C++23 Best Practices

Simple Rules with Specific Action Items for Better C++

Jason Turner

C++23 Best Practices

Jason Turner

This book is for sale at
http://leanpub.com/cpp23_best_practices

This version was published on 2024-01-01

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

For my wife, Jen.

Table of Contents

Part I: Introduction and Philosophy of Good C++

1: Introduction To The C++23 Edition

It's been about 3 years since I originally released the first edition of C++ Best Practices. At the time of release, the book did not contain much C++20 information.

I chose to release the C++20 updates (known as the 2nd Edition) for free to anyone who had purchased the Leanpub ebook version.

I considered releasing this C++23 update also as a free 3rd Edition. However, as I considered the updates that needed to occur, I decided it was time for an entirely new release of the book. This new book is needed mainly because C++23 changes many fundamental things with how we use the language (such as standard library modules). But the break from the previous version also allows me to reorganize the topics for better flow. I had avoided reorganization with any previous update to avoid confusion (so coworkers would reference item 12 and know they were all talking about the same item).

I have updated every relevant section of this book to represent how code should look in C++23, and reorganized many topics. Rest assured, this is a large update to the original C++ Best Practices book!

> I try to apply all Best Practices in every example where they are appropriate. This might make the examples more complex than they need to be, but it decreases the chances that an example will be seen out of context and incomplete.

2: Introduction To The Original Edition

My goal as a trainer and a contractor (seems to be) to work me out of a job. I want everyone to:

1. Learn how to experiment for themselves
2. Not just believe me, but test it
3. Learn how the language works
4. Stop making the same mistakes of the last generation

I'm thinking about changing my title from "C++ Trainer" to "C++ Guide." I always adapt my courses and material to the class I currently have. We might agree on a class about X, but I change it to Y halfway through the first day to meet the organization's needs.

Along the way, we experiment and learn as a group. I often also learn while teaching. Every group is unique; every class has new questions.

Many of the questions I get in classes are the same ones repeatedly to the point where I get to look like a mind reader as I anticipate the next question that will be asked.

Hence, this book, and the Twitter thread that it came from, to help spread the word on the long-standing best practices.

> I wrote the book I wanted to read. It is intentionally straightforward, short, to the point, and has specific action items.

3: About Best Practices

Best Practices, quite simply, are about

1. Reducing common mistakes
2. Finding errors quickly
3. Without sacrificing (and often improving) performance

3.1: Why Best Practices?

First and foremost, let's get this out of the way:

3.1.1: Your Project Is Not Special

If you are programming in C++, you, or someone at your company, cares about performance. Otherwise, they'd probably be using some other programming language. I've been to many companies who all tell me they are special because they need to do things fast!

Spoiler alert: they are all making the same decisions for the same reasons.

There are very few exceptions. The outliers who make different decisions: they are the organizations that are already following the advice in this book.

3.2: What's The Worst That Can Happen?

I don't want to be depressing, but let's take a moment to ponder the worst-case scenario if your project has a critical flaw.

Game
: Serious flaws lead to remote vulnerabilities or attack vectors.

Financial
: Serious flaws lead to large amounts of lost money, accelerating trades, market crash.

Aerospace
: Serious flaws lead to lost spacecraft or human life.

Your Industry
: Serious flaws lead to… Lost money? Lost jobs? Remote hacks? Worse?

3.3: Examples

Examples throughout this book use `struct` instead of `class`. The only difference between `struct` and `class` is that `struct` has all members and base classes by default `public`. Using `struct` makes examples shorter and easier to read.

3.4: Exercises

Each section has one or more exercises. Most do not have a right or wrong answer.

Exercise: Look for exercises

Throughout the following chapters, you'll see exercises like this one. Look for them!

Exercises are:

- Practical, and apply to your current code base to see immediate value.
- Make you think and understand the language a little bit deeper by doing your own research.

3.5: Links and References

I've made an effort to reference those who I learned from and link to their talks where possible. If I've missed something, please let me know.

4: Slow Down

Dozens of solutions exist in C++ for any given problem. Dozens of more opinions exist for which of these solutions are the best. Copying and pasting from one application to another is easy. Forging ahead with the solutions you are comfortable with is easy.

How many times have you said, “wow, this is going to take a complicated class hierarchy to implement this solution?” Or what about “I guess I need to add macros here to implement these common functions.”

- If the solution seems large or complex, stop.
- Now is a good time to go for a walk and ponder the solution.
- When you’re done with your walk, discuss the design with a coworker, pet, or rubber duck.

Still haven’t found a more straightforward solution you are happy with? Ask on Twitter or Slack if you can.

The key point is to not forge ahead blindly with the solutions with which you are comfortable. Be willing to stop for a minute. The older I get, the less time I spend programming, and the more time I spend thinking. In the end, I implement the solution as fast or faster than I used to and with less complexity.

5: Use AI Coding Assistants Judiciously

AI coding assistants are become ubiquitous, with nearly every IDE and tool having something built in. They appear to be rather powerful and are able to generate convincing results. However, these results are not necessarily correct. Therefore, I suggest these Best Practices for using your AI coding assistant.

1. Always double check the results you are given.
2. Use them mostly as a "smart rubber duck".
3. Always double check the results you are given.
4. Use them to "flatten the learning curve." If you ask the bot to generate an example of a certain technique or API usage, you'll likely get a meaningful answer, but one that also has some mistakes in it.
5. Always double check the results you are given.

5.1: Resources

- C++ Weekly - Ep 371 - Best Practices for Using AI Code Generators

6: C++ Is Not Magic

This section is just a reminder that we can reason about all aspects of C++. It's not a black box, and it's not magic.

If you have a question, it's usually easy to construct an experiment that helps you answer the question for yourself.

A favorite tool of mine is this simple class that prints a debug message whenever a special member function is called.

Figure 1. Understanding object lifetime tool

```
import std;

struct S {
  S(){ std::println("S()"); }
  S(const S &){ std::println("S(const S &)"); }
  S(S &&){ std::println("S(S &&)"); }
  S &operator=(const S &){
    std::println("operator=(const S &)");
    return *this;
  }
  S &operator=(S &&){
    std::println("operator=(S &&)");
    return *this;
  }
  ~S() { std::println("~S()"); }
};
```

Exercise: Build your first C++ experiment.

Do you have a question about C++ that's been nagging you? Can you design an experiment to test it? Remember that Compiler Explorer now allows you to execute code.

Exercise: Start collecting your experiments.

Once you have created an experiment and test, be sure to save it. Consider using GitHub gists as a simple way to save and share your tests with others.

6.1: Resources

- A quick start example with Compiler Explorer.

7: Remember: C++ Is Not Object-Oriented

I'm not the first person to state this, and I won't be the last. I think this concept is now well accepted, but I still see learners of C++ focusing on "OOP."

Bjarne Stroustrup in The C++ Programming Language 3rd Edition states:

> C++ is a general-purpose programming language with a bias towards systems programming that
>
> - is a better C,
> - supports data abstraction,
> - supports object-oriented programming, and
> - supports generic programming.

You must understand that C++ is a multi-discipline programming language to make the most of the language. C++ supports effectively all of the programming paradigms that exist today.

- Procedural
- Functional
- Object-Oriented
- Generic
- Compile-Time (`constexpr` and template metaprogramming)

Knowing when it is appropriate to use each of these tools is the key to writing good C++. Projects that rigidly stick to one paradigm miss out on the best features of the language.

Don't try to use every technique possible all of the time. You will end up with a mess of difficult to maintain and read code. Appropriately using the appropriate techniques at the appropriate times takes discipline and practice.

Exercise: Question your current design.

If you could break out of the current design your project is using, what would you do differently?

7.1: Resources

- Functional Programming in C++
- C++ Weekly Ep 137: C++ Is Not an Object Oriented Language

8: Learn Another Language

Considering that C++ is not an object-oriented language, you have to know many different techniques to make the most of C++.

The following exercises will help expose you to other languages. But the fact is that currently, few languages are pure single paradigm languages.

Every language has its preferred way of doing things that work within the language's preferred paradigm.

Ben Deane recommends this set of languages that all programmers should learn:

- ALGOL family (C and descendants)
- Forth
- Lisp and dialects
- Haskell
- Smalltalk
- Erlang

Exercise: Pick a functional language to learn

Can you find a pure functional language?

Exercise: Pick an object-oriented language to learn

Finding a pure object-oriented language is even harder! Even Java has lambda functions these days.

Exercise: Pick a language with a different syntax

Languages that look like C-family languages will likely be more comfortable for you. Try to find a language that looks different and stretches your mind.

8.1: Resources

- Execution in the Kingdom of Nouns - gets you thinking about different programming paradigms

9: Know Your Standard Library

The C++ standard library is continuously growing. In the C++23 standard the library is approximately 1403 pages, or 66% of the standard. In C++17 it was only 965 pages.

It is important to be aware of the changes that have happened in the standard. We will only touch on portions of them in this book, but some highlights from C++20 and C++23 include:

- greatly expanded date/time library (with calendar)
- many changes to concurrency and threading support
- `std::format`
- ranges
- concepts
- `<stacktrace>` and other diagnostics helpers

Exercise: Browse through the list of C++23 standard library changes and see what catches your eye https://en.cppreference.com/w/cpp/compiler_support/23

10: Use The Tools

Throughout this book you will see references to tooling, and an entire section on just “Use(ing) The Tools”.

C++, by default, has many gotchas, foot-guns and painful corner cases. We use a wide variety of tools to mitigate these issues in C++.

This is a core principle in using C++ correctly: enable every possible tool that you can!

11: Don’t Invoke Undefined Behavior

Ok, there’s a lot that’s Undefined Behavior (UB), and it’s hard to keep track of, so we’ll give some examples in the following sections. The critical thing that you need to understand is that UB’s existence breaks your entire program.

[intro.abstract]

> A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input.
>
> However, if any such execution contains an undefined operation, this document places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

Note the sentence “this document places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation)”

If you have UB, the entire program is suspect.

The next several items discuss ways to reduce the risk of undefined behavior in your project.

Exercise: Using UBSan, ASan and Warnings

Understanding all of Undefined Behavior is likely impossible. Fortunately, we do have tools that help. Hopefully, you already have your code enabled for UBSan, ASan, and have your warnings enabled. Now is a great time to go back and evaluate what options you have and see if there is anything new you can discover.

11.1: Resources

- C++Now 2018: John Regehr “Closing Keynote: Undefined Behavior and Compiler Optimizations”
- CppCon 2018: Barbara Geller & Ansel Sermersheim “Undefined Behavior is Not an Error”

12: Never Test for `this` To Be `nullptr`, It's UB

Figure 2. Invalid check for `this` to be `nullptr`.

```
1  int Class::member() {
2    if (this == nullptr) {
3      // removed by the compiler, it would be UB
4      // if this were ever null
5      return 42;
6    } else {
7      return 0;
8    }
9  }
```

Technically it isn't the check that is Undefined Behavior (UB). But it's impossible for the check ever to fail. If the `this` were to be equal to `nullptr`, you would be in a state of Undefined Behavior. People used to do this all the time, but it's always been UB. You cannot access an object outside its lifetime. Compilers today will always remove this check.

The only way it's theoretically possible for `this` to be null is when you call a member directly on a null object.

Bad examples lie ahead, do not repeat them.

Figure 3. Bad call of member on `nullptr`.

```
Type *obj = nullptr;
obj->do_thing(); // never do this
```

Even in the (technically OK, but never do this) scenario of calling `delete this`.

Figure 4. Bad example of `delete this`.

```
struct S {
  std::string data;

  void delete_yourself() {
    // do things
    delete this; // technically OK

    if (this) {
      // this block will always be executed, nothing changed
      // our view of `this`
    }

    // never do this
    data.size(); // UB, data's lifetime has ended
  }
};
```

There is no scenario in which a check for `if (this)` will return false on a modern compiler.

Exercise: Do you check for `this` to be `nullptr`?

A check for `nullptr` can hide as a check for `NULL` or a check against `0`. A check for `this` to be `NULL` is likely to only exist in very old code bases. Make sure you have your warnings enabled, then look for these cases.

It's probably interesting in general to search for `this ==` in your codebase and see what weird things are there.

12.1: Resources

- Porting to GCC-6 Optimizations remove null pointer checks for `this`

13: Never Test for A Reference To Be `nullptr`, It's UB

Figure 5. Tests for `null` references are removed

```
1  int get_value(int &thing) {
2    if (&thing == nullptr) {
3      // removed by compiler
4      return 42;
5    } else {
6      return thing;
7    }
8  }
```

It's UB to make a null reference, don't try it. Always assume a reference refers to a valid object. Use this fact to your advantage when designing API's.

Exercise: Check for checking the address of an object

There are many valid use cases for `&thing ==` to check for a specific address of an object, but there are also many ways this check can be wrong.

Search through your code for statements that check an object's memory address and understand what they are doing and how (or if) they work.

What other ways might the address of an object be checked besides `==`?

This exercise gives you some great experience working with various searching / grepping tools and playing with regex.

13.1: Resources

- -Wtautological-undefined-compare

Part II: Use The Tools

14: Use the Tools: Automated Tests

You need a single command to run tests. If you don't have that, no one will run the tests.

- Catch2 - popular and well supported testing framework from Phil Nash and Martin Hořeňovský
- doctest - similar to catch2, but trimmed for compile-time performance
- Google Test
- Boost.Test - testing framework, boost style.

ctest is a test runner for CMake that can be used with any of the above frameworks. It is utilized via the `add_test` feature of CMake.

An ideal build / test scenario with CMake might look like this:

Figure 6. Possible easy build/test steps

```
cmake -S ../src/dir -B builddir -G <Generator>
cmake --build builddir --config <Configuration>
cd builddir
ctest -C <Configuration>
```

You need to be familiar with these tools, what they do, and pick from them.

Without automated tests the rest of this book is pointless. You cannot apply the practical exercises if you cannot verify that you did not break the existing code.

Oleg Rabaev on CppCast stated:

- If a component is hard to test, it is not properly designed.
- If a component is easy to test, it is a good indication that it is properly designed.
- If a component is properly designed, it is easy to test.

Exercise: Can you run a single command to run a suite of tests?

- Yes: Excellent! Run the tests and make sure they all pass!
- No: Does your program produce output?
  - Yes: Start with "Approval Tests," which will give you the foundation you need to get started with testing.
  - No: Develop a strategy for how to implement some minimal form of testing.

14.1: Resources

- CppCon 2018: Phil Nash "Modern C++ Testing with Catch2"
- CppCon 2019: Clare Macrae "Quickly Testing Legacy C++ Code with Approval Tests"
- C++ on Sea 2020: Clare Macrae "Quickly and Effectively Testing Legacy C++ Code with Approval Tests"

- CppCast Ep 263: Unit Testing with Oleg Rabaev

15: Use the Tools: Continuous Builds

Without automated tests, it is impossible to maintain project quality.

In the C++ projects I have worked on throughout my career, I've had to support some combination of:

- x86
- x64
- SPARC
- ARM
- MIPSEL

On

- Windows
- Solaris
- MacOS
- Linux

When you start to combine multiple compilers across multiple platforms and architectures, it becomes increasingly likely that a significant change on one platform will break one or more other platforms.

To solve this problem, enable continuous builds with continuous tests for your projects.

- Test all possible combinations of platforms that you support
- Test Debug and Release separately
- Test all configuration options
- Test against newer compilers than you support or require

Sanitizers can only instrument code not removed by the optimizer, but the optimizer exploits some times of UB, so you need *at least* Debug, Debug + Sanitizers, Release, Release + Sanitizers for *at least* one platform you support!

If you don't require 100% tests passing, you will never know the code's state.

Exercise: Enable continuous builds

Understand your organization's current continuous build environment. If one does not exist, what are the barriers to getting it set up? How hard would it be to get something like GitLab, GitHub actions, Appveyor, or Travis set up for your projects?

16: Use the Tools: Compiler Warnings

There are many warnings you are not using, most of them beneficial. `-Wall` is *not* all warnings on GCC and Clang. `-Wextra` is still barely scratching the surface!

`/Wall` on MSVC is *all* of the warnings. Our compiler writers do not recommend using `/Wall` on MSVC or `-Weverything` on Clang, because many of these are diagnostic warnings that are not actionable. GCC does not provide an equivalent.

Strongly consider `-Wpedantic` (GCC/Clang). This command line options disable language extensions and get you closer to the C++ standard. The more warnings you enable today, the easier time you will have with porting to another platform in the future.

As of C++20 mode in MSVC `/permissive-` is no longer necessary, it is now the default setting for cl.exe

Exercise: Enable More Warnings

1. Explore the set of warnings available with your compiler. Enable as many as you can.
2. Fix the new warnings generated.
3. Goto 1.

MSVC has an excellent set of warnings that can be enabled by warning level. You can start with `/W1` and work your way up to `/W4` as you fix each set of warnings.

This process will feel tedious and meaningless, but these warnings will catch real bugs.

Exercise: Discuss enabling `-Werror` or `-WX` on your CI to ensure warnings do not accumulate.

K> I've observed that you need "Warnings as Errors" enabled on both developer machines and CI, otherwise developers treat the CI as their enemy, and they disable more and more warnings!

16.1: Resources

- C++ Best Practices website curated list of warnings
- GCC's full warning list
- Clang's full warning list
- MSVC's Compiler warnings that are off by default

- C++ Weekly Ep 168 - Discovering Warnings You Should Be Using

17: Use the Tools: Static Analysis

Static analysis tools are tools that analyze your code without compiling or executing it. Your compiler is one such tool and your first line of defense.

Many such tools are free and some are free for open source projects.

cppcheck and clang-tidy are two popular and free tools with major IDE and editor integration.

Exercise: Enable More Static Analysis

Visual Studio: look into Microsoft's static analyzer that ships with it. Consider using Clang Power Tools. Download cppcheck's addon for visual studio

CMake: Enable cppcheck and clang-tidy integration

Exercise: Enable Static Analysis in Your IDE

Most modern IDEs have built in support for clang-tidy and other static analysis tools. Investigate how to enable that support and configure your `.clang-tidy` file to have the analysis enabled that is appropriate for your project.

17.1: Resources

- cppbestpractices.com list of static analyzers

18: Use The Tools: Consider Custom Static Analysis

Remember to focus on making your interface hard to use wrong. Then, as a second line of defense, consider writing your own static analysis.

These might take the form of:

- custom clang-tidy analysis
- using clang-query to query the AST of your project for common errors
- writing a custom rule for cppcheck
- CodeQL custom check

Exercise: Look for common coding errors.

Simply discuss common issues you see in your code base with other members of your team. See if you can isolate one or two common issues that a custom analysis check would be able to find. Implement a check with one of the above tools.

18.1: Resources

- Lightning Talk: Using Clang Query to Isolate AST Elements - Kristen Shaker - C++ on Sea 2022
- Database of custom cppcheck rules to study
- CodeQL Tutorials

19: Use the Tools: Sanitizers

The sanitizers are runtime analysis tools for C++ and are built into GCC, Clang, and MSVC.

If you are familiar with Valgrind, the sanitizers provide similar functionality but many orders of magnitude faster than Valgrind.

Available sanitizers are:

- Address (ASan)
- Undefined Behavior (UBSan) (More on Undefined Behavior later)
- Thread
- DataFlow (use for code analysis, not finding bugs)
- Lib Fuzzer (addressed in a later chapter)

Address sanitizer, UB Sanitizer, Thread sanitizer can find many issues almost like magic. Support is currently increasing in MSVC at the time of this book's writing, while GCC and Clang have more established support for the sanitizers.

John Regehr recommends always enabling ASan and UBSan during development.

When an error such as an out of bounds memory access occurs, the sanitizer will give you a report of what conditions led to the failure, often with suggestions for fixing the problem.

You can enable Address and Undefined Behavior sanitizers with a command similar to:

Figure 7

```
1  gcc -fsanitize=address,undefined <filetocompile>
```

Sanitizers must also be enabled during the linking phase of the project build.

Examples for how to use sanitizers with CMake exist in the C++ CMake Template Project

K> Remember to combine Debug, Release, Sanitizers-on, and Sanitizers-off builds, as each combination can expose different code issues.

Exercise: Enable Sanitizers

- Investigate how to add sanitizer support for your existing project
- Enable ASan first
- Run the full test suite and investigate any problems found
- Enable UBSan second
- Run full test suite again

End goal: get all tests running with ASan, and UBSan enabled on your continuous build environment.

Exercise: Fallback to Valgrind or Dr Memory

If you are unable to run ASan and UBSan on your environment, investigate running your test suite with either Dr Memory (Windows/Linux/MacOS) or Valgrind (Linux/MacOS)

19.1: Resources

- AddressSanitizer (ASan) for Windows with MSVC
- Sanitizers source and documentation on GitHub
- Clang AddressSanitizer documentation
- Clang UndefinedBehaviorSanitizer documentation

20: Use The Tools: Hardening

Safety is important in many industries, and with many products. If safety and security are your main concerns, you should consider shipping your binaries with hardening.

TBD FILL OUT

Exercise: Consider UBSan Minimal Runtime

The “minimal runtime” version of UBSan is designed for binary hardening.

21: Use the Tools: Multiple Compilers

Support *at least* 2 compilers on your platform. Each compiler does different analyses and implements the standard in a slightly different way.

If you use Visual Studio, you should be able to switch between clang and cl.exe relatively easily. You can also use WSL and enable remote Linux Builds.

If you use Linux, you should be able to switch between GCC and Clang easily.

On MacOS, be sure the compiler you are using is what you think it is. The `gcc` command is likely a symlink to clang installed by Apple.

apple-clang is not the same thing as mainline clang and its version numbers don't match up. It's often difficult to know which features apple-clang supports. The cppreference compiler-support reference might help.

For installing newer or different compilers on your platform, the following is available:

Ubuntu / Debian

- GCC - Toolchain PPA
- Clang - apt packages

Windows

- GCC MinGW
- Clang official downloads

MacOS

- Homebrew / MacPorts

Exercise: Add Another Compiler

Since you have already enabled continuous builds of your system, it's time to add another compiler.

A new version of the compiler you currently require is always a good idea. But if you only support GCC, consider adding Clang. Or if you only support Clang, add GCC. If you're on Windows, add MinGW or Clang in addition to MSVC.

Exercise: Add Another Operating System

Hopefully, at least some portion of your project can be ported to another operating system. The exercise of getting parts of the project compiling on another operating system and toolchain will teach you a lot about your code's nature.

21.1: Resources

- C++Now 2015: Jason Turner “Thinking Portable: How and Why to make your C++ Cross Platform”

22: Use The Tools: Fuzzing and Mutating

Your imagination limits the tests that you can create. Do you try to be malicious when calling your APIs? Do you intentionally pass malformed data to your inputs? Do you process inputs from unknown or untrusted sources?

Generating all possible inputs to all possible function calls in all possible combinations is impossible. Fortunately, tools exist to solve this problem.

22.1: Fuzzing

Fuzz testers generate strings of random data of various lengths. The test harness you write consumes these strings of data and processes them in some way that is appropriate for your application. The fuzz tester analyzes coverage data generated from your test's execution and uses that information to remove redundant tests and generate new novel and unique tests.

In theory, a fuzz test will eventually reach 100% code coverage of your tested code, if left to run long enough. Combined with AddressSanitizer, this makes a powerful tool for finding bugs in your code. One interesting article from 2015 describes how the combination of a fuzz tester and AddressSanitizer could have found the security flaw "heartbleed" in OpenSSL in less than 6 hrs.

This 6 hrs is now drastically out of date. With modern fuzz testing tools and newer computers, a vulnerability like heartbleed can be discovered in just a few minutes.

Fuzz testing primarily finds memory and security flaws.

Many different fuzzing tools exist. For the sake of this section, I am going to focus on LLVM's libFuzzer. Most fuzz testers operate under the same premise.

You must provide some sort of entry point. The entry point generally takes the form of a function like:

Figure 8. libFuzzer entry point.

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data,
                                      size_t Size);
```

The `Data` pointer is always valid, and the `Size` parameter is >= 0.

If your library primarily parses input files (think libpng) then your job is quite easy:

Figure 9. libFuzzer data being used.

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data,
                                      size_t Size)
{
  parseInput(Data, Size);
}
```

If your functions take data structures instead of input strings, your job is slightly more complicated but doable.

Figure 10. Advanced libFuzzer data usage.

```
template<typename Type>
std::tuple<const uint8_t *, size_t, Type>
    createStruct(const uint8_t *Data, size_t Size)
{
  // we're only allowed to do this with trivial types
  static_assert(std::is_trivial_v<Type>);
  Type result{}; // default initialize
  const auto bytesToRead = std::min(sizeof(Type), Size);
  std::memcpy(&result, Data, bytesToRead);
  return {std::next(Data, bytesToRead), Size - bytesToRead, result};
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data,
                                      size_t Size)
{
  // This example is meant as inspiration, it has not been
  // tested in a real test
  auto [newDataPtr, remainingSize, Obj1]
    = createStruct<Type1>(Data,Size);
  auto [lastDataPtr, lastSize, Obj2]
    = createStruct<Type2>(newDataPtr, remainingSize);

  functionToTest(Obj1, Obj2);
}
```

The fuzzer will quickly learn that any new data input where `Size > sizeof(Type1) + sizeof(Type1)` does not generate new code paths and will focus on the appropriate amount of data.

Look into the newer (2022) fuzzing library from Google FuzzTest designed to simplify the process of hooking up fuzz tests into your project.

22.2: Mutating

Mutation testing works by modifying conditionals and constants in the code being tested.

Figure 11. Pseudo code example.

```
bool greaterThanFive(const int value) {
    return value > 5; // comparison
}

void tests() {
  assert(greaterThanFive(6));
  assert(!greaterThanFive(4));
}
```

A mutation tester could modify the constant `5` or the `>` so the resulting code might become

Figure 12. Mutated code.

```
1 bool greaterThanFive(const int value) {
2     return value < 5; // mutated
3 }
```

Any test that continues to pass is a “mutant that has survived” and may indicate either a flawed test or a bug in the code.

Exercise: Create a fuzz test harness.

Apply the examples demonstrated here to create fuzz testers for your code. What challenges do you hit?

Look at FuzzedDataProvider.h for more helper functions

Exercise: Investigate mutation testing.

The author of this book has no direct experience with mutation testing. Is it something you can use in your project? What interesting resources do you find?

Exercise: Try FuzzTest if you can.

Only very recent versions of clang are supported by FuzzTest, look into the currently supported compilers and see if you can try it with your code.

22.3: Resources

- C++Now 2018: Marshall Clow “Making Your Library More Reliable with Fuzzing”
- C++ Weekly Ep 85: Fuzz Testing
- CppCast: Alex Denisov “Mutation Testing With Mull”
- NDC TechTown 2019: Seph De Busser “Testing The Tests: Mutation Testing for C++”
- CppCon 2017: Kostya Serebryany “Fuzz or lose…”
- CppCon 2020: Barnabás Bágyi “Fuzzing Class Interfaces for Generating and Running Tests with libFuzzer” - Inspirational talk about using fuzzing in novel ways. Video is not yet on YouTube, but look for it after this book is published.
- Autotest - Library associated with “Fuzzing Class Interfaces for Generating and Running Tests with libFuzzer” talk.
- FuzzTest - Fuzz testing library from google.
- DeepState - Fuzz testing tools from TrailOfBits

- oss-fuzz - Continuous Fuzzing for Open Source Projects
- Mutate++ - Mutation testing tool

23: Use the Tools: Build Generators

- CMake
- Meson
- Bazel
- Others

Raw make files or Visual Studio project files make each of the things listed above too tricky to implement. Use a build tool to help you with maintaining portability across platforms and compilers.

Treat your build scripts like any other code. They have their own set of best practices, and it's just as easy to write unmaintainable build scripts as it is to write unmaintainable C++.

Build generators also help abstract and simplify your continuous build environment with tools like `cmake --build`, which does the correct thing regardless of the platform in use.

Exercise: Investigate your build system.

- Does your project currently use a build generator?
- How old are your build scripts?

See if there are current best practices you need to apply. Are there tidy-like or formatting tools you can run on your build scripts?

Read back over the previous best practices from this book and see how they apply to your build scripts.

- Are you repeating yourself?
- Are there higher-level abstractions available?

Recent versions of CMake have added tools like `--profiling-output` to help you see where the generator is spending its time.

23.1: Resources

- Professional CMake: A Practical Guide
- cmake-tidy
- C++Now 2017: Daniel Pfeiffer “Effective CMake”
- CppCon 2017: Mathieu Ropert “Using Modern CMake Patterns to Enforce a Good Modular Design”
- CppCon 2018: Jussi Pakkanen “Compiling Multi-Million Line C++ Code Bases Effortlessly with the Meson Build System”
- BazelCon 2019
- CppCon 2019: Mathieu Ropert “Cato the Elder” - Short rant about build script quality
- C++ Weekly Ep 218 - The Ultimate CMake / C++ Quick Start
- Twitter Discussion on CMake Resources

24: Use the Tools: Package Managers

Recent years have seen an explosion of interest in package managers for C++. These two have become the most popular:

- Vcpkg
- Conan

There is a definite advantage to using a package manager. Package managers help with portability and reducing maintenance load on developers.

I also want to draw your attention to CPM.

CPM provides a simple and straightforward direct integration into CMake for building and linking to "`fetch_content`-able" CMake projects.

Exercise: What are your dependencies?

Take time to inventory your project's dependencies. Compare your dependencies with what is available with the package managers above. Does any one package manager have all of your dependencies? How out of date are your current packages? What security fixes are you currently missing?

Part III: API and Code Design Guidelines

25: Make your interfaces hard to use wrong.

Your interface is your first line of defense. If you provide an interface that is easy to use wrong, your users *will* use it wrong.

If you provide an interface that's hard to use wrong, your users have to work harder to use it wrong. But this is still C++; they will always find a way.

Interfaces hard to use wrong will sometimes result in more verbose code where we would maybe like more terse code. You have to choose what is most important. Correct code or short code?

This is a high-level concept; specific ideas will follow.

25.1: Resources

- The Little Manual of API Design

26: Consider If Using the API Wrong Invokes Undefined Behavior

Do you accept a raw pointer? Is it an optional parameter? What happens if `nullptr` is passed to your function?

What happens if a value out of the expected range is passed to your function?

Some developers make the distinction between “internal” and “external” APIs. They allow unsafe APIs for internal use only.

Is there any guarantee that an external user will never invoke the “internal” API?

Is there any guarantee that your internal users will never misuse the API?

Exercise: Investigate Checked Types

The C++ Guideline Support Library (GSL) has a `not_null` pointer type that guarantees, because of zero cost abstractions, that the pointer passed is never `nullptr`. Would that work for your APIs that currently pass raw pointers (assuming that rearchitecting the API is not an option)?

`std::string_view` (C++17) and `std::span` (C++20) are great alternatives to pointer / length pairs passed to functions.

26.1: Resources

- boost::safe_numerics

27: Be Afraid of Global State

Reasoning about global state is hard.

Any non-`const static` value, or `std::shared_ptr<>` could potentially be global state. It is never known who might update the value or if it is thread-safe to do so.

Global state can result in subtle and difficult to trace bugs where one function changes global state, and another function either relies on that change or is adversely affected by it.

Exercise: Global State, What's Left?

If you've done the other exercises, you've already made all of your `static` variables `const`. This is great! You've possibly even made some of them `constexpr`, which is even better!

But you probably have global state still lurking. Do you have a global singleton logger? Could the logger be accidentally sharing state between the modules of your system?

What about other singletons? Can they be eliminated? Do they have threading initialization issues (what happens if two threads try to access one of the objects for the first time at the same time)?

27.1: Resources

- Retiring the Singleton Pattern - Peter Muldoon - Meeting C++ 2019

28: Use Stronger Types

Consider the API for POSIX `socket`:

Figure 13. POSIX socket API.

```
socket(int, int, int);
```

The parameters (in some order) represent:

- type
- protocol
- domain

This design is problematic, but there are less obvious ones lurking in our code.

Figure 14. Poorly defined constructor.

```
Rectangle(int, int, int, int);
```

This function could be `(x, y, width, height)`, or `(x1, y1, x2, y2)`. Less likely, but still possible, is `(width, height, x, y)`.

What do you think about an API that looks like this?

Figure 15. Strongly typed constructor.

```
Rectangle(Position, Size);
```

In many cases, it only takes a little effort to make more strongly typed APIs.

Figure 16. Stronger typed definitions.

```
struct Position {
  int x;
  int y;
};

struct Size {
  int width;
  int height;
};

struct Rectangle {
  Position position;
  Size size;
};
```

Which can then lead to other, logically composable statements with operator overloads such as:

Figure 17. Coupled type operator overload.

```
// Return a new rectangle that has been
// moved by the offset amount passed in
Rectangle operator+(Rectangle, Position);
```

It's possible that making structs can *increase* performance in some cases C++ Weekly Ep 119, Negative Cost Structs.

28.1: Avoid Boolean Arguments

This chapter's pre-release reader pointed out that Steve Maguire says, "Make code intelligible at the point of call. Avoid Boolean arguments," in Chapter 5 of his book Writing Solid Code.

In C++11, `enum class` gives you an easy way to add stronger typing, avoid boolean parameters, and make your API harder to use wrong.

Consider:

Figure 18. Non-obvious order of parameters.

```
struct Widget {
  // this constructor is easy to use wrong, we
  // can easily transpose the parameters
  Widget(bool visible, bool resizable);
}
```

Compared to:

Figure 19. Stronger typing with scoped enumerations.

```
struct Widget {
  enum struct Visible { True, False };
  enum struct Resizable { True, False };

  // still possible to use this wrong, but MUCH harder
  Widget(Visible visible, Resizable resizable);
}
```

Identify the problematic APIs in your existing code.

What function call do you regularly get out of order? How can it be fixed?

Exercise: Research strong typedef libraries for C++.

There are existing libraries that simplify some of the boilerplate code for you when making a strongly typed `int`. Jonathan Muller, Bjorn Fahller, and Peter Sommerlad have each written one, and others are available.

Exercise: Consider `=deleteing` problematic conversions.

Figure 20. Simple function declaration.

```
double high_precision_thing(double);
```

What if calling the above with a `float` is likely to be a bug?

Figure 21. Deleting a problematic accidental promotion from `float` to `double`.

```
double high_precision_thing(double);
double high_precision_thing(float) = delete;
```

Any function or overload can be `=deleted` in C++11.

Enable clang-tidy's Easily Swappable Parameters check

28.2: Resources

- C++ Weekly Ep 107: “The Power of `=delete`”
- Adi Shavit and Björn Fahller “The Curiously Recurring Pattern of Coupled Types”
- Research “Affine space types.”
- C++Now 2017: Jonathan Müller “Type-safe Programming”

29: Use `[[nodiscard]]` Liberally

`[[nodiscard]]` is a C++ attribute that tells the compiler to warn if a return value is ignored. It can be used on functions:

Figure 22. `[[nodiscard]]` example usage.

```
[[nodiscard]] int get_value();

int main()
{
  // warning, [[nodiscard]] value ignored
  get_value();
}
```

And on types:

Figure 23. `[[nodiscard]]` on types.

```
struct [[nodiscard]] ErrorCode{};

ErrorCode get_value();

int main()
{
  // warning, [[nodiscard]] value ignored
  get_value();
}
```

C++20 adds the ability to provide a description.

Figure 24. C++20's `[[nodiscard]]` with description.

```
[[nodiscard("Ignoring this result leaks resources")]]
```

Our `divide` example is a straightforward application of `[[nodiscard]]`.

Figure 25. `[[nodiscard]]` applied to `divide` function.

```
import std;

[[nodiscard]] constexpr auto divide(std::integral auto numerator,
                                    std::integral auto denominator) {
  // is integer division
  if (denominator == 0) {
    throw std::runtime_error("divide by 0!");
  }
  return numerator / denominator;
}

[[nodiscard]] constexpr auto divide(auto numerator, auto denominator) {
  // is floating point division
  return numerator / denominator;
}
```

And constructor:

Figure 26. Example of [[nodiscard]] constructor

```
struct Holder
{
  // warn if the result of this constructor is unused
  [[nodiscard]] Holder() = default;

  // bad practice, but exists so that GCC does, in fact,
  // generate a warning for this code below.
  int *p = new int();
};

int main()
{
  // should generate a warning
  Holder();
}
```

Exercise: Determine a set of rules for using `[[nodiscard]]`

Read the Reddit discussion "An Argument Pro Liberal Use Of nodiscard". Consider your types and functions. Which values should be `[[nodiscard]]`?

Should it be a compiler error or warning to call these functions and ignore the result?

- `vector.size()`
- `vector.empty()`
- `vector.insert()`

29.1: Resources

- "An Argument Pro Liberal Use Of nodiscard"
- C++ Weekly Ep 30: C++17's `[[nodiscard]]` Attribute
- C++ Weekly Ep 199: C++20's `[[nodiscard]]` Constructors And Their Uses

30: Forget Header Files Exist

C++23 has full support for C++ modules, and the standard library is now mandated to provide modules.

Figure 27. Before C++23 Modules

```
#include <string>
#include <vector>
#include <map>

int main()
{
  std::map<int, std::vector<std::string>> data;
  // do stuff
}
```

Figure 28. After C++23 Modules

```
import std;

int main()
{
  std::map<int, std::vector<std::string>> data;
  // do stuff
}
```

Evidence from Microsoft, who, at the time of the writing of this section, has the most complete modules implementation, shows that this is actually considerably faster than including header files.

Importing a module is basically “free”

No code has to be immediately parsed when a module is included, so there’s effectively no cost, which is why it’s OK that the entire standard library is now included under a single `import` directive

This is a very basic (but complete) example of defining a C++20 module. See the resources for more information about actually using modules.

Figure 29. Very basic .ixx module interface file

```
// my_module.ixx
export module my_module;

import std;

// I'm only exporting the float overload
export constexpr [[nodiscard]] float calc(float val) noexcept
{
  return val * 10.1f;
}

export void greet(std::string_view name);
```

Figure 30. Very basic .cpp module implementation file

```
// my_module.cpp
module my_module;

import std;

void greet(std::string_view name)
{
  std::println("Hello {}", name);
}
```

Figure 31. Very basic module usage

```
import my_module;

int main()
```

```
4 {
5   greet("Jason");
6 }
```

30.1: Resources

- Modules the beginner's guide - Daniela Engert - Meeting C++ 2019
- Contemporary C++ in Action - Daniela Engert - CppCon 2022
- So, You Want to Use C++ Modules … Cross-Platform? - Daniela Engert - C++ on Sea 2023

31: Export Module Overloads Consistently

C++23 modules should be used (remember to forget header files exist), but they do introduce a new challenge to making sure your library can be correctly used.

If you are familiar with using DLL's and explicit library exports (generally required with MSVC, but also necessary if you choose to not export all symbols from a dynamic library on any operating system) then you may have seen code like this:

Figure 32. Simple library export example

```
1  EXTERN_DLL_EXPORT int calculate(int input) {
2    return input * 10;
3  }
```

If you are inconsistent with your exports you will get a link error:

Figure 33. Simple library incorrect example

```
// library.hpp
[[nodiscard]] constexpr int calculate(int input) {
  return input * 10;
}

EXTERN_DLL_EXPORT [[nodiscard]] constexpr int calculate(float input) {
  return input * 10.1;
}
```

Figure 34. Simple library incorrect usage example

```
#include "library.hpp"
import std;

int main()
{
  // this will compile but give link-time errors
  // (either at static linking time or runtime linking)
  // because the float overload was found at compile-time
  // but not exported for library usage
  std::println("{}", calculate(3));
}
```

C++20 modules present us with a similar, but different, and potentially more insidious problem.

Figure 35. Partially exported overload set

```
// library.ixx
export module library;

// I'm only exporting the float overload
export [[nodiscard]] constexpr float calculate(float val) noexcept
{
  return val * 10.1f;
}

[[nodiscard]] constexpr int calculate(int val) noexcept
{
  return val * 10;
}
```

Figure 36. Partially exported overload usage

```
import library;
import std;

int main()
{
  // this will compile and link and issue no warnings, but will
  // print float "30.3" instead of the probably expected int "30"
  std::println("{}", calculate(3));
}
```

32: Prefer Stack Over Heap

Stack objects (locally scoped objects that are not dynamically allocated) are much more optimizer friendly, cache-friendly, and may be entirely eliminated by the optimizer. As Björn Fahller has said, “assume any pointer indirection is a cache miss.”

In the most simple terms:

Figure 37. OK idea, uses stack and can be optimized.

```
1  std::string make_string() { return "Hello World"; }
```

Figure 38. Bad idea, uses the heap.

```
1  std::unique_ptr<std::string> make_string() {
2    return std::make_unique<std::string>("Hello World");
3  }
```

Figure 39. OK idea.

```
1  void use_string() {
2    // This string lives on the stack
3    std::string value("Hello World");
4  }
```

Figure 40. Really bad idea, uses the heap and leaks memory.

```
1  void use_string() {
2    // The string lives on the heap
3    std::string *value = new std::string("Hello World");
4  }
```

Remember, `std::string` itself might allocate internally, and use the heap. If no heap usage at all is your goal, you will need to take other measures. The goal is no *unnecessary* heap allocations.

Generally speaking, objects created with `new` expressions (or via `make_unique` or `make_shared`) are heap objects, and have *Dynamic Storage Duration*. Objects created in a local scope are stack objects and have *Automatic Storage Duration*.

N> It’s much easier for the compiler and tools to find reads of uninitialized stack values than heap values.

Exercise: Look for heap usage

Sometimes developers with C and Java backgrounds have a hard time with this. For Java, it’s because `new` is required to create objects. For C, it is because the C compiler cannot perform the same kinds of optimizations that the C++ compiler can because of differences in the language.

So some of this unnecessary heap usage may have ended up in your current code.

Exercise: Run a heap profiler

There are several heap profiling tools, and there may even be one built into your IDE. Examine your heap usage and look for potential abuses of the heap in your

project. It's possible that most of your heap allocations come from accidental copies of containers such as `std::string` or `std::vector`.

32.1: Resources

- Code::Dive 2018: Björn Fahller "What Do You Mean By Cache Friendly?"
- heaptrack - a heap memory profiler for Linux
- Massif: a heap profiler

33: Don’t return raw pointers

Returning a raw pointer makes the reader of the code and user of the library think too hard about ownership semantics. Prefer a reference, smart pointer, non owning pointer wrapper, or consider an optional reference.

Figure 41. Function returning a raw pointer.

```
1  int *get_value();
```

Who owns this return value? Do I? Is it my job to `delete` it when I’m done with it?

Or even worse, what if the memory was allocated by `malloc` and I need to call `free` instead?

Is it a single `int` or an array of `int?`

This code has far too many questions, and not even `[[nodiscard]]` can help us.

Exercise: Find the potential leaks in your code

By now, you’ve done enough of these API related exercises to know what to do. Go and look for these in your code! See if there’s a better way! Can you return a value, reference, or `std::unique_ptr` instead?

34: Know Your Containers

Prefer your containers in this order:

- `std::array<>`
- `std::vector<>`

`std::array<>`

> A fixed-size stack-based contiguous container. The data size must be known at compile-time, and you must have enough stack space to hold the data. This container helps us prefer stack over heap. Known location and contiguousness results in `std::array<>` becoming a “negative cost abstraction.” The compiler can perform an extra set of optimizations because it knows the data’s size and location.

`std::vector<>`

> A dynamically-sized heap-based contiguous container. While the compiler does not know where the data will ultimately reside, it does know that the elements are laid out adjacent to each other in RAM. Contiguousness gives the compiler more optimization opportunities and is more cache-friendly.

Almost anything else needs a comment and justification for why. A flat map with linear search is likely better than an `std::map` for small containers.

But don’t be too enthusiastic about this. If you need key lookup, use `std::map` and evaluate if it has the performance and characteristics you want.

Exercise: Replace `vector` With `array`

Look for fixed-size `vectors` and replace them with `array` where possible. With C++17's Class Template Argument Deduction, this can be easier.

Figure 42. `const std::vector` with fixed-size data.

```
1  const std::vector<int> data{n+1, n+2, n+3, n+4};
```

can become

Figure 43. `const std::array` for fixed-size data.

```
1  const std::array<int, 4> data{n+1, n+2, n+3, n+4}; // C++11
2  const std::array data{n+1, n+2, n+3, n+4};          // C++17
```

You already made these `const`, now go back to `constexpr` them if you can.

34.1: Resources

- Bjarne Stroustrup "Are lists evil?"

35: Be Aware of Custom Allocation And PMR

C++17 added Polymorphic Memory Resources (PMR) which makes it trivially easy to add your own custom allocation strategies to the standard containers.

(Unfortunately, only GCC and MSVC implement this C++17 feature at the time of publication of this book.)

I do *not* recommend using PMR or other custom allocation strategies everywhere in the code. They should be mostly unnecessary if you follow the rest of the rules in this book.

However, I do consider it to be a best practice to know that these strategies exist and they can be used as tools to limit your use of the heap and get the “last mile” performance you might need.

A simple example is:

Figure 44. simple PMR example

```
import std;

int main() {
  // create stack space for data
  std::array<std::byte, 2048> stackBuf;

  // create monotonic_buffer_resource
  //   * no data is freed until the owning buffer is destroyed
  std::pmr::monotonic_buffer_resource
    rsrc(stackBuf.data(), stackBuf.size());

  // all list nodes are created in the stackBuf storage
  // without using any dynamic allocation
  std::pmr::list<int> listOfThings{
    {1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10},
    &rsrc};
}
```

The C++ standard provides several different allocation strategies that can be layered with fall-backs, and it's easy to create your own. Check out the resources for a comprehensive C++ Weekly playlist on this topic.

Exercise: Where Does PMR Fit In Your Project?

- Run a heap profiler and look for extraneous heap allocations. Eliminate those that you can.
- Experiment with PMR in the remaining hotspots to see where it might be able to help.

Exercise: Understand "Winking Out" Of Data

- This is an advanced topic and not for the faint of heart
- It is possible to actually create the contained object itself inside of the pmr resource
- When you do this it's possible to safely avoid calling all destructors if all data is owned by the one buffer resource
- Spend some time appreciating the concept

35.1: Resources

- C++ Weekly PMR Playlist
- Compiler Support Matrix On cppreference

36: Constrain Your Template Parameters With Concepts

Concepts will result in better error messages (eventually) and better compile times than SFINAE. Besides much more readable code than SFINAE.

If we continue to build on our `divide` example, we can take this `if constexpr` version from the Prefer `if constexpr` over SFINAE chapter.

Figure 45. `if constexpr` version of `divide` function.

```
import std;

template <typename Numerator, typename Denominator>
[[nodiscard]] constexpr auto divide(Numerator numerator, Denominator de\
nominator) {
  if constexpr (std::is_integral_v<Numerator> &&
                std::is_integral_v<Denominator>) {
    // is integral division
    if (denominator == 0) {
        throw std::runtime_error("divide by 0!");
    }
  }

  return numerator / denominator;
}
```

And we can split it back out as two different functions using concepts.

Concepts can be used in several different contexts. This version uses a simple `requires` clause after the function declaration.

Figure 46. Concepts in `requires` clause.

```
import std;

// overload resolution will pick the most specific version
template <typename Numerator, typename Denominator>
[[nodiscard]] constexpr auto divide(Numerator numerator, Denominator de\
nominator) requires
       (std::is_integral_v<Numerator>
        && std::is_integral_v<Denominator>) {
  // is integral division
  if (denominator == 0) {
    throw std::runtime_error("divide by 0!");
  }
  return numerator / denominator;
}

template <typename Numerator, typename Denominator>
[[nodiscard]] constexpr auto divide(Numerator numerator, Denominator de\
nominator) {
  return numerator / denominator;
}
```

This version uses concepts as function parameters. C++20 even has an "`auto` concept," which is an implicit template function.

Figure 47. Terse concepts requirement syntax.

```
import std;

[[nodiscard]] constexpr auto divide(std::integral auto numerator,
            std::integral auto denominator) {
  // is integer division
  if (denominator == 0) {
    throw std::runtime_error("divide by 0!");
  }
  return numerator / denominator;
}

[[nodiscard]] constexpr auto divide(auto numerator, auto denominator) {
  // is floating point division
  return numerator / denominator;
}
```

Concepts can define complex requirements, including expected members. This section only barely touches on the possibilities.

Exercise: Understand what concepts are provided with C++20.

As usual, cppreference helps by providing a list of concepts.

Exercise: Create your own concept.

Does this example give you some idea for an example of a concept that you would want, but isn't provided by `<concepts>`?

Look at the implementation of the very simple `std::integral` concept on cppreference and see if it inspires you.

36.1: Resources

- C++ Weekly Ep 194: From SFINAE To Concepts With C++20
- C++ Weekly Ep 196: What is `requires requires`

37: Understand `consteval` and `constinit`

I've already encouraged you to use `constexpr`.

C++20 added `consteval` and `constinit`, and it's important to understand what they do:

Variable Declaration:

- `consteval int x = /**/` - not valid, `consteval` applies only to functions
- `constexpr int x = /**/` - declares a `const` value that is usable at compile time, and may be evaluated at compile time, if required
- `static constexpr int x = /**/` - declares a `const` value that is usable at compile time, and is evaluated at compile time
- `constinit int x = /**/` - not valid
- `static constinit int x = /**/` - declares a non-`const` value (ie, mutable) that is not usable at compile time, but *is* evaluated at compile time

Function Declaration:

- `consteval int func(int)` - declares a function that *must* be evaluated at compile-time
- `constexpr int func(int)` - declares a function that *may* be called at compile-time
- `constinit int func(int)` - not valid, `constinit` applies only to variables

Exercise: Discuss if there is ever a time when a user-defined-literal should *not* be `consteval`?

User Defined Literals provide a shortcut for converting a literal into another type.

The standard library provides many different user defined literals. Two notable, but distinct ones are for `std::string` and `std::string_view`.

Figure 48. consteval udl

```
import std;

int main(const int argc, const char *[]) {
  using std::literals::string_literals;
  using std::literals::string_view_literals;

  // Currently the standard library provided `s` literal
  // is not `consteval`. Should it be?
  const auto my_string
    = "Hello World"s; // creates a string

  // Currently the standard library provided `sv` literal
  // is not `consteval`. Should it be?
  const auto my_string_view
    = "Hello World"sv; // creates a string_view
}
```

Exercise: What does it mean if a function *must* be called at compile time?

Figure 49. consteval exercise

```
consteval int get_value(const int input) {
  return input * 42;
}

int main(const int argc, const char *[]) {
  // will this code compile?
  const auto value = get_value(argc);
}
```

37.1: Resources

- Andreas Fertig's Blog: "A Neat Trick with consteval"
- C++ Weekly Ep 304: "C++23's `if consteval` vs C++20's `is_constant_evaluated` vs C++17's `if constexpr`"
- C++ Weekly Ep 308: "`if consteval` - There's More To This Story"

38: Prefer Spaceships

C++20 can generate any comparison operator for you (colloquially knows as the “spaceship operator” because of its shape).

- If you define `==`, the compiler will automatically generate `!=`
- If you define `<=>`, the compiler will generate all other comparisons (except for `==` and `!=`)
- You can explicitly default any comparison operation

Figure 50. basic spaceship example

```
struct MyData
{
  int i;
  int j;

  // provide all comparisons
  friend auto operator<=>(const MyData &, const MyData &) = default;
};
```

The C++ standard has started deprecating explicit comparison operations, look to it for examples.

Note that `std::string`, for the sake of performance, has a custom implemented `operator==` (provides `==` and `!=`) and `operator<=>` (provides the rest).

https://en.cppreference.com/w/cpp/string/basic_string/operator_cmp

If you provide a custom `operator<=>` for your type, the compiler will not provide a `operator==` or `operator!=` for you! This is similar to the “Rule of 0” and “Rule of 5” for special member functions.

- If you provide a custom `operator<=>` you must provide your own `operator==`
- It is unlikely that an explicitly defaulted `operator==` will do the correct thing if you need a custom `operator<=>`

Exercise: Implement a spaceship operator

Figure 51. spaceship operator exercise

```
import std;

struct Container
{
  // fixed-capacity container.
  std::array<int, 10> data;

  // size is the number of currently used elements
  std::size_t size;

  // what does the comparison operator need to look like?
  // will a defaulted one work?
  // do we need a custom operator==?
};
```

38.1: Resources

- [cppreference.com documentation]

39: Follow the Rule of 0

No destructor is always better when it's the correct thing to do. Empty destructors can destroy performance:

- They make the type no longer trivial
- Have no functional use
- Can affect inlining of destruction
- Implicitly disable move operations

If you need a destructor because you are doing resource management or defining a base class with virtual functions, you need to follow the Rule of 5.

`std::unique_ptr` can help you apply the Rule of 0 if you provide a custom deleter.

Exercise: Find Rule of 0 Violations in Your Code

Look for code like this (I guarantee you will find it).

Figure 52. Empty meaningless destructor.

```
1 struct S {
2   // a bunch of other things
3   ~S() {}
4 };
```

or worse:

Figure 53. Forward declared empty meaningless destructor.

```
1 // file.hpp
2 struct S {
3   ~S();
4 }
5
6 // file.cpp
7 S::~S() {}
```

Any mention of the special member functions implicitly disables the compiler-generated move operations. This includes `~S() = default;`.

Are these destructors necessary? Remove them if they are not.

If these destructors exist in types used in many places, you will likely be able to measure smaller binary sizes and better performance by taking this simple action.

> Some uses of the pImpl idiom require you to define a destructor. In this case, be sure to follow the Rule of 5.

Exercise: Use compiler-explorer to see one of the costs of breaking the Rule of 0.

Figure 54. Rule of 0 surprise impact

```
// experiment with this C++20 example in
// compiler-explorer.com
#include <vector>
#include <string>

struct S
{
  std::string data;

  // uncomment this line and observe the size of the compiled
  // binary in both -O3 and -O0 builds.
  //~S() = default;
};

void some_func(std::vector<S> &data){
    data.emplace_back();
}
```

39.1: Resources

- C++ Reference: The rule of three/five/zero
- C++ Weekly Ep 154: “One Simple Trick for Reducing Code Bloat”
- CppCon 2019: Jason Turner “Great C++ is_trivial”

40: If You Must Do Manual Resource Management, Follow the Rule of 5

If you provide a destructor because `std::unique_ptr` doesn't make sense for your use case, you *must* `=delete`, `=default`, or implement the other special member functions.

This rule was initially known as the Rule of 3 and is known as the Rule of 5 after C++11.

Figure 55. The special member functions.

```
struct S {
  S(); // default constructor
       // does not affect other special member functions

  // If you define any of the following, you must deal with
  // all the others.
  S(const S &);            // copy constructor
  S(S&&);                  // move constructor
  S &operator=(const S &); // copy assignment operator
  S &operator=(S &&);      // move assignment operator
};
```

`=delete` is a safe way of dealing with the special member functions if you don't know what to do with them!

You should also follow the Rule of 5 when declaring base classes with virtual functions.

Figure 56. Rule of 5 with polymorphic types.

```
struct Base {
  virtual void do_stuff();

  // because of the virtual function we know this class
  // is intended for polymorphic use, therefore our
  // tools will tell us to define a virtual destructor
  virtual ~Base() = default;

  // and now we need to declare the other special members
  // a good safe bet is to delete them, because properly and safely
  // copying or assigning an object via a reference or pointer
  // to a base class is hard / impossible

  S(const S &) = delete;
  S(S &&) = delete;
  S &operator=(const S &) = delete;
  S &operator=(S &&) = delete;
};

struct Derived : Base {
  // We don't need to define any of the special members
  // here, they are all inherited from `Base`.
}
```

Instead of `= delete` you can consider making these special members `protected`.

Exercise: Implement your own `unique_ptr<>` template

It's hard to get it 100% right. Write tests. Understand why the defaulted special member functions don't work.

Bonus points: implement it with C++20's `constexpr` dynamic allocation support.

Exercise: Look for Rule of 5 violations in your code

You are likely not providing consistent lifetime semantics in your existing code when you are defining the special member functions. To assess the impact, you can quickly `= delete;` any missing special member functions and see what breaks.

40.1: Resources

- C++ Reference: The rule of three/five/zero

Part IV: Code Implementation Guidelines

41: Don’t Copy and Paste Code

If you find yourself going to select a block of code and copy it: stop!

Take a step back and look at the code again.

- Why are you copying it?
- How similar will the source be to the destination?
- Does it make sense to make a function?
- Remember, Don’t Be Afraid of Templates

I have found that this simple rule has had the most direct influence on my code quality.

If the result of the paste operation was going in the current function, consider using a lambda.

C++14 style lambdas, with generic (aka `auto)` parameters, give you a simple and easy to use method of creating reusable code that can be shared with different data types while not having to deal with `template` syntax.

Exercise: Try CPD.

There are a few different copy-paste-detectors that look for duplicated code in your codebase.

For this exercise, download the PMD CPD tool and run it on your codebase.

If you use Arch Linux, this tool can be installed with AUR. The package is pmd; the tool is pmd-cpd.

Can you identify critical parts of your code that have been copied and pasted? What happens if you find a bug in one version? Will you be sure to see all of the locations that also need to be updated?

41.1: Resources

- Copy-Paste Programming
- The Last Line Effect
- i will not copy-paste code

42: Prefer `format` Over `iostream` Or c-formatting Functions

C++20 added the `<format>` header, which provides the function `std::format`. `std::format` takes a format string, parameters, and returns an `std::string` object. `format` is

- faster to compile than iostreams
- faster to execute than iostreams
- more readable than iostreams
- more type safe than `printf` family of functions

Figure 57. Simple format usage

```
1  import std;
2
3  int main()
4  {
5    const auto result = std::format("Hello {}!", "Jason");
6  }
```

Unfortunately, C++20's use cases were a little limited, with formatting to strings being the main mechanism, we tend to end up with code that looks like this:

Figure 58. Format cout usage

```
1  // using include because this is a C++20 example
2  #include <format>
3
4  int main()
5  {
6    std::cout << std::format("Hello {}!\n", "Jason");
7  }
```

To solve this problem, C++23 added the `<print>` header, which we can use via modules.

Figure 59. Using std::println

```
1  import std;
2
3  int main()
4  {
5    std::println("Hello {}!", "Jason");
6  }
```

There are several overloads for `std::print` and the helper `std::println` which automatically adds a newline to the end of the message.

This example is overly complicated example!

Figure 60. Using std::print's overloads

```
1  import std;
2
3  int main()
4  {
5    std::print(std::cout, "Hello");
6    std::println(stdout, " World");
7  }
```

Exercise: Learn the syntax for `std::print` and begin converting std::cout to `print` code.

Exercise: Use clang-tidy's modernize-use-std-print to upgrade any `printf` family functions you have.

Exercise: See if your AI coding assistant can convert `std::iostream` usage into `std::format` and `std::print` commands (ChatGPT is known to be reasonably good at this).

43: `constexpr` All The Things!

I'm using a much stronger argument for `constexpr` than I ever have before!

C++23 and modules together eliminate almost every remaining reason to not be using `constexpr`.

`constexpr` functions declared in your module interface file should now be your default.

- With modules it is truly "pay for what you use" (almost 0 cost to the consumer of the module)
- `constexpr` enables use cases you have not considered for compile-time computation
- A powerful technique is to compute as much as possible at compile-time, then continue computation at runtime from a known point
- These techniques are only possible if our core libraries are `constexpr` enabled.

Gone are the days of `#define`. `constexpr` should be your new default! Unfortunately, people over-complicate `constexpr`, so let's break down the simplest thing.

You need C++26 to get `constexpr` trig functions from your standard library.

If you see something like (I've seen in real code):

Figure 61. `static const` data known at compile time.

```
1  static const std::vector<int> angles{-90,-45,0,45,90};
```

This really needs to be:

Figure 62. Moving `static const` to `static constexpr`.

```
1  static constexpr std::array angles{-90,-45,0,45,90};
```

`static constexpr` here is necessary to make sure the object is not reinitialized each time the function / declaration is encountered. With `static` the variable lasts for the lifetime of the program, and we know it will be initialized exactly once.

The difference is threefold.

- The size of the array is now known at compile time
- We've removed dynamic allocations
- We no longer pay the cost of accessing a static

For globals in header files, prefer `inline constexpr` over `static constexpr` so that the linker merges data structures and reduces code bloat.

The main gains come from the first two, but we need a `constexpr` mindset to be looking for this kind of opportunity. We also need `constexpr` knowledge to see how to apply it in the more complex cases.

The difference can be significant.

There might be times where a non-`static constexpr` local variable is the most efficient option, because the data lives on the stack instead of a different section of the binary. Check if this matters during your optimization passes.

Technically non-`static constexpr` variables don't have be calculated at compile-time. However, it's almost certain that they are, and it is handy to think of them as calculated at compile-time.

Exercise: `constexpr` Your `const` Values

While reading code, look at all `const` values. Ask, "is this value known at compile time?" If it is, what would it take to make the value `constexpr?`

Exercise: `static constexpr` Your `static const` Values

Go through your current code base and look for code that is currently `static const`. You probably have something, somewhere.

- If it's currently `static const`, it's likely the size and data are known at compile time.
- Can this code become `constexpr?`
- What is preventing it from being `constexpr?`
- How much work would it take to modify the functions populating the `static const` data so that they are also `constexpr?`
- Remember that if it's a global in a header file, you should prefer `inline constexpr`.

Exercise: Make header code `constexpr`

If you have functions and types that are already defined in header files, try to make those functions and types fully `constexpr` enabled.

Exercise: Move header defined `constexpr` functions into module interface files

We want to avoid continuing to re-parse `constexpr` functions, and module interface files give us that way.

Exercise: Look for additional non-IO functions to make `constexpr`

At this point in your journey basically any non-IO function can be made into a `constexpr` function.

- Move these functions into your module interface file.
- Make the functions `constexpr`.

43.1: Resources

- C++Now 2017: Ben Deane & Jason Turner "`constexpr` ALL the things (a bit out of date with modern `constexpr` techniques)
- C++ Weekly Ep 233: `constexpr` map vs `std::map`
- Meeting C++ 2017: Jason Turner "Practical `constexpr`"
- C++ Russia 2019: Hana Dusíková "A state of compile time regular expressions"
- C++ Weekly Ep 312: Stop Using `constexpr` (And Use This Instead!)
- C++ Weekly: `constexpr` vs `static constexpr`
- C++ Weekly: `static constexpr` vs `inline constexpr`
- cons_expr: a compile-time capable scheme-like scripting language

44: Make globals in headers `inline constexpr`

1. Be afraid of global state!
2. Forget header files exist!

All global data should be `constexpr`

But global `constexpr` values are perfectly safe. They cannot mutate and they cannot affect “spooky action at a distance.”

If you still have header files because you haven’t completely moved over to modules yet…

All global `constexpr` values should be `inline constexpr`

Inside of module interfaces, `constexpr` globals and `inline constexpr` globals both have “external linkage” so we don’t have to worry about that.

Figure 63. static constexpr example

```
// my_library.hpp

// the object `dataset` will be duplicated in each .cpp
// file that includes this .hpp file
static constexpr auto dataset = make_data();
```

Figure 64. inline constexpr example

```
// my_library.hpp

// the object `dataset` will exist once in the entire binary
inline constexpr auto dataset = make_data();
```

45: `const` Everything That's Not `constexpr`

Many people (like Kate Gregory and James McNellis) have said this many times. Making objects `const` does two things:

1. It forces us to think about the initialization and lifetime of objects, which affects performance.
2. Communicates meaning to the readers of our code.

And as an aside, if it's a static object, the compiler is now free to move it into the constants portion of the binary, which can affect the optimizer.

Exercise: Look for `const` opportunities.

As you read through your code, you should look for variables that are not `const` and make them `const`.

- If a variable is not `const`, ask why not?
- Would using a lambda or adding a named function allow you to make the value `const`?

Figure 65

```
const auto data = [](){ // no parameters
  std::vector<int> result;
  // fill result with things.
  return result;
}(); // immediately invoked
```

Because of RVO, using a lambda will likely not add any overhead and may increase performance.

Did you make any `static` variables `const` in the process? Then go to the `constexpr` exercise.

`const` for values that are going to be returned can break implicit moves in some cases!

However, it's important to note that relying on implicit moves for return values and RVO can be a little fragile in general. Best is to simply not ever give a name to the object you are returning.

Figure 66

```
ReturnType some_function(int value)
{
  // if the types of result and ReturnType differ,
  // and an implicit conversion exists, you break
  // implicit moves.
  // If they are the same then you are hoping the
  // compiler applies NRVO, which doesn't always work
  // code with many branches
  const auto result = get_value(value + 42);
  return result;
}
```

Figure 67. Prefer not naming temporaries

```
ReturnType some_function(int value)
{
  // if the types of result and ReturnType differ,
  // and an implicit conversion exists, you get
  // implicit moves.
  // If they are the same, then guaranteed
  // copy/move elision applies from C++17
  return get_value(value + 42);
}
```

The clang-tidy No Automatic Move analysis is largely broken. It warns even when NRVO copy elision applies, and doesn't warn when it matters! (as of 2022-02-23). See this analysis: https://compiler-explorer.com/z/a4K76nbhq.

You probably don't want to make `class` members `const`; it can break essential things such as move construction and move assignment, and sometimes silently.

45.1: Resources

- CppCon 2014: James McNellis & Kate Gregory "Modernizing Legacy C++ Code"
- CppCon 2019: Jason Turner "C++ Code Smells"
- The implication of const or reference member variables in C++
- C++Now 2018: Ben Deane "Easy to Use, Hard to Misuse: Declarative Style in C++" (Builds on techniques that make applying `const` easier.)

46: Always Initialize Your non-`const`, non-`auto` Values

The ideal is to `const` everything, which forces you to initialize. However, that's not always possible.

Similarly, if you use `auto`, you are forced to initialize an object.

- The compiler will "throw away" operations on uninitialized values
- Make sure you have your `-Wunitialized` style warnings enabled

Be aware that there can be a difference between default initialization and initialization that appears empty in some cases.

Compilers are not always perfect at catching uninitialized value usage. For example, GCC (as of 2023-12-30) requires optimizations enabled to catch this uninitialized variable access, which is UB.

Figure 68. Uninitialized variable read that GCC doesn't catch without optimizations enabled

```
#include <span>

float sum(std::span<float> values)
{
  float result;

  for (const auto f : values) {
    result += f;
  }

  return result;
}
```

Exercise: Understand which constructor you are calling

Take this example code and play with it at various optimization levels in compiler-explorer.com to understand the difference that each constructor call might perform.

Figure 69. which constructor is called?

```
import std;

int main()
{
  // std::string is not trivially constructible, so this calls
  // the default constructor and it is initialized to ""
  std::string str;

  // explicitly call the default constructor
  std::string str2{};

  // call the constructor that takes a `const char *`
  // try commenting this out as you play with various
  // optimization levels
  std::string str3 = "";
}
```

Exercise: Make sure `-Wuninitialized` is not disabled, and take it seriously.

46.1: Tools

- Valgrind's Memcheck
- Dr Memory

46.2: Resources

- C++ Weekly - Ep 257 - Garbage In, Garbage Out - Why Initialization Matters
- COVID-19 Research and Uninitialized Variable
- Fuzzing Image Parsing in Windows, Part Two: Uninitialized Memory

47: Prefer `auto` in Many Cases.

I'm not an Almost Always Auto (AAA) person, but let me ask you this: What is the result type of `std::count`?

My answer is, "I don't care."

Figure 70. `const auto`

```
1  const auto result = std::count( /* stuff */ );
```

or, if you prefer:

Figure 71. `auto const`

```
1  auto const result = std::count( /* stuff */ );
```

Using auto avoids unnecessary conversions and data loss. Same as ranged-for loops. `auto` requires initialization, the same as `const`, the same reasoning for why that's good.

Figure 72. auto requires initializtion

```
1  auto i; // cannot compile
2  auto i = int; // cannot compile
```

Example:

Figure 73. Possible expensive conversion.

```
1  const std::string value = get_string_value();
```

What is the return type of `get_string_value()`? If it is `std::string_view` or `const char *`, we will get a potentially costly conversion on all compilers with no diagnostic.

Figure 74. No possible expensive conversion.

```
1  // avoids conversion
2  const auto value = get_string_value();
```

Furthermore, `auto` return types actually can significantly simplify generic code.

Figure 75. C++ 98 template usage.

```
1  // our example from "Don't Be Afraid of Templates"
2  template<typename Arithmetic>
3  Arithmetic divide(Arithmetic numerator, Arithmetic denominator) {
4    return numerator / denominator;
5  }
```

This code forces us to use the same type for both the numerator and denominator (play with this and see the weird compile errors you get).

Figure 76. C++ 98 template made more generic?

```
1  template<typename Numerator, typename Denominator>
2  /*what's the return type*/
3  divide(Numerator numerator, Denominator denominator) {
4    return numerator / denominator;
5  }
```

C++98 provides no solution to this problem, but C++11 does.

Figure 77. C++11 trailing return types.

```
1  // use trailing return type
2  template<typename Numerator, typename Denominator>
3  auto divide(Numerator numerator, Denominator denominator)
4    -> decltype(numerator / denominator)
5  {
6    return numerator / denominator;
7  }
```

But in C++14, we can leave off the return type altogether (remember to Skip C++11).

Figure 78. C++14 `auto` return types.

```
template<typename Numerator, typename Denominator>
auto divide(Numerator numerator, Denominator denominator)
{
  return numerator / denominator;
}
```

In C++20 we can simplify this code further with `auto` function parameters (which create implicit templates for us)

Figure 79. C++20 `auto` parameters.

```
auto divide(auto numerator, auto denominator)
{
  return numerator / denominator;
}
```

Consider constraining these parameters if it makes sense for your application

Figure 80. Constrained auto parameters.

```
// this should also be constexpr and [[nodiscard]]
std::floating_point auto divide(
    std::floating_point auto numerator, std::floating_point auto denomi\
nator)
{
  return numerator / denominator;
}
```

Exercise: Become familiar with `auto` deduction.

Figure 81. Ex1: what is the type of val?

```
1  const int *get();
2
3  int main() {
4    const auto val = get();
5  }
```

Figure 82. Ex2: what is the type of val?

```
1  const int &get();
2
3  int main() {
4    const auto val = get();
5  }
```

Figure 83. Ex3: what is the type of val?

```
1  const int *get();
2
3  int main() {
4    const auto *val = get();
5  }
```

Figure 84. Ex4: what is the type of val?

```
1  const int &get();
2
3  int main() {
4    const auto &val = get();
5  }
```

Figure 85. Ex5: what is the type of val?

```
1  const int *get();
2
3  int main() {
4    const auto &val = get();
5  }
```

Figure 86. Ex6: what is the type of val?

```
1  const int &get();
2
3  int main() {
4    const auto &&val = get();
5  }
```

Exercise: Build your experiment library

The above exercise is perfect for building into a set of experiments that are saved in your GitHub gists mentioned in C++ Is Not Magic

Exercise: Understand how `auto` and `template` deduction relate

Understand the rules for type deduction of templates and how they relate to `auto`. Read the section in the C++ Programming Language Standard [dcl.spec.auto].

47.1: Resources

- clang-tidy modernize-use-auto
- Almost Always Auto

48: Use Ranges and Views For Correctness and Readability

At their best C++20's ranges and views can drastically increase readability and correctness of your code without impacting performance or compile times.

At their worst C++20's ranges and views can drastically increase compile times, runtimes, and affect the readability of your code.

Because of some of the potential drawbacks to ranges and views, there are some who recommend never using them. This book does not call for that. Instead, we say that you should learn their strengths and weaknesses.

This author considers the humble case of "loop over all elements except for the first item" to be the "killer feature" of ranges and views.

Figure 87. Skip first element, without ranges.

```
1  void print_all_but_first(const std::vector<int> &values) {
2    // forget this and you have UB with 0 element container
3    if (values.empty()) { return; }
4
5    for (auto itr = next(begin(values)); itr != end(values); ++itr) {
6      std::println("{}", *itr);
7    }
8  }
```

Figure 88. Skip first element, with ranges.

```
1  void print_all_but_first(const std::vector<int> &values) {
2    for (const auto val : values | std::views::drop(1)) {
3      std::println("{}", val);
4    }
5  }
```

Be aware of these very simple but very powerful use cases, besides the much more complex and composable features of the ranges library.

Exercise: Read through the set of views available https://en.cppreference.com/w/cpp/ranges.

Exercise: Use compiler explorer to compare and contrast the two functions above at different optimization levels.

48.1: Resources

- C++ Weekly - Ep 391 - Finally! C++23's std::views::enumerate
- C++ Weekly - Ep 398 - C++23's zip_view
- C++ Weekly - Ep 399 - C++23's slide_view vs adjacent_view
- C++ Weekly - Ep 401 - C++23's chunk view and stride view

- Effective Ranges: A Tutorial for Using C++2x Ranges - Jeff Garland - CppCon 2023

49: Don’t Reuse Views

Range views can hold state in unexpected ways, which can result in unexpected code evaluation.

Figure 89. Reuse of drop, with caching effects

```
import std;

int main()
{
    std::list<int> values{1,2,3,4,5,6,7,8,9};

    auto drop_2 = values | std::views::drop(2);

    std::println("{}", drop_2);

    values.erase(values.begin());

    std::println("{}", drop_2);
}
```

You might be surprised to know that the above code prints:

Figure 90. Reuse of drop output, with list.

```
> 3, 4, 5, 6, 7, 8, 9
> 3, 4, 5, 6, 7, 8, 9
```

However, if we change the code to use `std::vector`, we get the expected output.

Figure 91. Reuse of drop, without caching effects

```
import std;

int main()
{
    std::vector<int> values{1,2,3,4,5,6,7,8,9};

    auto drop_2 = values | std::views::drop(2);

    std::println("{}", drop_2);

    values.erase(values.begin());

    std::println("{}", drop_2);
}
```

Figure 92. Reuse of drop output, with vector.

```
> 3, 4, 5, 6, 7, 8, 9
> 4, 5, 6, 7, 8, 9
```

Some people would tell you to avoid views entirely, because of issues like this.

We will take a more judicious approach.

Don’t reuse views!

The relatively simple solution to this problem:

Figure 93. Using a lambda to create the view.

```
import std;

int main()
{
    std::list<int> values{1,2,3,4,5,6,7,8,9};

    auto drop_2 = []{ return values | std::views::drop(2); };

    std::println("{}", drop_2());
```

```
10
11      values.erase(values.begin());
12
13      std::println("{}", drop_2());
14  }
```

49.1: Resources

- Denver C++ Meetup: 2023-04 - Tyler Weaver - Reading Ranges (Don’t Take My Word For It)
- Belle Views on C++ Ranges, their Details and the Devil - Nico Josuttis - Keynote Meeting C++ 2022

50: Prefer Algorithms Over Loops

Algorithms communicate meaning and help us apply the "`const` All The Things" rule. In C++20, we get ranges, which make algorithms more comfortable to use.

It's possible, taking a functional approach and using algorithms, that we can write C++ that reads like a sentence.

Figure 94. Algorithms with ranges

```
1  const auto has_value
2    = std::any_of(container, greater_than(12));
```

Figure 95. Algorithms with explicit iterators

```
1  const auto has_value
2    = std::any_of(begin(container), end(container),
3                  greater_than(12));
```

Note that in some rare cases, your static analysis tools might be able to suggest an algorithm to use.

Exercise: Study existing loops

Next time you are reading through a loop in your codebase, cross-reference it with the C++ `<algorithm>` header and try to find an algorithm that applies instead.

This book only barely mentions C++20's ranges. Compilers are just now getting support for ranges as of the publication of this book. Ranges can be composed and have full support for `constexpr`.

50.1: Resources

- GoingNative 2013: Sean Parent "C++ Seasoning"
- CppCon 2018: Jonathan Boccara "105 Algorithms in Less Than an Hour"
- C++ Now 2019: Conor Hoekstra "Algorithm Intuition"
- MeetingC++ 2019: Conor Hoekstra "Better Algorithm Intuition"
- Conor Hoekstra "The Twin Algorithms"
- C++ Weekly Ep 187 "C++20's `constexpr` Algorithms"
- C++ Weekly Ep 105 "Learning "Modern" C++ 5: Looping And Algorithms
- Algorithm Selection

51: Use Ranged-For Loops When Views and Algorithms Cannot Help

(Carefully) prefer ranges, views, and algorithms, but use ranged-for loops as your next possible option.

Figure 96. `int` vs `std::size_t` when looping.

```
for (int i = 0; i < container.size(); ++i) {
  // oops mismatched types
}
```

Figure 97. Mismatched containers while looping.

```
for (auto itr = container.begin();
     itr != container2.end();
     ++itr) {
  // oops, most of us have done this at some point
}
```

Figure 98. Example of ranged-for loop.

```
for(const auto &element : container) {
  // eliminates both other problems
}
```

Never mutate the container itself while iterating inside of a ranged-for loop.

Exercise: Modernize Your Loops

You probably have old-style loops in your code.

1. Apply clang-tidy's modernize-loop-convert check.
2. Look for loops that could not be converted.
    - Loops that could not be converted might represent bugs in the code
    - Loops that could not be converted, but do not have bugs, are good candidates for simplification

51.1: Resources

- clang-tidy modernize-loop-convert

52: Use `auto` in ranged for loops

Not using `auto` can make it easier to have silent mistakes in your code.

Figure 99. Accidental conversions

```
for (const int value : container_of_double) {
  // accidental conversion, possible warning
}
```

Figure 100. Accidental slicing

```
for (const base value : container_of_derived) {
  // accidental silent slicing
}
```

Figure 101. No problem

```
for (const auto &value : container) {
  // no possible accidental conversion
}
```

Prefer:

- `const auto &` for non-mutating loops
- `auto &` for mutating loops
- `auto &&` only when you have to work with weird types like `std::vector<bool>`, or if moving elements out of the container

Exercise: Understand `std::map` and ranged `for` loops

Understand what this code is doing. Is it making a copy? Why and how?

Figure 102. Accidental copy?

```
1 std::map<std::string, int> get_map();
2
3 using element_type = std::pair<std::string, int>;
4
5 for (const element_type & : get_map())
6 {
7 }
```

Modern compilers can directly catch the above issue, but you have to enable your warnings to see it!

Exercise: Enable ranged-loop related warnings

Make sure `-Wrange-loop-construct` is enabled in your code, which is automatically included with `-Wall`.

53: Avoid `default` In `switch` Statements

This is an issue that is best described with a series of examples. Starting from this one:

Figure 103. `switch` with warnings

```
enum class Values {
  val1,
  val2
};

[[nodiscard]] constexpr std::string_view get_name(Values value) {
  switch (value) {
  case Values::val1: return "val1";
  case Values::val2: return "val2";
  }
}
```

If you have enabled all of your warnings, then you will likely get a “not all code paths return a value” warning here. Which is technically correct. We could call `get_name(static_cast<Values>(15))` and not violate any part of C++ [dcl.enum/5] except for the Undefined Behavior of not returning a value from a function.

You'll be tempted to fix this code like this:

Figure 104. `switch` with `default` to avoid warnings

```
enum class Values {
  val1,
  val2
};

[[nodiscard]] constexpr std::string_view get_name(Values value) {
  switch (value) {
  case Values::val1: return "val1";
  case Values::val2: return "val2";
  default: return "unknown";
  }
}
```

But this introduces a new problem

Figure 105. Unhandled case

```
enum class Values {
  val1,
  val2,
  val3 // added a new value
};

[[nodiscard]] constexpr std::string_view get_name(Values value) {
  switch (value) {
  case Values::val1: return "val1";
  case Values::val2: return "val2";
  default: return "unknown";
  }
  // no compiler diagnostic that `val3` is unhandled
}
```

Instead, prefer code like this:

Figure 106. Prefered version

```
enum class Values {
  val1,
  val2,
  val3 // added a new value
};

[[nodiscard]] constexpr std::string_view get_name(Values value) {
  switch (value) {
  case Values::val1: return "val1";
  case Values::val2: return "val2";
  } // unhandled enum value warning now

  return "unknown";
}
```

You shouldn't ever get an "unreachable code" warning in the above example because the range of valid values is nearly always larger than the values you have defined.

Some modern tools can detect these uses of `default` for you.

Exercise: Look for `default:`

What do you find in your code base? Did enabling warnings in previous exercises find uses of `default:` for you already?

Exercise: Consider `std::unreachable`.

`std::unreachable` (added in C++23) explicitly invokes Undefined Behavior if it is executed. It can be used for the “unreachable” case of a switch statement, to ensure that the compiler does the optimal thing.

This is potentially frightening. Do you really want undefined behavior because it’s “impossible” for code to be reached? Discuss this with your coworkers and consider where / when / if you would use this tool. See the C++ Weekly episode in the resources for more details.

53.1: Resources

- CppCon 2018: Jason Turner “Applied Best Practices”
- C++ Weekly - Ep 393 - C++23’s std::unreachable* `-Wswitch-enum`
- `-Wswitch`

54: Prefer Scoped `enums`

C++11 introduced scoped enumerations, intended to solve many of the common problems with `enum` inherited from C.

Figure 107. C++98 `enum`s

```
enum Choices {
  option1 // value in the global scope
};

enum OtherChoices {
  option2
};

int main() {
  int val = option1;
  val = option2; // no warning
}
```

- `enum Choices;`
- `enum OtherChoices;`

These two can easily get mixed up, and they each introduce identifiers in the global namespace.

- `enum class Choices;`
- `enum class OtherChoices;`

The values in these enumerations are scoped and more strongly typed.

Figure 108. C++11 scoped enumeration.

```
enum class Choices {
  option1
};

enum class OtherChoices {
  option2
};

int main() {
  int val = option1; // cannot compile, need scope
  int val2 = Choices::option1; // cannot compile, wrong type
  Choices val = Choices::option1; // compiles
  val = OtherChoices::option2; // cannot compile, wrong type
}
```

These `enum class` versions cannot get mixed up without much effort, and their identifiers are now scoped, not global.

> `enum struct` and `enum class` are equivalent. Logically `enum struct` makes more sense since they are public names. Which do you prefer?

Exercise: `enum struct` or `enum class`

Decide if you prefer `enum struct` or `enum class` and develop a well-reasoned answer as to why.

Moving to scoped enumerations will probably find many bugs in your code!

54.1: Resources

- CppCon 2018: Victor Ciura “Better Tools in Your Clang Toolbox” (Discusses bugs found by moving to `enum class`)
- cppreference.com Enumeration Declaration

55: Prefer `if constexpr` over SFINAE

SFINAE is kind-of write-only code. `if constexpr` doesn't have the same flexibility, but use it when you can.

Let's take our divide example last seen in Prefer `auto` in Many Cases:

Figure 109. C++14 divides template.

```
1  template<typename Numerator, typename Denominator>
2  auto divide(Numerator numerator, Denominator denominator)
3  {
4    return numerator / denominator;
5  }
```

We now want to add different behavior if we are doing integral division. Before C++17, we would have used SFINAE ("Substitution Failure Is Not An Error"). Essentially this means that if a function fails to compile, then it is removed from overload resolution.

Figure 110. SFINAE `divide` function.

```
#include <stdexcept>
#include <type_traits>
#include <utility>

template <typename Numerator, typename Denominator,
          std::enable_if_t<std::is_integral_v<Numerator> &&
                           std::is_integral_v<Denominator>,
                           int> = 0>
auto divide(Numerator numerator, Denominator denominator) {
  // is integer division
  if (denominator == 0) {
    throw std::runtime_error("divide by 0!");
  }
  return numerator / denominator;
}

template <typename Numerator, typename Denominator,
          std::enable_if_t<std::is_floating_point_v<Numerator> ||
                           std::is_floating_point_v<Denominator>,
                           int> = 0>
auto divide(Numerator numerator, Denominator denominator) {
  // is floating point division
  return numerator / denominator;
}
```

The `if constexpr` construct in C++17 can simplify this code:

Figure 111. `if constexpr` option for compile time behavior change.

```
import std;

// note that we could use `auto` for numerator and denominator
// but it would actually complicate the if constexpr code
// by requiring that we use `decltype` to get the type info
// back.
template <typename Numerator, typename Denominator>
[[nodiscard]] constexpr auto divide(Numerator numerator, Denominator de\
nominator) {
  if constexpr (std::is_integral_v<Numerator> &&
                std::is_integral_v<Denominator>) {
    // is integral division
    if (denominator == 0) {
        throw std::runtime_error("divide by 0!");
    }
  }

  return numerator / denominator;
}
```

Consider concepts over `if constexpr` for readability in many cases

Figure 112. Constraints instead of `if constexpr`

```
import std;

// default overload chosen
[[nodiscard]] constexpr auto divide(auto numerator, auto denominator) {
  return numerator / denominator;
}

// version called only if both parameters are integral
[[nodiscard]] constexpr auto divide(
  std::integral auto numerator, std::integral auto denominator) {

  // is integral division
  if (denominator == 0) {
    throw std::runtime_error("divide by 0!");
  }

  return numerator / denominator;
}
```

The code inside the `if constexpr` block must still be syntactically correct. `if constexpr` is not the same as a `#define`.

Code that you might normally choose to put outside of the `if` block might now need to live inside of the `else` to make sure it is not instantiated with invalid types

Exercise: Look into “design by introspection”

The combination of `if constexpr`, concepts, and `requires` results in a very powerful, almost reflection-like capability.

Figure 113. 'Design by Introspection' example

```
constexpr void add_values(auto &container, auto first, auto second)
{
  // we ask, at compile time, if this container
  // has a reserve and a size member.
  if constexpr(requires {container.reserve(container.size() + 2); }) {
    // if this is a valid operation, then perform it at runtime
    container.reserve(container.size() + 2);
  }

  // either way, add the values
  container.push_back(first);
  container.push_back(second);
}
```

This technique was first proposed by Andrei Alexandrescu in 2001, then brought into C++20 by Kris Jusiak.

55.1: Resources

- C++ Weekly Special Edition: Using C++17’s constexpr if
- C++ Weekly - Ep 122 - `constexpr` with `optional` and `variant`
- CppCon 2017: Jason Turner “Practical C++17”
- C++ Weekly - Ep 242 - Design By Introspection in C++20
- C++17 In Tony Tables: `constexpr if`

56: De-template-ize Your Generic Code

Move things outside of your templates when you can. Use other functions. Use base classes. The compiler is still free to inline them or leave them out of line.

De-template-ization will improve compile times and reduce binary sizes. Both are helpful. It also eliminates the thing that people think of as “template code bloat” (which IMO doesn’t exist) (article formatting got broken at some point, sorry).

Figure 114. A new lambda for each function template instantiation.

```
template<typename T>
constexpr void do_things()
{
  // this lambda must be generated for each
  // template instantiation
  auto lambda = [](){ /* some lambda that doesn't capture */ };
  auto value = lambda();
}
```

Compared to:

Figure 115. Shared logic between template instantiations.

```
constexpr auto some_function() { /* do things*/ }

template<typename T>
constexpr void do_things()
{
  auto value = some_function();
}
```

Now only one version of the inner logic is compiled, and it’s up to the compiler to decide if they should be inlined.

Similar techniques apply to base classes and templated derived classes.

Exercise: Bloaty McBloatface and `-ftime-trace`.

We’re getting more and more tools available to look for bloat in our binaries and analyze compile times. Look into these tools and other tools available on your platform.

Run them against your binary and see what you find.

When using clang’s `-ftime-trace`, also look into ClangBuildAnalyzer.

56.1: Resources

- Templight
- C++ Weekly Ep 89: “Overusing Lambdas”
- C++ Weekly Christmas Class 2019 - Chapter 3 (This is the first episode of chapter 3, and it introduces the question of how and why two different options differ. The next several episodes in that playlist give some background, and the start of chapter 4 gives the answers. It is very much related to template bloat questions.)
- Effective C++ (3rd Edition) Item 44 - Factor parameter-independent code out of templates

57: Use Lippincott Functions

Same arguments as de-template-izing your code: This is a do-not-repeat-yourself principle for exception handling routines.

If you have many different exception types to handle, you might have code that looks like this:

Figure 116. Duplicated exception handling.

```
1  void use_thing() {
2    try {
3      do_thing();
4    } catch (const std::runtime_error &) {
5      // handle it
6    } catch (const std::exception &) {
7      // handle it
8    }
9  }
10
11 void use_other_thing() {
12   try {
13     do_other_thing();
14   } catch (const std::runtime_error &) {
15     // handle it
16   } catch (const std::exception &) {
17     // handle it
18   }
19 }
```

A Lippincott function (named after Lisa Lippincott) provides a centralized exception handling routine.

Figure 117. Lippincott de-duplicated exception handling.

```
1  void handle_exception() {
2    try {
3      throw; // re-throw exception already in flight
4    } catch (const std::runtime_error &) {
5    } catch (const std::exception &) {  }
6  }
7
8  void use_thing() {
9    try {
10     do_thing();
11   } catch (...) {
12     handle_exception();
13   }
14 }
15
16 void use_other_thing() {
17   try {
18     do_other_thing();
19   } catch (...) {
20     handle_exception();
21   }
22 }
```

This technique is not new - it has been available since the pre-C++98 days.

Exercise: Do You Use Exceptions?

If your project uses exceptions, there's probably some ground for simplifying and centralizing your error handling routines. If it does not use exceptions, then you likely have other types of error handling routines that are duplicated. Can these be simplified?

57.1: Resources

- C++ Secrets: Using a Lippincott Function for Centralized Exception Handling
- C++ Weekly Ep 91: Using Lippincott Functions

58: No More `new`!

You're already avoiding the heap and using smart pointers for resource management, right?!

Take this to the next level and be sure to use `std::make_unique<>()` (C++14) in the rare cases that you need the heap.

In the very rare cases you need shared ownership, use `std::make_shared<>()` (C++11).

Exercise: Do you use Qt or some other widget library?

Have you ever thought about writing your own `make_qobject` helper? Give it the semantics you need and be sure to use `[[nodiscard]]`.

In any case, you can limit your use of `new` to a few core library helper functions.

Exercise: Use clang-tidy modernize fixes.

With clang-tidy, you can automatically convert `new` statements into `make_unique<>` and `make_shared<>` calls. Be sure to use `-fix` to apply the change after it's been discovered.

58.1: Resources

- clang-tidy modernize-make-shared
- clang-tidy modernize-make-unique

59: Avoid `std::bind` and `std::function`

While compilers continue to improve and the optimizers work around these types' complexity, it's still very possible for either to add considerable compile-time and runtime overhead.

C++14 lambdas, with generalized capture expressions, are capable of the same things that `std::bind` is capable of.

`std::function` is not `constexpr` enabled, which is another reason it might need to be avoided in your code.

Figure 118. `std::bind` to change parameter order

```
 1 import std;
 2
 3 [[nodiscard]] constexpr double divide(double numerator, double denomina\
 4 tor) {
 5   return numerator / denominator;
 6 }
 7
 8 auto inverted_divide = std::bind(divide,
 9                                   std::placeholders::_2,
10                                   std::placeholders::_1);
```

Figure 119. Lambda to change parameter order

```
import std;

[[nodiscard]] constexpr double divide(double numerator, double denomina\
tor) {
  return numerator / denominator;
}

auto inverted_divide = [](const auto numerator,
                          const auto denominator) {
  return divide(denominator, numerator)
}
```

Exercise: Compare the possibilities.

Take these options in Compiler Explorer. How do the compile times and resulting assembly look?

Figure 120. `std::function` and `std::bind`

```
import std;

// std::function is not constexpr enabled, so this
// cannot be `constexpr`
[[nodiscard]] std::function<int (int)> bind_3(auto func)
{
  return std::bind(func, std::placeholders::_1, 3);
}

int main(int argc, const char *[])
{
  return bind_3(std::plus<>{})(argc);
}
```

Figure 121. `std::bind` only, for bonus points, what type is returned from the function `bind_3`?

```
#include <functional>

// std::bind is constexpr enabled
[[nodiscard]] constexpr auto bind_3(auto func)
{
  return std::bind(func, std::placeholders::_1, 3);
}

int main(int argc, const char *[])
{
  return bind_3(std::plus<>{})(argc);
}
```

Figure 122. Only lambdas, no std library wrappers.

```
#include <functional>

[[nodiscard]] constexpr auto bind_3(auto func)
{
  return [func](const int value){ return func(value, 3); };
}

int main(int argc, const char *[])
{
  return bind_3(std::plus<>{})(argc);
}
```

Exercise: Look at std::bind_front, std::bind_back

Figure 123. C++23's bind_back

```
#include <functional>

[[nodiscard]] constexpr auto bind_3(auto func)
{
  return std::bind_back(func, 3);
```

```
 6 }
 7
 8 int main(int argc, const char *[])
 9 {
10   return bind_3(std::plus<>{})(argc);
11 }
```

59.1: Resources

- CppCon 2015: Stephan T. Lavavej ": What's New, And Proper Usage"
- C++ Weekly Ep 16: "Avoiding `std::bind`"

60: Don’t Use `initializer_list` For Non-Trivial Types

“Initializer List” is an overloaded term in C++. “Initializer Lists” are used to directly initialize values. `initializer_list` is used to pass a list of values to a function or constructor.

Exercise: Understand the overhead `initializer_list` can bring

Use Andreas Fertig’s awesome cppinsights.io to understand what these two examples do

Figure 124. `initializer_list` constructor with `shared_ptr`.

```
#include <vector>
#include <memory>

std::vector<std::shared_ptr<int>> vec{
  std::make_shared<int>(40), std::make_shared<int>(2)
};
```

Figure 125. `std::array` construction with `shared_ptr`.

```
#include <array>
#include <memory>

std::array<std::shared_ptr<int>, 2> data{
  std::make_shared<int>(40), std::make_shared<int>(2)
};
```

And explain the difference. If you can do this, you understand more than most C++ developers.

Exercise: Understand why this doesn’t compile

Figure 126. `initializer_list` construction with `unique_ptr`.

```
#include <vector>
#include <memory>

std::vector<std::unique_ptr<int>> data{
  std::make_unique<int>(40), std::make_unique<int>(2)
};
```

60.1: Resources

- C++Now 2018: Jason Turner “Initializer Lists Are Broken, Let’s Fix Them” (deep dive into the issues around these topics)
- C++ Insights

61: Consider Designated Initializers (C++20)

Direct-Initialization provides a highly efficient way of initializing public data members.

Figure 127. direct-init example

```
#include <string>

struct Data
{
  std::string first;
  std::string second;
};

int main()
{
  // directly-initialize the data members `first` and `second`
  // this has no copy or move overhead nor questions
  // about how to write an efficient constructor for it.
  Data d{"Hello", "World"};
}
```

The downside is that this code is not very readable. We don't know what those two parameters `"Hello"` and `"World"` are initializing.

C++20 added "designated initializers" that allow you to specify the name of the object you are initializing.

- They must be initialized in order
- Items may be skipped
- Names must be consistently provided

Figure 128. designated-init example

```
#include <string>

struct Data
{
  std::string first;
  std::string second;
};

int main()
{
  Data d{.first = "Hello", .second = "World"};
}
```

Important: compilers are inconsistent about warning if you have left out a parameter when using designated initializers. Make sure you compile with multiple compilers and high warning levels.

Exercise: Discuss with your team what you prefer

- Is being able to skip a parameter an upside, or a downside?
- How important is it that parameters have a name?
- Does it matter if you have strong typing as a discipline in your system?

Exercise: Compare a simple `struct` with public members to one with private and constructors.

Figure 129. implement a constructor and getters

```cpp
#include <string>

class Data
{
public:
  // add constructor(s), getters and setters
  // * Is this better?
  // * How much compile-time and run-time
  //   overhead to does add?
private:
  std::string first;
  std::string second;
};
```

In many cases simple public structs are better than complex classes with getters, setters, and constructors. But this is only true if there are no interdependencies between the values. (invariants)

61.1: Resources

- C++ Weekly Ep 127: C++20's Designated Initializers
- C++ Weekly Ep 274 - Why Is My Pair 310x Faster Than `std::pair`?
- C++ Stories: C++20: Designated Initializers
- Modernes C++: Designated Initializers
- Abseil Tip of the Week #172: Designated Initializers

Part V: Bonus Chapters

62: Improving Build Time

A few practical considerations for making build time less painful

- De-template-ize your code where possible
- Use forward declarations where it makes sense to
- Enable PCH (precompiled headers) in your build system
- Use ccache or similar (many other options that change regularly, Google for them)
- Be aware of unity builds
- Know the possibilities and limitations of `extern template`
- Use a build analysis tool to see where build time is spent

62.1: Use an IDE

This is the most surprising side effect of using a modern IDE that I have observed: IDE's do realtime analysis of the code. Realtime analysis means that you know as you are typing if the code is going to compile. Therefore, you spend less time waiting for builds.

Exercise: What are build times costing you?

Try to figure out how much build times are costing in developer time and see how much could be saved if build times were lessened.

62.2: Resources

- A guide to unity builds
- Unity builds with Meson
- Unity builds with CMake
- PCH with Meson
- PCH with CMake
- ccache
- CMake Compiler Launcher
- Clang Build Analyzer
- Getting started with C++ Build Insights
- Introducing vcperf /timetrace for C++ build time analysis

63: Continue Your C++ Education

You must continually learn if you want to become better at what you do, and many resources are available to you to continue your C++ education.

63.1: Know How To Ask Questions

Kate Gregory has published an excellent article on how to ask questions.

Some key points are:

- Don't use screenshots
- Use good variable names
- Add some tests
- Listen to what people are telling you

63.2: Conferences And Local User Groups

There is almost certainly one near you. It's a great way to network and learn new things. Check out the ISO C++ Conferences Worldwide List and Meeting C++'s User Groups List.

I am finishing this book during the global COVID-19 pandemic. So conferences and user groups are mostly on hold right now. But this presents an attractive new opportunity for many.

Many of those conferences and user groups are now meeting online. It's now possible for us all to attend each other's user groups. The North Denver Metro C++ Meetup, for example, regularly has one attendee from Thailand each month.

Note from the author: when interacting with the C++ community remember to treat others with dignity and respect; be patient; take time to understand the rules and norms of the particular community with which you are interacting.

63.3: C++ Weekly

This book references C++ Weekly throughout as a resource to go back to for more information and examples to share with your coworkers. At this moment, the show has been going for 235 weeks straight with many special editions, extras, and live streams.

63.4: cppreference.com

The website is fantastic, but you might not know that you can create an account and customize the content to the version of C++ you are using. Also, you can execute examples and download an offline version!

63.5: Hire a Trainer to Come Onsite for Your Company

Team training gets your team thinking in a new direction, improves morale, and boosts employee retention. Since you made it this far, I'm going to offer you a coupon.

If you mention this book, you'll get 10% off onsite training costs at your company from me. (travel costs not discounted). Hopefully, travel restrictions will not last much longer.

63.6: YouTube

- C++ Weekly (Author's Channel)
- Andreas Fertig's Channel
- C++ on Sea
- C++Now
- code_report
- code::dive
- CopperSpice
- Core C++
- CppCon
- CppNorth

63.7: Podcasts

- CppCast

63.8: Blogs and Useful Websites

- The Pasture
- eel.is / current C++ draft
- wg21.link / links to papers and standards
- C++ Stories
- Fluent C++

64: Thank You

64.1: Sponsors

Thank you to all of my Book Supporter patrons who helped make this book possible!

Adam Albright, Adam P Shield, Alejandro Lucena, Alexander Roper, Ali Raein, Andrei Sebastian Cîmpean, Anton Smyk, Arman Imani, Ashley Gay, Bill Baker, Björn Fahller, Brendan Nolan, Cem Dervis, Clint Rajaniemi, Cooper Healy, Corentin Gay, David C Black, David Poole, Dennis Börm, Emyr Williams, Fedor Alekseev, Ferdinand Stapenhorst, Florian Sommer, Gwendolyn Hunt, Ivan Pakhomov, Jack Glass, Jaewon Jung, Jakub Sanestrzik, Jeff Bakst, jimmy, Jonathan Watmough, Kacper Kołodziej, Kedar Bhat, Kevin Stone, Kitty Raven, Lars Ove Larsen, Luke Valenty, Magnus Westin, Marcin Zdun, Mark Guidarelli, Martin Hammerchmidt, Matt Godbolt, Matthew Guidry, Michael Pearce, Michael Pettit, michel morel, Mo Xiaoming, Namgoo Lee, Natalya Kochanova, Olafur Waage, Panos Gourgaris, Pi, Ralph Jeffrey Steinhagen, Reiner Eiteljoerge, royaltrashfire, Samuel Egger, Sebastian Raaphorst, Sergii Lovygin, Sergii Zaiets, Silver, Stefan Goetschi, Tim Butler, Tobias Dieterich, Tomasz Cwik, Volker Schwaberow, Wenbo, William Hawkins, Y, Yacob Cohen-Arazi, Yang, Ólafur Waage, Šimon Bařinka

64.2: Reviewers of C++17/20 Edition

Craig Scott and Alexander Roper, thank you for extensive notes and feedback during prerelease.

65: Bonus: Understand The Lambda

A surprising complexity hides behind the simple lambda of C++. Initially added in C++11, it was initially constrained. With each version of C++, the lambda becomes more flexible and powerful.

Lambdas reverse some of the defaults from the rest of C++. Default `const` and automatically `constexpr` when possible; they give us some of what we wish the rest of the language could have.

Figure 130. Lambda grammar.

```
lambda-expression:
  lambda-introducer lambda-declarator(opt) compound-statement
  lambda-introducer < template-parameter-list > requires-clause(opt)
       lambda-declarator(opt) compound-statement
lambda-introducer:
  [ lambda-capture(opt) ]
lambda-declarator:
  ( parameter-declaration-clause ) decl-specifier-seq(opt)
       noexcept-specifier(opt) attribute-specifier-seq(opt)
       trailing-return-type(opt) requires-clause(opt)
```

If you can read standard-eze, you can dig into all of the features of C++20's lambdas yourself.

Figure 131. Allowed lambdas as of C++20.

```
// valid empty lambda, does nothing
[]{};
// optional to have parameter list
[](){};
// C++17 explicit constexpr and void return
[]() constexpr -> void {};
// immediately invoked lambda
auto i = [](){ return 42; }();
// Not allowed before C++17, because constexpr
constexpr auto j = []{ return 42; }();
// generic lambda, C++14
[](auto x){ return x + 42; };
// variadic lambda, C++14
[](auto ... x){ return std::vector<int>(x...); };
// capture by copy, C++11
[i](){ return i + 42; };
// generalized capture, C++14 (what's the type of i?)
[i = 42]{ return i + 42; };
// stateful lambda, C++11
[i]() mutable { return ++i; };
// explicit template, C++20
[]<typename T>(T x){ return x + 42; };

// C++14 generic lambda returning a C++20 lambda with variadic
// capture expression which returns a fold expression summation
// of the captured values.
[](auto ... val){ return [...val = val]{ return (val + ...); }; };
```

If you understand every aspect of C++'s lambdas and how the compiler implements them, you know everything important about C++.

This is why I put together my C++ class on YouTube about lambdas.

In 2018 when compilers first started supporting C++20's new lambdas, I implemented this mostly standards-compliant version of `std::bind` using lambdas.

(Continued on next page.)

Figure 132. `std::bind` implemented with C++20 lambdas.

```
template <std::size_t Idx>
struct Placeholder {};

template <typename T>
struct Bound {
  constexpr decltype(auto) operator()(auto &&...param) const {
    return t(std::forward<decltype(param)>(param)...);
  }

  T t;
};

template <typename T>
Bound(T) -> Bound<T>;

template <std::size_t Idx, typename T>
constexpr decltype(auto) get_param(const Placeholder<Idx> &,
                                   T &&t) {
  return std::get<Idx>(t);
}

template <typename Param, typename T>
constexpr decltype(auto) get_param(Param &&param, T &&) {
  return std::forward<Param>(param);
}

template <typename Param, typename T>
constexpr decltype(auto) get_param(const Bound<Param> &b,
                                   T &&t) {
  return std::apply(b, std::forward<T>(t));
}

constexpr decltype(auto) bind(auto &&callable, auto &&...param) {
  return Bound{
      [callable = std::forward<decltype(callable)>(callable),
       ... xs = std::forward<decltype(param)>(param)]
      (auto &&...values) {
        auto passed_params =
            std::forward_as_tuple(
               std::forward<decltype(values)>(values)...);
        return std::invoke(callable,
                           get_param(xs, passed_params)...);
      }
  };
}
```

I haven't looked at this code in 2 years, but here is a Compiler Explorer link for you to play with.

https://godbolt.org/z/hhde3P

Exercise: Understand the given example and critique it.

What should I have done differently with the above example? Can it be constrained with concepts? Does it need better names? What would you do differently?