using stringstream to parse 'whitespace'
39     if(myFile.is_open()){
40         result.bytes = std::filesystem::file_size(filename);
41         std::string line{};
42         while(std::getline(myFile, line)){
43             ++result.lines;
44             result.words += wordsInLine(line);
45         }
46     }else{
47         std::cerr << "File could not be read: " << filename << std::endl;
48     }
49
50     return result;
51 }

```

```

60     // structured binding
61     auto [lines, words, bytes] = wc(argv[1]);
62     std::println("\t{}\t{}\t{}", lines, words, bytes);

```

wc23 (word count, C++ 23 standard)

```
24 /// Helper function for returning file size
25 size_t GetFileSize(const char* filename){
26     std::ifstream myFile(filename, std::ios::ate | std::ios::in | std::ios::binary);
27     return myFile.tellg();
28 }
```

- No `std::filesystem` in C++98, so need hack for reading file size, versus a 1-line function call in Modern C++

features

- I'll highlight a few for comparison

```
14 struct wcInfo{
15     size_t lines{0};
16     size_t words{0};
17     size_t bytes{0};
18 };
19
20 /// Main function for retrieving the file size
21 wcInfo wc(const char* filename){
22     wcInfo result{};
23     // Open file for input
24     std::ifstream myFile(filename, std::ios::in);
25
26     // Helper lambda function for counting words in line
27     auto wordsInLine = [](auto line){
28         size_t count{};
29         std::stringstream s(line);
30         std::string word{};
31         while(s >> word){
32             ++count;
33         }
34         return count;
35     };
36
37     // Iterate through each line and each word
38     // using stringstream to parse 'whitespace'
39     if(myFile.is_open()){
40         result.bytes = std::filesystem::file_size(filename);
41         std::string line{};
42         while(std::getline(myFile, line)){
43             ++result.lines;
44             result.words += wordsInLine(line);
45         }
46     }else{
47         std::cerr << "File could not be read: " << filename << std::endl;
48     }
49
50     return result;
51 }
```

```
60     // structured binding
61     auto[lines, words, bytes] = wc(argv[1]);
62     std::println("\t{}\t{}\t{}", lines, words, bytes);
```

wc23 (word count, C++ 23 standard)

```
wcInfo results = wc(argv[1]);
std::cout <<
    << results.lines << "\t"
    << results.words << "\t"
    << results.bytes << "\t" << std::endl;
```

- No structured binding
 - Both a pro and con -- need to still be careful of ordering of struct
 - Easy however to retrieve local variables
- Can use auto in C++11 and beyond
 - Potentially more 'refactorable code'
 - Use type when there is a need to be explicit
- Can use std::println in C++23 with format strings

```
14 struct wcInfo{
15     size_t lines{0};
16     size_t words{0};
17     size_t bytes{0};
18 };
19
20 // Main function for retrieving the file size
21 wcInfo wc(const char* filename){
22     wcInfo result{};
23     // Open file for input
24     std::ifstream myFile(filename, std::ios::in);
25
26     // Helper lambda function for counting words in line
27     auto wordsInLine = [](auto line){
28         size_t count{};
29         std::stringstream s(line);
30         std::string word{};
31         while(s >> word){
32             ++count;
33         }
34         return count;
35     };
36
37     // Iterate through each line and each word
38     // using stringstream to parse 'whitespace'
39     if(myFile.is_open()){
40         result.bytes = std::filesystem::file_size(filename);
41         std::string line{};
42         while(std::getline(myFile, line)){
43             ++result.lines;
44             result.words += wordsInLine(line);
45         }
46     }else{
47         std::cerr << "File could not be read: " << filename << std::endl;
48     }
49
50     return result;
51 }
```

```
60 // structured binding
61 auto [lines, words, bytes] = wc(argv[1]);
62 std::println("\t{}\t{}\t{}", lines, words, bytes);
```

wc23 (word count, C++ 23 standard)

```
// Iterate through each line and each word
// using stringstream to parse 'whitespace'
if(myFile.is_open()){
    result.bytes = GetFileSize(filename);
    std::string line;
    while(std::getline(myFile,line)){
        ++result.lines;
        std::stringstream s(line);
        std::string word;
        while(s >> word){
            ++result.words;
        }
    }
}else{
    std::cerr << "File could not be read: " << filename << std::endl;
}
```

- Local lambda functions
 - Simplifies algorithm
 - Measure to see if performance is impacted
 - Use lambda if function only needed once, or otherwise as a tool for encapsulation

```
14 struct wcInfo{
15     size_t lines{0};
16     size_t words{0};
17     size_t bytes{0};
18 };
19
20 // Main function for retrieving the file size
21 wcInfo wc(const char* filename){
22     wcInfo result{};
23     // Open file for input
24     std::ifstream myFile(filename, std::ios::in);
25
26     // Helper lambda function for counting words in line
27     auto wordsInLine = [](auto line){
28         size_t count{};
29         std::stringstream s(line);
30         std::string word{};
31         while(s >> word){
32             ++count;
33         }
34         return count;
35     };
36
37     // Iterate through each line and each word
38     // using stringstream to parse 'whitespace'
39     if(myFile.is_open()){
40         result.bytes = std::filesystem::file_size(filename);
41         std::string line{};
42         while(std::getline(myFile,line)){
43             ++result.lines;
44             result.words += wordsInLine(line);
45         }
46     }else{
47         std::cerr << "File could not be read: " << filename << std::endl;
48     }
49
50     return result;
51 }
```

```
60 // structured binding
61 auto[lines, words, bytes] = wc(argv[1]);
62 std::println("\t{}\t{}\t{}", lines, words, bytes);
```


wc23 (word count, C++ 23 standard)

- Full list of improvements

- struct with constructor needed
 - Need to carefully default initialize
- No `std::filesystem`, so need hack for reading file size
- No structured binding
 - Both a pro and con -- need to still be careful of ordering of struct
 - Easy however to retrieve local variables
- Local lambda functions
 - Simplifies algorithm
 - Use lambda if function only needed once, or otherwise as a tool for encapsulation
- Can use `auto` in C++11 and beyond
 - Potentially more 'refactorable code'
 - Use `type` when there is a need to be explicit
- Can use `std::println` in C++23 with format strings

```
14 struct wcInfo{
15     size_t lines{0};
16     size_t words{0};
17     size_t bytes{0};
18 };
19
20 // Main function for retrieving the file size
21 wcInfo wc(const char* filename){
22     wcInfo result{};
23     // Open file for input
24     std::ifstream myFile(filename, std::ios::in);
25
26     // Helper lambda function for counting words in line
27     auto wordsInLine = [](auto line){
28         size_t count{};
29         std::stringstream s(line);
30         std::string word{};
31         while(s >> word){
32             ++count;
33         }
34         return count;
35     };
36
37     // Iterate through each line and each word
38     // using stringstream to parse 'whitespace'
39     if(myFile.is_open()){
40         result.bytes = std::filesystem::file_size(filename);
41         std::string line{};
42         while(std::getline(myFile, line)){
43             ++result.lines;
44             result.words += wordsInLine(line);
45         }
46     }else{
47         std::cerr << "File could not be read: " << filename << std::endl;
48     }
49
50     return result;
51 }
```

```
60     // structured binding
61     auto [lines, words, bytes] = wc(argv[1]);
62     std::println("\t{}\t{}\t{}", lines, words, bytes);
```

wc23 (word count, C++ 23 standard)

- (wc23.cpp)
 - `g++ -g -Wall -std=c++23 wc23_1.cpp -o wc23_1`
- Within our word counting function, we can make a further revision by using `std::tuple`
 - This removes any temporary or ‘one use’ structs where we need to return multiple values.
 - Furthermore, our function at this point is fully encapsulated
 - The function is merely an interface generating an output given inputs, and few details are otherwise exposed to the caller.

```
14 std::tuple<size_t, size_t, size_t> wc(const char* filename){
15     size_t lines{};
16     size_t words{};
17     size_t bytes{};
```

```
43     return std::make_tuple<size_t, size_t, size_t>(lines, words, bytes);
```

Question #5: What Modern C++ features should I focus on?



What Modern C++ features should I focus on?

- The real trick is to work iteratively
 - For me, I often write code in the most simple way.
 - Then I figure out if there is some abstraction I can use to make code:
 - Safer
 - More understandable
 - More amenable to change (while designing)
 - More generic (if a library)
 - Over time your ‘muscle memory’ will help you exercise features correctly (for your definition of ‘correct’ on your project) otherwise on an earlier iteration

```
12 struct wcInfo
13     size_t bytes;
14     size_t lines;
15     size_t words;
16     // @-initialize everything in constructor
17     wcInfo() {
18         bytes=0;
19         lines=0;
20         words=0;
21     };
22 };
23
24 // Helper function for returning file size
25 size_t GetFileSize(const char* filename){
26     std::ifstream myFile(filename, std::ios::ate | std::ios::in | std::ios::binary);
27     return myFile.tellg();
28 }
29
30 // Main function for retrieving the file size
31 wcInfo wc(const char* filename){
32     wcInfo result;
33
34     // Open file for input
35     std::ifstream myFile(filename, std::ios::in);
36
37     // Iterate through each line and each word
38     // using stringstream to parse 'whitespace'
39     if(myFile.is_open()){
40         result.bytes = GetFileSize(filename);
41         std::string line;
42         while(std::getline(myFile, line)){
43             ++result.lines;
44             std::stringstream s(line);
45             std::string word;
46             while(s >> word){
47                 ++result.words;
48             }
49         }
50     }else{
51         std::cerr << "File could not be read: " << filename << std::endl;
52     }
53
54     return result;
55 }
```

word counter C++98 code

```
14 struct wcInfo{
15     size_t lines{0};
16     size_t words{0};
17     size_t bytes{0};
18 };
19
20 // Main function for retrieving the file size
21 wcInfo wc(const char* filename){
22     wcInfo result{};
23
24     // Open file for input
25     std::ifstream myFile(filename, std::ios::in);
26
27     // Helper lambda function for counting words in line
28     auto wordsInLine = [](auto line){
29         size_t count{};
30         std::stringstream s(line);
31         std::string word{};
32         while(s >> word){
33             ++count;
34         }
35         return count;
36     };
37
38     // Iterate through each line and each word
39     // using stringstream to parse 'whitespace'
40     if(myFile.is_open()){
41         result.bytes = std::filesystem::file_size(filename);
42         std::string line{};
43         while(std::getline(myFile, line)){
44             ++result.lines;
45             result.words += wordsInLine(line);
46         }
47     }else{
48         std::cerr << "File could not be read: " << filename << std::endl;
49     }
50     return result;
51 }
52
53 // structured binding
54 auto [lines, words, bytes] = wc(argv[1]);
55 std::println("\t{}\t{}\t{}", lines, words, bytes);
```

word counter C++20 code

A Very Quick Overview

- To the right you'll see a summary of just the new features in C++11!
- You can find most all of the C++11 updates here:
 - <https://en.cppreference.com/w/cpp/11>
- I now want to focus on a few of the new language and library features from C++11 to 23 that I think every C++ programmer should know in this short duration
 - Note: This is not an exhaustive list -- but a few that I find myself often using!

- auto and decltype
- defaulted and deleted functions
- final and override
- trailing return type
- rvalue references
- move constructors and move assignment operators
- scoped enums
- constexpr and literal types
- list initialization
- delegating and inherited constructors
- brace-or-equal initializers
- nullptr
- long long
- char16_t and char32_t
- type aliases
- variadic templates
- generalized (non-trivial) unions
- generalized PODs (trivial types and standard-layout types)
- Unicode string literals
- user-defined literals
- attributes
- lambda expressions
- noexcept specifier and noexcept operator
- alignof and alignas
- multithreaded memory model
- thread-local storage
- GC interface (removed in C++23)
- range-for (based on a Boost library)
- static_assert (based on a Boost library)

Headers

- <array>
- <atomic>
- <cfenv>
- <chrono>
- <cstdint>
- <condition_variable>
- <cstdlib>
- <uchar>
- <forward_list>
- <future>
- <initializer_list>
- <mutex>
- <random>
- <ratio>
- <regex>
- <scoped_allocator>
- <system_error>
- <thread>
- <tuple>
- <typeindex>
- <type_traits>
- <unordered_map>
- <unordered_set>

A Few Examples to Focus on

- While I cannot highlight everything in this duration -- here are several features I enjoy in Modern C++:
 - C++11
 - nullptr
 - auto
 - ranged-for
 - constexpr
 - Smart Pointers
 - Move Semantics
 - Lambda's
 - thread
 - unordered_map
 - C++20
 - Span
 - Ranges

nullptr (Introduced in -std=c++11)

- Purpose:
 - Eliminate ‘surprises’ where nullptr can be treated as an integer -- clear when you are testing for ‘null’ versus testing for ‘0’
- When to Use:
 - Anywhere you previously used ‘NULL’
- Example Notes:
 - See Core Guideline:
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#es47-use-nullptr-rather-than-0-or-null>

```
1 // @file nullptr.cpp
2 // Compile and run: g++ -std=c++23 nullptr.cpp -o prog && ./prog
3 #include <print>
4 #include <vector>
5
6
7 void PointerParam(int* value){ std::print("PointerParam called"); }
8 void IntegerParam(int value){ std::print("IntegerParam called"); }
9
10 int main(){
11
12     // NULL is effectively '0' (zero).
13     // These are all 'equivalent'
14     // 'pGood' is a nullptr, which is more 'searchable'
15     // NULL may however be useful in a 'C-style API'
16     int* pGood = nullptr;
17     int* pOld = NULL;
18     int* pBad = 0;
19     int integer = 0;
20
21     if(integer ==0){ } // Allowed
22
23     // Not allowed -- caught by type-system
24     // if(integer == nullptr){}
25
26     PointerParam(0);
27     PointerParam(NULL);
28     PointerParam(nullptr);
29
30     IntegerParam(0);
31     // IntegerParam(NULL); -- Some compiler *may* allow with warning -- even worse!
32     //IntegerParam(nullptr); -- Error!
33
34     return 0;
35 }
```

auto (Introduced in -std=c++11)

- Purpose:
 - Placeholder for a type that will be deduced later
- When to Use:
 - Useful when typing a long type would be redundant
 - Useful for 'generic code'
 - Useful for making code more malleable
 - **Not useful** if you need to be very specific about type
- Example Notes:
 - 'auto' greatly simplifies generic programming (see lines 8-10) -- effectively this is the equivalent of a template.
 - [See video for more](#)

```
1 // @file auto.cpp
2 // Compile and run: g++ -std=c++23 auto.cpp -o prog && ./prog
3 #include <print>
4 #include <vector>
5
6 // 'auto' useful for supporting 'generic' programming
7 // Note: Still need to add 'qualifiers' like 'const'
8 auto sum(const auto a, const auto b){
9     return a + b;
10 }
11
12 int main(){
13
14     // Not a good use of auto, unclear if short, long, int, etc
15     // auto integer = 7;
16     std::vector v{1,3,5,7,9};
17
18     // Use of 'auto' here useful to deduce underlying type
19     // in collection
20     for(auto elem: v){
21         std::println("{} ",sum(elem,1));
22     }
23
24     // 'auto' useful for 'redundant' or 'long' declarations
25     // std::vector<int>::iterator start= v.begin();
26     auto start = v.begin(); // This is fine
27
28     return 0;
29 }
```

ranged-for (Introduced in -std=c++11)

- Purpose:
 - Iterate through a collection
- When to Use:
 - When your intent is to iterate through an entire range using iterators
- Example Notes:
 - Observe in for loop the use of '&'
 - Observe in the bottom-right that the ranged-for is transformed to use iterators
 - [Video for more](#)

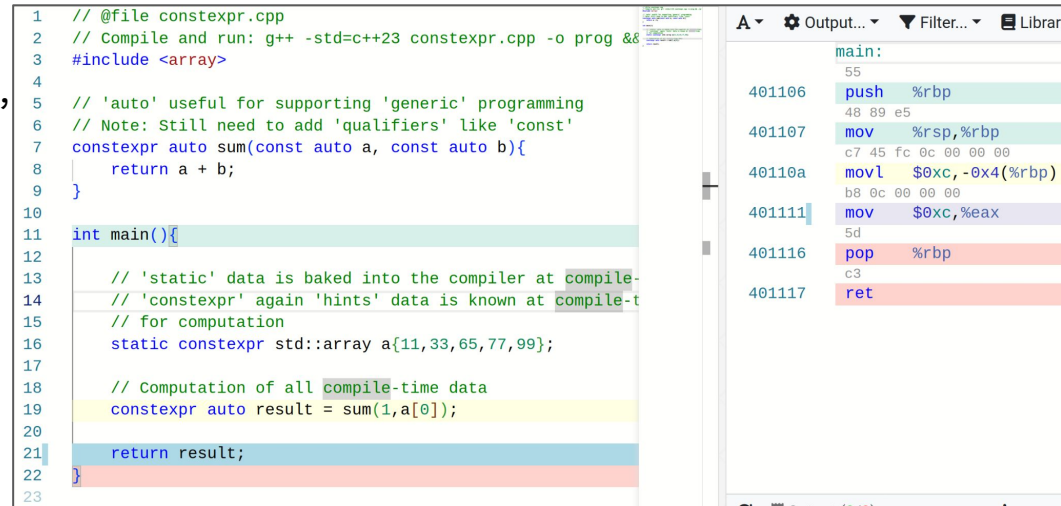
```
1 // @file rangedfor.cpp
2 // Compile and run: g++ -std=c++23 rangedfor.cpp -o prog && ./prog
3 #include <print>
4 #include <vector>
5
6 struct Student{
7     long mID;
8     long mGrade;
9 };
10
11 int main(){
12
13     // Use initializer list to populate vector.
14     // Explicitly supply the 'type' so we don't interpret as a 'pair'
15     std::vector<Student> students = {{123,100},
16                                     {124,99}};
17
18     // The '&' is important in the ranged-for loop so that we do not
19     // make a 'copy'. Generally for types less than size of a pointer
20     // on your architecture this matters.
21     // Note -- you can still break out of loops if needed.
22     for(auto& elem : students){
23         std::println("{} - {}",elem.mID, elem.mGrade);
24     }
25
26     return 0;
27 }
```

```
mike@mike-MS-7B17:features$ g++ -std=c++23 rangedfor.cpp -o prog && ./prog
123 - 100
124 - 99
```

```
std::vector<Student, std::allocator<Student> > students = std::vector<Student, std::allocator<Student>
{
    std::vector<Student, std::allocator<Student> > & __range1 = students;
    std::_wrap_iter<Student *> __begin1 = __range1.begin();
    std::_wrap_iter<Student *> __end1 = __range1.end();
    for(; std::operator!=(__begin1, __end1); __begin1.operator++()) {
        Student & elem = __begin1.operator*();
        std::println(std::basic_format_string<char, long &, long &>("{} - {}"), elem.mID, elem.mGrade);
    }
}
return 0;
```

constexpr (Introduced in -std=c++11)

- Purpose:
 - Further enable 'compile-time' computations
- When to Use:
 - When you want something 'const' and also useable (if possible) at compile-time
 - Can enable very powerful meta-programming
- Example Notes:
 - This program really just outputs a '12' (0xC)
 - [More introduction here](#)



```
1 // @file constexpr.cpp
2 // Compile and run: g++ -std=c++23 constexpr.cpp -o prog &&
3 #include <array>
4
5 // 'auto' useful for supporting 'generic' programming
6 // Note: Still need to add 'qualifiers' like 'const'
7 constexpr auto sum(const auto a, const auto b){
8     return a + b;
9 }
10
11 int main(){
12
13     // 'static' data is baked into the compiler at compile-
14     // 'constexpr' again 'hints' data is known at compile-t
15     // for computation
16     static constexpr std::array a{11,33,65,77,99};
17
18     // Computation of all compile-time data
19     constexpr auto result = sum(1,a[0]);
20
21     return result;
22 }
23
```

The assembly output on the right shows the following instructions for the `main` function:

```
main:
55
401106 push %rbp
48 89 e5
401107 mov %rsp,%rbp
c7 45 fc 0c 00 00 00
40110a movl $0xc,-0x4(%rbp)
b8 0c 00 00 00
401111 mov $0xc,%eax
5d
401116 pop %rbp
c3
401117 ret
```


Smart Pointers (Introduced in -std=c++11)

- Purpose:
 - Wrap a 'pointer' for safer C++ code and focus on 'ownership' and 'lifetime' of allocated memory
 - If you make everything `std::unique_ptr` -- you effectively have *much* safer code!
- When to Use:
 - Almost always use `std::unique_ptr` if only one owner
 - Consider `shared_ptr` if managing complicated lifetimes
 - Consider `std::weak_ptr` when non-owner needs access when pointer is not null or otherwise expired.
 - May need regular raw pointers when interfacing with C APIs
- Example Notes:
 - [More on pointers](#)
 - [More on the 'why' of smart pointers](#)
 - Consider that you can also create your own 'smart pointers' as well!

```
10 // @file shared.cpp
11 #include <iostream> // I/O stream library
12 #include <memory>   // Access smart pointers
13
14 class Object{
15 public:
16     Object() { std::cout << "Constructor\n"; }
17     ~Object() { std::cout << "Destructor\n"; }
18 };
19
20
21 // Entry point to program 'main' shared
22 int main(int argc, char* argv[]){
23
24
25     // This is how we 'always' want to create shared_ptr using
26     // 'make_shared'
27     // Again, we avoid using 'new'
28     {
29         std::shared_ptr<Object> mySharedObjectPtr;
30         {
31             // Make a second pointer
32             std::shared_ptr<Object> mySharedObjectPtr2 = std::make_shared<Object>();
33             // Assign our shared pointer to another shared pointer
34             mySharedObjectPtr = mySharedObjectPtr2;
35         }
36     } // At this point, mySharedObjectPtr will 'die' but only because all of its
37     // references have gone out of scope.
38 }
39 return 0;
40 }
```

One example with only 'shared_ptr' which counts the number of 'strong references' (i.e. other shared_ptr) to it.

Move Semantics (Introduced in -std=c++11)

- Purpose:
 - For efficiency and idea of 'ownership' (safety)
- When to Use:
 - Whenever you want to avoid copies and ensure 'one' copy of data has an 'owner'
 - Think of using in tandem with `std::unique_ptr`
- Example Notes:
 - `std::move` does not 'move' -- but rather casts to an 'rvalue reference' that enables move assignment operator to be called.
 - If you do not have move constructor and/or move assignment operator, fallback will be to copy constructor and/or copy assignment operator.
 - [Video on Move](#)

```
10 // @file move.cpp
11 #include <iostream>
12 #include <utility> // std::move
13
14 class T{
15 public:
16     T(){ std::cout << "constructor" << std::endl;}
17     ~T(){ std::cout << "destructor" << std::endl;}
18     T(const T& copy){
19         std::cout << "copy constructor" << std::endl;
20     }
21     // Move constructor which transfers ownership of our
22     // object, rather than creating a new object
23     T(T&& old){
24         m_string = old.m_string;
25         old.m_string = nullptr;
26     }
27     // Move assignment operator
28     T& operator=(T&& old){
29         if(this!=&old){
30             m_string = old.m_string;
31             old.m_string = nullptr;
32         }
33         return *this;
34     }
35
36     char* m_string;
37 };
38
39 void swap(T a, T b){
40     T tmp{std::move(a)};
41     a = std::move(b);
42     b = std::move(tmp);
43 }
44
45 // Entry point to program 'main' move.cpp
46 int main(){
47     T a;
48     T b;
49     swap(a,b);
50
51     return 0;
52 }
53
54 }
55
```

Lambdas (Introduced in -std=c++11)

- Purpose:
 - Make code more beautiful -- i.e. a better syntax than a functor
 - In many ways to encourage
- When to Use:
 - Writing smaller functions that are used in algorithms
 - As 'local' functions encapsulated in a larger function
- Example Notes:
 - I've omitted the 'capture', but it may be useful to [understand functors first](#) to understand how 'state' is stored as a member of a functor.
 - i.e. Lambdas in C++11 and beyond are syntactic sugar for functors.
 - [More videos](#)

```
1 // @file lambda.cpp
2 // Compile and run: g++ -std=c++23 lambda.cpp -o prog && ./prog
3 #include <iostream>
4 #include <vector>
5 #include <algorithm>
6 #include <print>
7 #include <ranges>
8
9 int main(){
10
11     std::vector v{1,3,5,7,9};
12     // Local lambda function
13     auto square = [](int x){
14         return x*x;
15     };
16
17     // Another local lambda function
18     auto printer = [](int elem){
19         std::println("{} ",elem);
20     };
21
22     // Modify the 'view' of the data
23     // Much of the STL uses unary or binary functions to change behavior.
24     // Lambdas are thus very convenient to pass or directly instantiate.
25     //
26     // Note: Use std::ranges::copy if you want to 'keep' the data
27     std::ranges::for_each(std::views::transform(v,square),printer);
28     // Just print the original data
29     std::ranges::for_each(v,printer);
30
31     return 0;
32 }
"lambda.cpp" 33L, 849B written
```

```
mike@mike-MS-7B17:features$ g++ -std=c++23 lambda.cpp -o prog && ./prog
```

```
1
9
25
49
81
1
3
5
7
9
```

Thread (Introduced in -std=c++11)

- Purpose:
 - Usually for performance, but sometimes nature of task splits better into 'jobs'
- When to Use:
 - Again, usually for performance
 - Consider using `std::jthread` so you don't have to worry about 'joining' threads
 - Consider `std::async`
- Example Notes:
 - [More Videos](#)

```
1 // @file thread4.cpp
2 // g++-10 -std=c++20 thread4.cpp -o prog -lpthread
3 #include <iostream>
4 #include <thread> // Include the thread library
5 #include <vector>
6
7 int main() {
8
9     // This time create a lambda function
10    auto lambda = [](int x){
11        std::cout << "thread.get_id:" << std::this_thread::get_id() << std::endl;
12        std::cout << "Argument passed in:" << x << std::endl;
13    };
14
15    // Note: We now have a jthread
16    // No joins in the program
17    std::vector<std::jthread> threads;
18    // Create a collection of threads
19    for(int i=0; i < 10; i++){
20        threads.push_back(std::jthread(lambda,i));
21    }
22
23    // Continue executing the main thread
24    std::cout << "Hello from the main thread!" << std::endl;
25
26    return 0;
27 }
```

Unordered_map (Introduced in -std=c++11)

- Purpose:
 - Improve performance of map from $\log_2(n)$ to $O(1)$ in average case
 - unordered_set and other unordered containers also introduced
- When to Use:
 - Whenever ordering does not matter
 - structured bindings handy for iteration
 - Generally a drop-in replacement for std::map
- Example Notes:
 - [More Videos](#) (less trivial example)

```
1 // @file unordered_map.cpp
2 // Compile and run: g++ -std=c++23 unordered_map.cpp -o prog && ./prog
3 #include <iostream>
4 #include <string>
5 #include <unordered_map>
6 #include <map>
7 #include <print>
8
9 int main(){
10     std::unordered_map<int, std::string> database;
11     database.insert({123, "Mike"});
12     database.insert({124, "Ankur"});
13
14     if(database.contains(123)){
15         std::cout << "User 123 exists...continuing\n";
16     }
17
18     // Insert if not existing, or otherwise update
19     database[125] = "CppIndia";
20
21     // Structured binding to extract keys and values
22     for(auto const& [key, value] : database){
23         std::cout << key << " | " << value << std::endl;
24     }
25
26     // Statistics of our unordered_map's hashtable
27     std::println("bucket_count = {} and load_factor={}",
28                 database.bucket_count(), database.load_factor());
29 }
30 }
31
```

```
mike@mike-MS-7B17:features$ g++ -std=c++23 unordered_map.cpp -o prog && ./prog
User 123 exists...continuing
125 | CppIndia
124 | Ankur
123 | Mike
bucket_count = 13 and load_factor=0.23076923
```

std::span (Introduced in -std=c++20)

- Purpose:
 - Refers to an object that is contiguous (i.e. array, vector)
 - Span is a ‘fat-pointer’ with a ‘length’ and pointer to data
- When to Use:
 - It is effectively a ‘view’ of data that can read/write
 - Because it is a fat pointer, it prevents an array to decay to a pointer -- so size information is preserved
 - Useful if you want to pass any sized contiguous data structure into a function and not have it ‘owned’, but may want to read/write data.
- Example Notes:
 - [More Videos](#)

```
1 // @file span.cpp
2 // Compile and run: g++ -std=c++23 span.cpp -o prog && ./prog
3 #include <iostream>
4 #include <span>
5 #include <array>
6 #include <vector>
7
8 // Will always be a dynamic extent. You can assign (or pass) a static extent to
9 // a dynamic one, but not the other way around.
10 // If you want this function to preserve 'span' extent, then make it a template
11 // e.g.
12 // template<typename T, int size>
13 // void PrintIntData(std::span<T,size> param){
14 void PrintIntData(std::span<int> param){
15     std::cout << param.extent << std::endl;
16     if(param.extent == std::dynamic_extent){
17         std::cout << "My extent is dynamic (by default) " << std::endl;
18     }else{
19         std::cout << "Nt extent is static " << std::endl;
20     }
21
22     std::cout << "My size is:" << param.size() << std::endl << "\t";
23     for(auto& elem: param){
24         std::cout << elem << ", ";
25     }
26     std::cout << "\n\n";
27 }
28
29 int main(){
30
31     // Create fixed-size array
32     std::array<int, 5> arr = {2,4,6,8,10};
33     // Span 'wraps' the 'array'
34     std::span<int,5> mySpan{arr};
35     // The 'extent' is the static size.
36     // If the extent is 'dynamic' it will be a large integer
37     std::cout << mySpan.extent << std::endl;
38
39     // Can pass array (of any size) or span into function
40     PrintIntData(arr);
41     PrintIntData(mySpan);
42     PrintIntData(mySpan.subspan(1,3));
43
44     std::vector<int> myVector = {1,2,3,4,5,6,7};
45     PrintIntData(std::span(myVector.begin(),3));
46     PrintIntData(myVector);
47
48     return 0;
49 }
```

Ranges (Introduced in -std=c++20)

- Purpose:
 - Make code more composable and less error-prone
- When to Use:
 - Anywhere in the STL you were passing in two iterators to iterate over a collection
 - In combination with views, can perform 'lazy' evaluation (for infinite ranges, or otherwise optimization purposes)
- Example Notes:
 - You'll need C++20 for ranges, but generally speaking ranges are the path forward.

```
1 // @file lambda.cpp
2 // Compile and run: g++ -std=c++23 lambda.cpp -o prog && ./prog
3 #include <iostream>
4 #include <vector>
5 #include <algorithm>
6 #include <print>
7 #include <ranges>
8
9 int main(){
10
11     std::vector v{1,3,5,7,9};
12     // Local lambda function
13     auto square = [](int x){
14         return x*x;
15     };
16
17     // Another local lambda function
18     auto printer = [](int elem){
19         std::println("{} ",elem);
20     };
21
22     // Modify the 'view' of the data
23     // Much of the STL uses unary or binary functions to change behavior.
24     // Lambdas are thus very convenient to pass or directly instantiate.
25     //
26     // Note: Use std::ranges::copy if you want to 'keep' the data
27     std::ranges::for_each(std::views::transform(v,square),printer);
28     // Just print the original data
29     std::ranges::for_each(v,printer);
30
31     return 0;
32 }
```

"lambda.cpp" 33L, 849B written








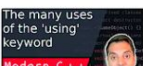
```
mike@mike-MS-7B17:features$ g++ -std=c++23 lambda.cpp -o prog && ./prog
```

```
1
9
25
49
81
1
3
5
7
9
```

Wrapping up and More Resources









Further C++ resources and training materials

- The full C++ playlist
 - <https://www.youtube.com/playlist?list=PLvv0ScY6vfd9wBflF0f6ynlDQuaeKYzyc>

	Classes Part 33 - Nested Classes Modern Cpp Series Ep. 98 Mike Shah · 3.6K views · 1 year ago
	Funcctors() - Function objects - functions with state Modern Cpp Series Ep. 99 Mike Shah · 10K views · 1 year ago
	C++ Lambdas Part 1 - Unnamed function objects (closures) in C++ Modern Cpp Series Ep. 100!!! Mike Shah · 7.5K views · 1 year ago
	Classes Part 34 - mutable keyword to override const and the M&M rule Modern cpp series Ep. 101 Mike Shah · 1.9K views · 1 year ago
	C++ Lambdas Part 2 - 'The capture' Modern cpp series Ep. 102 Mike Shah · 4.1K views · 1 year ago
	C++ Lambdas Part 3 - Capturing 'this' (Using lambda's in member functions) Modern cpp Series Ep. 103 Mike Shah · 2.3K views · 1 year ago
	C++ Handling Program arguments (argc, argv, env) Modern Cpp Series Ep. 104 Mike Shah · 4.4K views · 1 year ago
	The 'using' keyword (using namespace and alias declarations) Modern Cpp Series Ep. 105 Mike Shah · 4.1K views · 1 year ago

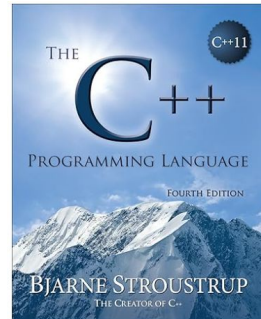
Further C++ resources and training materials

- C++ Software Design & Patterns
 - <https://www.youtube.com/playlist?list=PLvv0ScY6vfd8j-tlhYVPYgiIyXduu6m-L>
- C++ Concurrency
 - https://www.youtube.com/playlist?list=PLvv0ScY6vfd_ocTP2ZLicgqKnvq50OCXM

	Design Patterns - Command Pattern Explanation and Implementation in C++ Mike Shah • 12K views • 2 years ago
	Design Patterns - Singleton Pattern Explanation and Implementation in C++ Mike Shah • 4.4K views • 2 years ago
	Design Patterns - Factory Method Pattern Explanation and Implementation in C++ Mike Shah • 5.6K views • 2 years ago
	Design Patterns - Factory Method Pattern Adding More Power to Count Allocated Objects in C++ Mike Shah • 1.7K views • 2 years ago
	Design Patterns - The Extensible Factory Pattern in C++ Register Objects at Runtime Mike Shah • 2K views • 2 years ago
	Design Patterns - Iterator Pattern Explanation and usage with STL in C++ Mike Shah • 1.6K views • 2 years ago
	The Observer Design Pattern in C++ - Part 1 of n - A simple implementation Mike Shah • 3.8K views • 11 months ago
	The Observer Design Pattern in C++ - Part 2 of n - Extensibility and Abstraction Mike Shah • 1.7K views • 11 months ago

Summary

- C++ is a great language, and there is always room to grow with it
 - You do not need to learn every feature -- but using some of the modern ones will make your life easier
 - Doing small projects (e.g. building a game, an app, etc.) is how you will improve and see these features in context.
 - I encourage you to read as much code as you write.
- C++ I foresee continuing to be the leading language for performance-driven applications, and I think that will trend will continue to drive its evolution alongside more safety features
- C++ is here to stay, and it is a language well worth learning



{C++;}
CppIndia

CppIndiaCon 2024

The C++ festival of India.

Aug
23 & 24

{C++;}
CppIndia

Getting Started with Modern C++

Thank you CppIndia for
having me!



-- A Tour of Features

-- in C++

with Mike Shah

Social: [@MichaelShah](https://twitter.com/MichaelShah)

Web: mshah.io

Courses: courses.mshah.io

 **YouTube**

www.youtube.com/c/MikeShah

<http://tinyurl.com/mike-talks>

10:00 - 11:00 IST Sat, August 24, 2024
(12:30 AM - 1:30 AM EDT - Sun. August 25, 2024)

60 minutes with Q&A
Introductory/Intermediate Audience