C++ Best Practices

45ish Simple Rules with Specific Action Items for Better C++

Jason Turner

C++ Best Practices

45ish Simple Rules with Specific Action Items for Better C++

Jason Turner

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

This version was published on 2020-08-28

For my wife Jen, and her love of silly alpaca.

The cover picture was taken by my wife while visiting Red Fox Alpaca Ranch in Evergreen, Colorado.

Contents

1. Introduction 1
2. About Best Practices 2
3. Use the Tools: Automated Tests 4
4. Use the Tools: Continuous Builds 6
5. Use the Tools: Compiler Warnings 8
6. Use the Tools: Static Analysis 10
7. Use the Tools: Sanitizers 11
8. Slow Down 13
9. C++ Is Not Magic 14
10. C++ Is Not Object-Oriented 16
11. `const` everything that's not `constexpr` 17
12. `constexpr` Everything Known at Compile Time. 19
13. Prefer `auto` In Many Cases. 21
14. Prefer ranged-for Loop Syntax Over Old Loops 25
15. Use `auto` in ranged for loops 27
16. Prefer Algorithms Over Loops 29

17. Don't Be Afraid of Templates **31**

18. Don't Copy and Paste Code **32**

19. Follow the Rule of 0 . . . . . **34**

20. If You Must Do Manual Resource Management, Follow the Rule of 5 **36**

21. Don't Invoke Undefined Behavior **38**

22. Never Test for `this` To Be `nullptr`, It's UB **40**

23. Never Test for A `&` To Be `nullptr`, It's UB **42**

24. Avoid `default` In `switch` Statements **44**

25. Prefer Scoped `enum`s **47**

26. Prefer `if constexpr` over SFINAE **48**

27. Constrain Your Template Parameters With Concepts (C++20) **49**

28. De-template-ize Your Generic Code **50**

29. Use Lippincott Functions **52**

30. Be Afraid of Global State **55**

31. Make your interfaces hard to use wrong. **56**

32. Consider If Using the API Wrong Invokes Undefined Behavior **57**

33. Use `[[nodiscard]]` Liberally **59**

34. Use Stronger Types **61**

35. Don't return raw pointers **64**

36. Prefer stack over heap **65**

37. No More `new`! **67**

38. Know Your Containers **69**

39. Avoid `std::bind` and `std::function` **71**

40. Skip C++11 . . . . . **73**

41. Don't Use `initializer_list` For Non-Trivial Types. **75**

42. Use the Tools: Build Generators **77**

43. Use the Tools: Package Managers **79**

44. Improving Build Time **80**

45. Use the Tools: Multiple Compilers **82**

46. Fuzzing and Mutating **84**

47. Continue Your C++ Education **85**

48. Sponsors **87**

1. Introduction

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 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 learn something also. Every group is unique; every class has new questions.

But a lot of the questions are still the same ones over and over (to the point where I get to look like a mind reader, that bit's fun

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

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

2. About Best Practices

Best Practices, quite simply, are about

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

Why Best Practices?

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

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 really 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.

What's The Worst Than 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 vulnerability or attack vector.

Financial
: Serious flaws lead to large amounts of lost money, accelerating trades, market crash[1].

Aerospace
: Serious flaws lead to lost spacecraft or human life[2].

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

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 intended:

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

Links and References

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

[1] https://en.wikipedia.org/wiki/2010_flash_crash

[2] https://spectrum.ieee.org/aerospace/aviation/how-the-boeing-737-max-disaster-looks-to-a-software-developer

3. Use the Tools: Automated Tests

You need a single command to run tests.
If you don't have that your tests will not be run.

- catch2[1] - popular and well supported testing frame work from Phil Nash[2] and Martin Hořeňovský[3]
- doctest[4] - similar to catch2, but trimmed for compile-time performance
- googletest[5]
- boost::test[6] - testing framework, boost style.

ctest[7] is a test runner for CMake that can be used with any of the above frameworks. It is utilized via the `add_test`[8] feature of CMake.

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 existing code.

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[9]" which will give you the foundation you

[1] https://github.com/catchorg/Catch2
[2] https://twitter.com/phil_nash
[3] https://twitter.com/horenmar_ctu
[4] https://github.com/onqtam/doctest
[5] https://github.com/google/googletest
[6] https://www.boost.org/doc/libs/1_74_0/libs/test/doc/html/index.html
[7] https://cmake.org/cmake/help/latest/manual/ctest.1.html
[8] https://cmake.org/cmake/help/latest/command/add_test.html
[9] https://cppcast.com/clare-macrae/

need to start started with testing.
  - No: Develop a strategy for how to implement some minimal form of testing.

Resources

- CppCon 2018: Phil Nash “Modern C++ Testing with Catch2”[10]
- CppCon 2019: Clare Macrae “Quickly Testing Legacy C++ Code with Approval Tests”[11]
- C++ on Sea 2020: Clare Macrae “Quickly and Effectively Testing Legacy C++ Code with Approval Tests”[12]

[10] https://youtu.be/Ob5_XZrFQH0
[11] https://youtu.be/3GZHvcdq32s
[12] https://youtu.be/tXEuf_3VzRE

4. 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 Build separately
- Test all configuration options

- Test against newer compilers than you support or require

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?

5. Use the Tools: Compiler Warnings

You have many, 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 actually recommend using `/Wall` on MSVC or `-Weverything` on Clang, because many of these are diagnostic warnings. GCC does not provide an equivalent.

Strongly consider `-Wpedantic` (GCC/Clang) and `/permissive-` (MSVC). These command line options disable language extensions and get you closer to the C++ standard. This will help a lot with portability in the future.

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 a very nice 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.

Resources

- C++ Best Practices website curated list of warnings[1]
- GCC's full warning list[2]
- Clang's full warning list[3]
- MSVC's Compiler warnings that are off by default[4]
- C++ Weekly Ep 168 - Discovering Warnings You Should Be Using[5]

[1] https://github.com/lefticus/cppbestpractices/blob/master/02-Use_the_Tools_Available.md#compilers
[2] https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
[3] https://clang.llvm.org/docs/DiagnosticsReference.html
[4] https://docs.microsoft.com/en-us/cpp/preprocessor/compiler-warnings-that-are-off-by-default?view=vs-2019
[5] https://youtu.be/lOo8gTDMFkM

6. 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.

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

7. Use the Tools: Sanitizers

The sanitizers are runtime analysis tools for C++ 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[1] 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:

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

[1] https://twitter.com/johnregehr

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++ Starter Project[2]

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.

Resources

- AddressSanitizer (ASan) for Windows with MSVC[3]
- Sanitizers source and documentation on GitHub[4]
- Clang AddressSanitizer documentation[5]
- Clang UndefinedBehaviorSanitizer documentation[6]

[2] https://github.com/lefticus/cpp_starter_project
[3] https://devblogs.microsoft.com/cppblog/addresssanitizer-asan-for-windows-with-msvc/
[4] https://github.com/google/sanitizers
[5] https://clang.llvm.org/docs/AddressSanitizer.html
[6] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

8. Slow Down

For any given problem there are dozens of solutions in C++. If the solution you are implementing seems complex, try to slow down and find another solution.

This section is still in development

9. 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 can print whenever a special member function is called.

```
#include <cstdio>

struct S {
  S(){ puts("S()"); }
  S(const S &){ puts("S(const S &)"); }
  S(S &&){ puts("S(S &&)"); }
  S &operator=(const S &){
    puts("operator=(const S &)");
    return *this;
  }
  S &operator=(S &&){
    puts("operator=(S &&)");
    return *this;
  }
  ~S() { puts("~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.

Resources

- A quick start example with Compiler Explorer.[1]

[1] https://godbolt.org/z/3eGP56

10. C++ Is Not Object-Oriented

You must understand that C++ is a multi discipline programming language that supports:

- Procedural
- Functional
- Object-Oriented
- Generic
- Compile-Time

programming methodologies.

This section is incomplete

Resources

- Functional Programming in C++[1]
- C++ Weekly Ep 137: C++ Is Not an Object Oriented Language[2]

[1] https://www.manning.com/books/functional-programming-in-c-plus-plus?a_aid=FPinCXX&a_bid=441f12cc
[2] https://youtu.be/AUT201AXeJg

11. `const` everything that's not `constexpr`

Many people (like Kate Gregory and James McNellis) have said this many times. This does 2 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 as well.

Exercise: Look For `const` Opportunities

As you read your code 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`? (remember with RVO this will likely not add any overhead and may increase performance)

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

You probably don't want to make `class` members `const`, it can break important things, and sometimes silently.

Resources

- CppCon 2014: James McNellis & Kate Gregory "Modernizing Legacy C++ Code"[1]
- CppCon 2019: Jason Turner "C++ Code Smells"[2]
- The implication of const or reference member variables in C++[3]
- C++Now 2018: Ben Deane "Easy to Use, Hard to Misuse: Declarative Style in C++"[4] (Builds on techniques that make applying `const` easier.)

[1] https://youtu.be/LDxAgMe6D18
[2] https://youtu.be/f_tLQl0wLUM
[3] https://lesleylai.info/en/const-and-reference-member-variables/
[4] https://youtu.be/2ouxETt75R4

12. `constexpr` Everything Known at Compile Time.

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.

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

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

This really needs to be:

```
static constexpr std::array<int, 5> angles{-90,-45,0,45,90};
```

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

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 huge.

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: `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 is 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 they are also `constexpr`?

Resources

- C++Now 2017: Ben Deane & Jason Turner "`constexpr` ALL the things[1] (a bit out of date with modern `constexpr` techniques)
- C++ Weekly Ep 233: `constexpr` map vs `std::map`[2]
- Meeting C++ 2017: Jason Turner "Practical `constexpr`"[3]
- C++ Russia 2019: Hana Dusíková "A state of compile time regular expressions"[4]

[1] https://youtu.be/HMB9oXFobJc
[2] https://www.youtube.com/watch?v=INn3xa4pMfg
[3] https://youtu.be/xtf9qkDTrZE
[4] https://youtu.be/r_ZASJFQGQI

13. 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."

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

or, if you prefer:

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

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

Example:

```
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.

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

Exercise: Become familiar with `auto` deduction.

```
const int *get();

int main() {
  // what is the type of val?
  const auto val = get();
}
```

```
const int &get();

int main() {
  // what is the type of val?
  const auto val = get();
}
```

```
const int *get();

int main() {
  // what is the type of val?
  const auto *val = get();
}
```

```
const int &get();

int main() {
  // what is the type of val?
  const auto &val = get();
}
```

```
const int *get();

int main() {
  // what is the type of val?
  const auto &val = get();
}
```

```
const int &get();

int main() {
  // what is the type of val?
  const auto &&val = get();
}
```

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].

Resources

- clang-tidy modernize-use-auto[1]

[1] https://clang.llvm.org/extra/clang-tidy/checks/modernize-use-auto.html

- Almost Always Auto[2]

[2] https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

14. Prefer ranged-for Loop Syntax Over Old Loops

We'll illustrate this point with a series of examples.

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

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

```
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

Resources

- clang-tidy modernize-loop-convert[1]

[1] https://clang.llvm.org/extra/clang-tidy/checks/modernize-loop-convert.html

15. Use `auto` in ranged for loops

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

Accidental Conversions

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

Accidental Slicing

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

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 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?

```
std::map<std::string, int> get_map();

using element_type = std::pair<std::string, int>;
for (const element_type & : get_map())
{
}
```

Exercise: Enable Ranged Loop Related Warnings

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

16. Prefer Algorithms Over Loops

Algorithms communicate meaning and helps us apply the "`const` All The Things" rule. In C++20 we get ranges, which make algorithms easier to use.
It's possible, taking a functional approach and using algorithms, that we can write C++ that reads like a sentence.

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

Exercise: Study Existing Loops

Next time you are reading through a loop in your code base cross reference it with https://en.cppreference.com/w/cpp/algorithm and try to find an algorithm that applies instead.

Resources

- GoingNative 2013: Sean Parent "C++ Seasoning"[1]
- CppCon 2018: Jonathan Boccara "105 Algorithms in Less Than an Hour"[2]
- C++ Now 2019: Connor Hoekstra "Algorithm Intuition"[3]
- Code::Dive 2018: Connor Hoekstra "Better Algorithm Intuition"[4]
- C++ Weekly Ep 187 "C++20's `constexpr` Algorithms"[5]

[1] https://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning
[2] https://youtu.be/2olsGf6JIkU
[3] https://youtu.be/48gV1SNm3WA
[4] https://youtu.be/0z-cv3gartw
[5] https://youtu.be/9YWzXSr2onY

- C++ Weekly Ep 105 “Learning “Modern” C++ 5: Looping And Algorithms[6]

[6] https://youtu.be/A0-x-Djey-Q

17. Don’t Be Afraid of Templates

This is the ultimate DRY principle in C++. Templates can be complex, daunting and Turing complete, but they don’t have to be. 15 years ago, it seemed the prevailing attitude is “templates aren’t for normal people.”

Fortunately, this is less true today. And we have more tools today, concepts, generic lambdas, etc.

This section is incomplete

18. 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 with 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 to be 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 types of data 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 code base.

For this exercise, download the PMD CPD tool[1]
and run it on your code base.

[1] https://pmd.github.io/latest/pmd_userdocs_cpd.html

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 copy and pasted? What happens if you find a bug in one version? Will you
be sure to find all of the versions that also need to be updated?

Resources

- Copy-Paste Programming[2]
- The Last Line Effect[3]
- i will not copy-paste code[4]

[2] https://www.viva64.com/en/t/0068/
[3] https://www.viva64.com/en/b/0260/
[4] https://twitter.com/bjorn_fahller/status/1072432257799987200

19. 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

Keep in mind that if you need a destructor because you are doing resource management or defining a virtual base class, 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).

```
struct S {
  // a bunch of other things
  ~S() {}
};
```

or worse:

```hpp
// file.hpp
struct S {
  ~S();
}
```

```cpp
// file.cpp
S::~S() {}
```

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

If they are in types that are 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.

Resources

- C++ Reference: The rule of three/five/zero[1]
- C++ Weekly Ep 154: “One Simple Trick for Reducing Code Bloat”[2]
- CppCon 2019: Jason Turner “Great C++ is_trivial”[3]

[1] https://en.cppreference.com/w/cpp/language/rule_of_three
[2] https://youtu.be/D8eCPl2zit4
[3] https://youtu.be/ZxWjii99yao

20. 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 was originally known as the rule of 3 and is known as the rule of 5 after C++11.

```
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!

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

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

Resources

- C++ Reference: The rule of three/five/zero[1]

[1] https://en.cppreference.com/w/cpp/language/rule_of_three

21. Don’t Invoke Undefined Behavior

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

http://eel.is/c++draft/intro.compliance#intro.abstract-5

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.

See also: https://youtube.com/watch?v=XEXpwis_deQ

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.

Resources

- C++Now 2018: John Regehr “Closing Keynote: Undefined Behavior and Compiler Optimizations”[1]

[1] https://youtu.be/AeEwxtEOgH0

22. Never Test for `this` To Be `nullptr`, It's UB

```
int Class::member() {
  if (this == nullptr) {
    // removed by the compiler, it would be UB
    // if this were ever null
    return 42;
  } else {
    return 0;
  }
}
```

Technically it isn't the check that is Undefined Behavior (UB). But it's impossible for the check to ever 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.

Do You Check For `this` To Be `nullptr`?

This can hide as a check for `NULL` or a check against `0`. This is likely to only exist in very old code bases. Make sure you have you warnings enabled, then go look.

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

Resources

- Porting to GCC-6 Optimizations remove null pointer checks for `this`[1]

[1] https://www.gnu.org/software/gcc/gcc-6/porting_to.html#this-cannot-be-null

23. Never Test for A & To Be `nullptr`, It's UB

```
int get_value(int &thing) {
  if (&thing == nullptr) {
    // removed by compiler
    return 42;
  } else {
    return thing;
  }
}
```

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 can go wrong.

Search through your code for statements that check the address of an object 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.

Resources

- -Wtautological-undefined-compare[1]

[1] https://clang.llvm.org/docs/DiagnosticsReference.html#wtautological-undefined-compare

24. Avoid `default` In `switch` Statements

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

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

std::string_view get_name(Values value) {
  switch (value) {
  case val1: return "val1";
  case 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:

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

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

But this introduces a new problem

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

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

Instead, prefer code like this:

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

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

  return "unknown";
}
```

Exercise: Look For `default:`

I'm not currently aware of any tools that would warn on the use of `default:`, so you'll have to search for it manually.

What do you find in your code base?

Resources

- CppCon 2018: Jason Turner "Applied Best Practices"[1]
- `-Wswitch-enum`[2]
- `-Wswitch`[3]

[1] https://youtu.be/DHOlsEd0eDE
[2] https://clang.llvm.org/docs/DiagnosticsReference.html#wswitch-enum
[3] https://clang.llvm.org/docs/DiagnosticsReference.html#wswitch

25. Prefer Scoped `enum`s

```
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;
```

These above cannot get mixed up without much effort and their identifiers are now scoped
See also talks from Victor Ciura about Clang Power Tools, where he mentions several anecdotes about this.

> `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 come up with a well reasoned answer as to why.

Exercise: clang-tidy modernize

Clang-tidy's modernizer can add `class` to your `enum` declarations.

26. Prefer `if constexpr` over SFINAE

SFINAE is kind of write-only code. `if constexpr` doesn't have all the same flexibility, but use it when you can. See Practical C++17

This section is incomplete.

Resources

- C++ Weekly Special Edition: Using C++17's constexpr if[1]
- C++ Weekly Ep 122: `constexpr` with `optional` and `variant`[2]
- CppCon 2017: Jason Turner "Practical C++17"[3]
- C++17 In Tony Tables: `constexpr if`[4]

[1] https://youtu.be/_Ny6Qbm_uMI
[2] https://youtu.be/2eCV_udkP_o
[3] https://youtu.be/nnY4e4faNp0
[4] https://github.com/tvaneerd/cpp17_in_TTs/blob/master/if_constexpr.md

27. Constrain Your Template Parameters With Concepts (C++20)

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

This section is incomplete.

Resources

- C++ Weekly Ep 194: From SFINAE To Concepts With C++20[1]
- C++ Weekly Ep 196: What is `requires requires`[2]

[1] https://youtu.be/dR64GQb4AGo
[2] https://youtu.be/tc0hVIOJk_U

28. 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.

This 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[1] (article formatting got broken at some point, sorry)

```
template<typename T>
void do_things()
{
  // this lambda must be generated for each
  // template instantiation
  auto lambda = [](){ /* some lambda */ };
  auto value = lambda();
}
```

Compared to:

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

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

[1] https://articles.emptycrate.com/2008/05/06/nobody_understands_c_part_5_template_code_bloat.html

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.

Bloaty McBloatface And `-ftime-trace`

We're getting more and more tools available that know how 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.

Resources

- Templight[2]
- C++ Weekly Ep 89: "Overusing Lambdas"[3]
- C++ Weekly Christmas Class 2019 - Chapter 3[4] (This is the first episode of chapter 3, and it introduces the question of how and why two different options differ[5]. The next several episodes in that play list give some background and the start of chapter 4 gives the answer. It is very much related to template bloat questions.)

[2] https://github.com/mikael-s-persson/templight
[3] https://youtu.be/OmKMNQFx_8Y
[4] https://www.youtube.com/watch?v=VEqOOKU8RjQ&list=PLs3KjaCtOwSY_Awyliwm-fRjEOa-SRbs-&index=16
[5] https://godbolt.org/z/b4znvK

29. Use Lippincott Functions

Same arguments as de-templatizing 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:

```
void use_thing() {
  try {
    do_thing();
  } catch (const std::runtime_error &) {
    // handle it
  } catch (const std::exception &) {
    // handle it
  }
}

void use_other_thing() {
  try {
    do_other_thing();
  } catch (const std::runtime_error &) {
    // handle it
  } catch (const std::exception &) {
    // handle it
  }
}
```

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

```
void handle_exception() {
  try {
    throw; // re-throw exception already in flight
  } catch (const std::runtime_error &) {
  } catch (const std::exception &) {  }
}

void use_thing() {
  try {
    do_thing();
  } catch (...) {
    handle_exception();
  }
}

void use_other_thing() {
  try {
    do_other_thing();
  } catch (...) {
    handle_exception();
  }
}
```

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?

Resources

- C++ Secrets: Using a Lippincott Function for Centralized Exception Handling[1]
- C++ Weekly Ep 91: Using Lippincott Functions[2]

[1] https://cppsecrets.blogspot.com/2013/12/using-lippincott-function-for.html
[2] https://youtu.be/-amJL3AyADI

30. Be Afraid of Global State

Global state is hard.

Any non-`const` `static` value, or `std::shared_ptr<>` is the potential for 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? Is it possible for there to be accidentally shared state between modules of your system via the logger?

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)?

31. 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 then your users have to work harder to use it wrong. But this is still C++, they will still find a way.

This 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.

32. 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 you 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 use the API incorrectly?

Exercise: Investigate Checked Types

The C++ Guideline Support Library (GSL) has a `not_null` pointer type that guarantees with zero cost abstractions that the pointer passed is never `nullptr`. Would that work for your API's 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.

Resources

- boost::safe_numerics[1]

[1] https://github.com/boostorg/safe_numerics

33. Use `[[nodiscard]]` Liberally

`[[nodiscard]]` (C++17) is a C++ attribute that tells the compiler to warn if a return value is ignored. It can be used on functions:

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

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

And 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.

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

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

Read the Reddit discussion "An Argument Pro Liberal Use Of nodiscard"[1].
Read the above Reddit discussion. Consider your own 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()

Resources

- "An Argument Pro Liberal Use Of nodiscard"[2]
- C++ Weekly Ep 30: C++17's [[nodiscard]] Attribute[3]
- C++ Weekly Ep 199: C++20's [[nodiscard]] Constructors And Their Uses[4]

[1] https://www.reddit.com/r/cpp/comments/9us7f3/an_argument_pro_liberal_use_of_nodiscard/
[2] https://www.reddit.com/r/cpp/comments/9us7f3/an_argument_pro_liberal_use_of_nodiscard/
[3] https://youtu.be/l_5PF3GQLKc
[4] https://youtu.be/E_ROB_xUQQQ

34. Use Stronger Types

Consider the api for POSIX `socket`

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

The parameters (in some order) represent:

- type
- protocol
- domain

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

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

This could be easily 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?

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

In many cases it only takes a little effort to make more strongly typed API's

```
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 composible statements with operator overloads such ask:

```
// 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 actually *increase* performance in some cases (C++ Weekly Ep 119, Negative Cost Structs) https://youtu.be/FwsO12x8nyM

Identify The Problematic API's In Your Existing Code

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

Research Strong Typedef Libraries For C++

There are existing libraries that simplify some of the boiler plate code for you when trying to make a strongly typed `int`. Jonathan Muller and Bjorn Fahller have each written one, and there are others available.

Consider `=delete`ing Problematic Conversions

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

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

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

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

Resources

- C++ Weekly Ep 107: "The Power of `=delete`"[1]
- Adi Shavit and Björn Fahller "The Curiously Recurring Pattern of Coupled Types"[2]
- Research "Affine space types"
- C++Now 2017: Jonathan Müller "Type-safe Programming"[3]

[1] https://youtu.be/aAvjUU0m6AU
[2] https://youtu.be/msi4WNQZyWs
[3] https://youtu.be/iihlo9A2Ezw

35. 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.

```
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 it 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: You know the drill

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?

36. Prefer stack over heap

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

In the most simple terms:

```
// OK Idea, uses stack can optimized
std::string make_string() { return "Hello World"; }
```

```
// Bad Idea, uses the heap
std::unique_ptr<std::string> make_string() {
  return std::make_unique<std::string>("Hello World");
}
```

```
// OK Idea
void use_string() {
  // This string lives on the stack
  std::string value("Hello World");
}
```

```
// Really bad idea, uses the heap and leaks memory
void use_string() {
  // The string lives on the heap
  std::string *value = new std::string("Hello World");
}
```

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."

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 Profilers

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.

Resources

- Code::Dive 2018: Björn Fahller "What Do You Mean By Cache Friendly?"[1]

[1] https://youtu.be/Fzbotzi1gYs

37. 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<>()`[1] (C++14) in the rare cases that you need the heap.

In the very rare cases you need shared ownership, use `std::make_shared<>()`[2] (C++11).

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.

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.

[1] https://en.cppreference.com/w/cpp/memory/unique_ptr/make_unique
[2] https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared

Resources

- clang-tidy modernize-make-shared[3]
- clang-tidy modernize-make-unique[4]

[3] https://clang.llvm.org/extra/clang-tidy/checks/modernize-make-shared.html
[4] https://clang.llvm.org/extra/clang-tidy/checks/modernize-make-unique.html

38. 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 size and location of the data.

`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.

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

can become

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

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

Resources

- Bjarne Stroustrup “Are lists evil?”[1]

[1] https://www.stroustrup.com/bs_faq.html#list

39. Avoid `std::bind` and `std::function`

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

Exercise: Compare the possibilities

Take these options in Compiler Explorer and compare the compile times and what the resulting assembly looks like.

```
#include <functional>

template<typename Func>
std::function<int (int)> bind_3(Func func)
{
  return std::bind(func, std::placeholders::_1, 3);
}

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

```cpp
#include <functional>

template<typename Func>
auto bind_3(Func func)
{
  return std::bind(func, std::placeholders::_1, 3);
}

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

```cpp
#include <functional>

template<typename Func>
auto bind_3(Func func)
{
  return [func](const int value){ return func(value, 3); };
}

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

Resources

- CppCon 2015: Stephan T. Lavavej “<functional>: What’s New, And Proper Usage”[1]
- C++ Weekly Ep 16: “Avoiding `std::bind`”[2]

[1] https://youtu.be/zt7ThwVfap0
[2] https://youtu.be/ZlHi8txU4aQ

40. Skip C++11

If you're currently looking to finally move to "modern" C++ please skip C++11. C++14 fixes several holes in C++11.

Language Features

- C++11's version of `constexpr` implies `const` for member functions, this is changed in C++14
- C++11 is missing `auto` return type deduction for regular functions (lambdas have it)
- C++11 does not have `auto` or variadic lambda parameters
- C++14 adds `[[deprecated]]` attribute
- C++14 adds `'` digit separator, example: `1'000'000`
- `constexpr` functions can be more than just a single return statement in C++14

Library Features

- `std::make_unique` was added in C++14, which enables the "no raw `new`" standard
- C++11 doesn't have `std::exchange`
- C++14 adds some `constexpr` support for `std::array`
- `cbegin`, `cend`, `crbegin` and `crend` free functions added for consistency with `begin` and `end` free functions and member functions added to standard containers in C++11.

Exercise: Can You Use C++14 Today?

Even if your project is currently stuck on a C++11 compiler, it is highly likely that you can use C++14 features if you enable `-std=c++1y` or `-std=c++14` mode.

Compare the C++14 language feature chart[1] from cppreference.com with the compiler you currently require for your project. How many features could you be taking advantage of today?

As of GCC 5, all of C++14 is supported, but as early as 4.9 provided many C++14 features.

Exercise: Can You Go Beyond C++14?

Ask if it's possible to upgrade your current compiler requirements. With very rare exceptions each new compiler version brings

- Better performance
- Fewer bugs
- Better warnings
- Better standards conformance

Resources

- C++ Weekly Ep 173: The Important Parts of C++98 in 13 Minutes[2]
- C++ Weekly Ep 176: The Important Parts of C++11 in 12 Minutes[3]
- C++ Weekly Ep 178: The Important Parts of C++14 in 9 Minutes[4]
- C++ Weekly Ep 190: The Important Parts of C++17 in 10 Minutes[5]

[1] https://en.cppreference.com/w/Template:cpp/compiler_support/14
[2] https://youtu.be/78Y_LRZPVRg
[3] https://youtu.be/D5n6xMUKU3A
[4] https://youtu.be/mXxNvaEdNHI
[5] https://youtu.be/QpFjOlzg1r4

41. Don't Use `initializer_list` For Non-Trivial Types.

This is a somewhat 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.

Watch "Initializer Lists Are Broken, let's Fix Them"

Exercise: Understand the overhead `initializer_list` can bring

Use Andreas Fertig's awesome http://cppinsights.io to understand exactly what these two examples do

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

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

```
#include <vector>
#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

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

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

Resources

- C++Now 2018: Jason Turner "Initializer Lists Are Broken, Let's Fix Them"[1] (deep dive into the issues around these topics)
- C++ Insights[2]

[1] https://youtu.be/sSlmmZMFsXQ
[2] https://cppinsights.io/

42. Use the Tools: Build Generators

- CMake[1]
- Meson[2]
- Bazel[3]
- Others[4]

Raw make files or Visual Studio project files make each of the things listed above too difficult 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 platform in use.

Exercise: Investigate Your Build System

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

[1] https://cmake.org
[2] https://mesonbuild.com/
[3] https://bazel.build/
[4] https://github.com/lefticus/cppbestpractices/blob/master/02-Use_the_Tools_Available.md

See if there are current best practices you need to apply. Are there formatting or tidy-like 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.

Resources

- [Professional CMake:

A Practical Guide](https://crascit.com/professional-cmake/)

- C++Now 2017: Daniel Pfeiffer “Effective CMake”[5]
- C++ Weekly Ep 218 - The Ultimate CMake / C++ Quick Start[6]
- BazelCon 2019[7]
- CppCon 2018: Jussi Pakkanen “Compiling Multi-Million Line C++ Code Bases Effortlessly with the Meson Build System”[8]
- cmake-tidy[9]

[5] https://youtu.be/bsXLMQ6WgIk
[6] https://youtu.be/YbgH7yat-Jo
[7] https://www.youtube.com/playlist?list=PLxNYxgaZ8Rsf-7g43Z8LyXct9ax6egdSj
[8] https://youtu.be/SCZLnopmYBM
[9] https://github.com/MaciejPatro/cmake-tidy

43. 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[1]
- Conan[2]

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

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?

[1] https://github.com/Microsoft/vcpkg
[2] https://conan.io/

44. Improving Build Time

A few practical considerations for making build time less painful

- Use an IDE. IDE's do real-time analysis of the code and you spend less time waiting for builds to see if the code will compile.
- 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 being spent

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.

Resources

- A guide to unity builds[1]
- Unity builds with Meson[2]
- Unity builds with CMake[3]

[1] https://onqtam.com/programming/2018-07-07-unity-builds/
[2] https://mesonbuild.com/Unity-builds.html
[3] https://cmake.org/cmake/help/latest/prop_tgt/UNITY_BUILD.html

- PCH with Meson[4]
- PCH with CMake[5]
- ccache[6]
- CMake Compiler Launcher[7]
- Clang Build Analyzer[8]
- Getting started with C++ Build Insights[9]

[4] https://mesonbuild.com/Precompiled-headers.html
[5] https://cmake.org/cmake/help/latest/command/target_precompile_headers.html
[6] https://ccache.dev/
[7] https://cmake.org/cmake/help/latest/prop_tgt/LANG_COMPILER_LAUNCHER.html?highlight=ccache
[8] https://github.com/aras-p/ClangBuildAnalyzer
[9] https://docs.microsoft.com/en-us/cpp/build-insights/get-started-with-cpp-build-insights?view=vs-2019

45. Use the Tools: Multiple Compilers

Support *at least* 2 compilers on your platform. Each compiler does different analysis 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 easily switch between gcc and clang.

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

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

Ubuntu / Debian

- GCC - Toolchain PPA https://launchpad.net/~ubuntu-toolchain-r/+archive/ubuntu/ppa
- Clang - apt packages https://apt.llvm.org/

Windows

- GCC MinGW - http://mingw.org/
- Clang official downloads - https://releases.llvm.org/download.html

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 portions of the project compiling on another operating system and tool chain will teach you a lot about the nature of your code.

Resources

- C++Now 2015: Jason Turner "Thinking Portable: How and Why to make your C++ cross platform"[1]

[1] https://youtu.be/cb3WIL96N-o

46. Fuzzing and Mutating

This section is incomplete

Resources

- C++Now 2018: Marshall Clow “Making Your Library More Reliable with Fuzzing”[1]
- C++ Weekly Ep 85: Fuzz Testing[2]
- CppCast: Alex Denisov “Mutation Testing With Mull”[3]
- NDC TechTown 2019: Seph De Busser “Testing The Tests: Mutation Testing for C++”[4]

[1] https://youtu.be/LlLJRHToyUk
[2] https://youtu.be/gO0KBoqkOoU
[3] https://cppcast.com/alex-denisov/
[4] https://youtu.be/M-5_M8qZXaE

47. Continue Your C++ Education

Many resources are available to you to continue you C++ education.

This section is in progress!

Conferences And Local User Groups

There is almost certainly one near you. It's a great way to network and learn new things. https://isocpp.org/wiki/faq/conferences-worldwide https://meetingcpp.com/usergr

C++ Weekly

Over 233 weeks straight now!!

cppreference.com

The website is awesome, 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! https://en.cppreference.com/w/Cppreferen

Hire a Trainer to Come Onsite for Your Company

This 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).

YouTube

- Andreas Fertig's Channel[1]

[1] https://www.youtube.com/channel/UCxJfIsPGHFS3_nRDv1u-Q8g

48. Sponsors

Thank you to all of my Book Supporter patrons!

Current

Adam Albright,
Adam P Shield,
Andrei Sebastian Cîmpean,
Anton Smyk,
Björn Fahller,
Corentin Gay,
David C Black,
Dennis Börm,
Fedor Alekseev,
Florian Sommer,
Gwendolyn Hunt,
Jack Glass,
Jaewon Jung,
Jeff Bakst,
Kacper Kołodziej,
Lars Ove Larsen,
Magnus Westin,
Martin Hammerchmidt,
Matt Godbolt,
Matthew Guidry,
Michael Pearce,
Olafur Waage,
Panos Gourgaris,
Sebastian Raaphorst,
Tim Butler,

Tobias Dieterich,
Yacob Cohen-Arazi

Former

Alejandro Lucena,
Emyr Williams,
Natalya Kochanova,
Reiner Eiteljoerge