Google C++ Style Guide

Author: Benjy Weinberger & Craig Silverstein & Gregory Eitzmann & Mark Mentovai & Tashana Landray

Institute: Google

Date: January 5, 2023

Version: commit 78c5bdd8514b7639dffea8d74d96a03ab8b47b38

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

The Philosophy of Google's C++ Style Guide[1]

Titus Winters (titus@google.com)

Introduction

❏ *About Us*
❏ *The underpinnings of Google's C++ Style Guide*
❏ *The Contentious Rules*
❏ *Recap*

0.1 About Us

- 4K-ish C++ engineers
- Shared codebase
- Strong testing culture
- Good indexer (Kythe)
- Wild variance in C++ background
- Good code review policies
- We expect we'll be around for a while, and should plan accordingly

Most projects check into the same codebase. Most engineers have read access to most code. Most projects use the same infrastructure (libraries, build system, etc). Code is going to live a long time, and be read many times. We choose explicitly to optimize for the reader, not the writer.

0.2 The underpinnings of Google's C++ Style Guide

#1 Optimize for the Reader, not the Writer

- We're much more concerned with the experience of code readers.

#2 Rules Should Pull Their Weight

- We aren't going to list every single thing you shouldn't do. Rules for dumb stuff should be handled at a higher level ("Don't be clever").

#3 Value the Standard, but don't Idolize

- Tracking the standard is valuable (cppreference.com, stackoverflow,etc). Not everything in the standard is equally good.

#4 Be Consistent

Pros:

- Consistency allows easier expert chunking.
- Consistency allows tooling.
- Consistency allows us to stop arguing about stuff that doesn't matter.

[1]This is just the main part of Titus Winters's great presentation at CppCon2014. You can watch the full talk on Youbute : The Philosophy of Google's C++ Style Guide

Practices:

- Include guard naming / formatting
- Parameter ordering (input, then output, unless consistency with other things matters)
- Namespaces (naming)
- Declaration order
- 0 and NULL vs. nullptr
- Naming
- Formatting
- ~~Don't use streams~~

#5 If something unusual is happening, leave explicit evidence for the reader

- Old Example: "No non-const references" leads to "The extra '&' means it could be mutated".

```
int main(int argc, char** argv) {
  ParseCommandLineFlags(&argc, &argv, true);
}
```

- New Example: The design of `std::unique_ptr` makes it fit perfectly into a codebase with pre-C++-style pointers.

```
// Taking ownership: new from old.
std::unique_ptr<Foo> my_foo(NewFoo());

// or old from new
Foo* my_foo = NewFoo().release();

// or new from new
std::unique_ptr<Foo> my_foo = NewFoo();

// Yielding ownership (new to old)
TakeFoo(my_foo.release());

// or new to new
TakeFoo(std::move(my_foo));

// or old to new
TakeFoo(std::make_unique<Foo>(my_foo));
```

- Rules that help leave a trace for the reader include:
  - `override` or `final`
  - Interface classes - Name them with the "Interface" suffix
  - Function overloading - If it matters which overload is being called, make it obvious by inspection
  - No Exceptions - Error handling is explicit

#6 Avoid constructs that are dangerous or surprising

- Waivers here are probably rare, and would require a strong argument, and probably some comments to mitigate the chance of copy and paste re-using those patterns unsafely. Examples include:
  - Static and global variables of complex type (danger at shutdown)
  - Use override or final (avoid surprise)
  - Exceptions (dangerous)
- Most code should avoid the tricky stuff. Waivers may be granted if justified.
  - Avoid macros (non-obvious, complicated)
  - Template metaprogramming (complicated, often non-obvious)
  - Non-public inheritance (surprising)
  - Multiple implementation inheritance (hard to maintain)

#7 Avoid polluting the global namespace

Waivers here are unlikely except in very extreme cases.

- Put your stuff in a namespace
- Don't "using" into the global namespace from a header
- Inside a .cc: We don't care much
  - Still a distinction between using vs. using namespace

#8 Concede to optimization and practicalities when necessary

Sometimes we make rulings just to state that an optimization may be healthy and necessary. (These are usually explicit "is allowed".)

- Allow forward declarations ("optimizing" build times)*
- Inline functions
- Prefer pre-increment (++i)

0.3 The Contentious Rules

There are two (very) contentious rules:

- No non-const references as function arguments
- No use of exceptions

0.3.1 non-const references

Three rules apply:

- Consistency
- Leave a trace/explicitness
- Dangerous/surprising constructs: reference lifetime issues

0.3.2 no exceptions

Some rules apply:

- Value the standard, but don't idolize
- Consistency

  - This stems from old compiler bugs, but once that happened ...
- Leave a trace
- Dangerous/surprising constructs
- Avoid hard to maintain constructs
  - Consider cases where exception types are changed
- Concede to optimization
  - On average, code locality matters.

0.4 Recap

- Have a style guide. Tailor it to your situation.
- Use your guide to encourage "good" and discourage "bad".
- Re-evaluate.

Contents

The Philosophy of Google's C++ Style Guide **i**

0.1 About Us i

0.2 The underpinnings of Google's C++ Style Guide i

0.3 The Contentious Rules iii

0.4 Recap iv

1 Background **1**

1.1 Goals of the Style Guide 1

2 C++ Version **5**

3 Header Files **7**

3.1 Self-contained Headers 7

3.2 The `# define` Guard 7

3.3 Include What You Use 8

3.4 Forward Declarations 8

3.5 Inline Functions 9

3.6 Names and Order of Includes 10

4 Scoping **13**

4.1 Namespaces 13

4.2 Internal Linkage 16

4.3 Nonmember, Static Member, and Global Functions 16

4.4 Local Variables 17

4.5 Static and Global Variables 18

4.6 `thread_local` Variables 21

5 Classes **25**

5.1 Doing Work in Constructors 25

5.2 Implicit Conversions 26

5.3 Copyable and Movable Types 27

5.4 Structs vs. Classes 30

5.5 Structs vs. Pairs and Tuples 30

5.6 Inheritance 30

5.7 Operator Overloading 31

5.8 Access Control 33

5.9 Declaration Order 33

6 Functions **35**

6.1 Inputs and Outputs 35

6.2 Write Short Functions 35

6.3 Function Overloading . . . 36
6.4 Default Arguments . . . 37
6.5 Trailing Return Type Syntax . . . 37

7 Google-Specific Magic 39
7.1 Ownership and Smart Pointers . . . 39
7.2 cpplint . . . 40

8 Other C++ Features 41
8.1 Rvalue References . . . 41
8.2 Friends . . . 42
8.3 Exceptions . . . 42
8.4 `noexcept` . . . 44
8.5 Run-Time Type Information (RTTI) . . . 45
8.6 Casting . . . 46
8.7 Streams . . . 47
8.8 Preincrement and Predecrement . . . 48
8.9 Use of const . . . 49
8.10 Use of constexpr . . . 50
8.11 Integer Types . . . 50
8.12 64-bit Portability . . . 52
8.13 Preprocessor Macros . . . 52
8.14 0 and nullptr/NULL . . . 53
8.15 sizeof . . . 53
8.16 Type Deduction (including `auto`) . . . 54
8.17 Class Template Argument Deduction . . . 59
8.18 Designated Initializers . . . 60
8.19 Lambda Expressions . . . 61
8.20 Template Metaprogramming . . . 64
8.21 Boost . . . 65
8.22 Other C++ Features . . . 66
8.23 Nonstandard Extensions . . . 66
8.24 Aliases . . . 67

9 Inclusive Language 69

10 Naming 71
10.1 General Naming Rules . . . 71
10.2 File Names . . . 72
10.3 Type Names . . . 73
10.4 Variable Names . . . 73

11 Comments 77
11.1 Comment Style 77
11.2 File Comments 77
11.3 Class Comments 78
11.4 Function Comments 78
11.5 Variable Comments 80
11.6 Implementation Comments 80
11.7 Function Argument Comments 80
11.8 Don’ts 81
11.9 Punctuation, Spelling, and Grammar 82
11.10TODO Comments 82

12 Formatting 83
12.1 Line Length 83
12.2 Non-ASCII Characters 84
12.3 Spaces vs. Tabs 84
12.4 Function Declarations and Definitions 85
12.5 Lambda Expressions 86
12.6 Floating-point Literals 87
12.7 Function Calls 87
12.8 Braced Initializer List Format 89
12.9 Conditionals 90
12.10Loops and Switch Statements 91
12.11Pointer and Reference Expressions 93
12.12Boolean Expressions 94
12.13Return Values 94
12.14Variable and Array Initialization 95
12.15Preprocessor Directives 95
12.16Class Format 96
12.17Constructor Initializer Lists 97
12.18Namespace Formatting 97
12.19Horizontal Whitespace 98

13 Exceptions to the Rules 103
13.1 Existing Non-conformant Code 103
13.2 Windows Code 103

A Singular “They” 105
A.1 Forms of the singular “they” 105
A.2 Alternatives to the generic singular “they” 105

B Misc Use After Move 107
B.1 Unsequenced moves, uses, and reinitializations 108
B.2 Move 108

B.3 Use . . . 109
B.4 Reinitialization . . . 109

C absl/memory/memory.h 111

D Hyrum's Law 127
D.1 Who's Hyrum? . . . 128

E Tip of the Week #119: Using-declarations and namespace aliases 129
E.1 Summary . . . 129
E.2 Background . . . 130
E.3 The Problem . . . 130
E.4 Unnamed Namespaces . . . 132
E.5 Non-aliased names . . . 133
E.6 Acknowledgments . . . 133

F Tip of the Week #42: Prefer Factory Functions to Initializer Methodsr 135

G Tip of the Week #55:Name Counting and `unique_ptr` 137

H Tip of the Week #65: Putting Things in their Place 139
H.1 The Old Way and the New Way . . . 139
H.2 Using Emplace Methods for Move-Only Operations . . . 140
H.3 Conclusion . . . 141

I Tip of the Week #77: Temporaries, Moves, and Copies 143
I.1 Can You Count to 2? . . . 143
I.2 Name Counting, in Brief . . . 143
I.3 Two Names: It's a Copy . . . 143
I.4 One Name: It's a Move . . . 143
I.5 Zero Names: It's a Temporary . . . 144
I.6 Beware: Zombies . . . 144
I.7 Wait, `std::move` Doesn't Move? . . . 145
I.8 Aaaagh! It's All Complicated! Why!?! . . . 145

J Tip of the Week #86: Enumerating with Class 147
J.1 Unscoped Enumerations . . . 147
J.2 Scoped Enumerations . . . 147
J.3 Underlying Enumeration Types . . . 148
J.4 Conclusion . . . 148

K Tip of the Week #109: Meaningful `const` in Function Declarations 151
K.1 Meaningful `const` in Function Declarations . . . 151
K.2 Rules of Thumb . . . 152

L Tip of the Week #112: `emplace` vs. `push_back` 153

M Tip of the Week #117: Copy Elision and Pass-by-value **155**
M.1 When to Use Copy Elision 156

N Tip of the Week #123: `absl::optional` and `std::unique_ptr` **159**
N.1 How to Store Values 159
N.2 Conclusion 160

O Tip of the Week #126: `std::make_unique` is the new `new` **163**
O.1 Why Avoid `new` ? 163
O.2 How Should We Choose Which to Use? 164
O.3 Summary 164

P Tip of the Week #130: Namespace Naming **165**
P.1 Name Lookup 165
P.2 The Problem 166
P.3 Two Consistent Options, Three Approaches 166
P.4 But It Keeps Things Organized! 167
P.5 Best Practices 167

Q Tip of the Week #140: Constants: Safe Idioms **169**
Q.1 Constants in Header Files 170
Q.2 Other Common Mistakes 174
Q.3 Initialization Cheat Sheet 177
Q.4 Further Reading and Collected Links 177
Q.5 Conclusion 177

R Tip of the Week #148: Overload Sets **179**
R.1 What is an Overload Set? 179
R.2 String-ish Parameters 179
R.3 StrCat 180
R.4 Parameter Sinks 180
R.5 Overloaded Accessors 181
R.6 Copy vs. Move 182
R.7 Conclusion 182

Chapter 1
Background

Introduction

❑ *Goals of the Style Guide*

C++ is one of the main development languages used by many of Google's open-source projects. As every C++ programmer knows, the language has many powerful features, but this power brings with it complexity, which in turn can make code more bug-prone and harder to read and maintain.

The goal of this guide is to manage this complexity by describing in detail the dos and don'ts of writing C++ code . These rules exist to keep the code base manageable while still allowing coders to use C++ language features productively.

Style, also known as readability, is what we call the conventions that govern our C++ code. The term Style is a bit of a misnomer, since these conventions cover far more than just source file formatting.

Most open-source projects developed by Google conform to the requirements in this guide.

Note that this guide is not a C++ tutorial: we assume that the reader is familiar with the language.

1.1 Goals of the Style Guide

Why do we have this document?

There are a few core goals that we believe this guide should serve. These are the fundamental **why**s that underlie all of the individual rules. By bringing these ideas to the fore, we hope to ground discussions and make it clearer to our broader community why the rules are in place and why particular decisions have been made. If you understand what goals each rule is serving, it should be clearer to everyone when a rule may be waived (some can be), and what sort of argument or alternative would be necessary to change a rule in the guide.

The goals of the style guide as we currently see them are as follows:

1.1.1 Style rules should pull their weight

The benefit of a style rule must be large enough to justify asking all of our engineers to remember it. The benefit is measured relative to the codebase we would get without the rule, so a rule against a very harmful practice may still have a small benefit if people are unlikely to do it anyway. This principle mostly explains the rules we don't have, rather than the rules we do: for example, `goto` contravenes many of the following principles, but is already vanishingly rare, so the Style Guide doesn't discuss it.

1.1.2 Optimize for the reader, not the writer

Our codebase (and most individual components submitted to it) is expected to continue for quite some time. As a result, more time will be spent reading most of our code than writing it. We explicitly choose to optimize for the experience of our average software engineer reading, maintaining, and debugging code in our codebase rather than ease when writing said code. "Leave a trace for the reader" is a particularly common sub-point of this principle: When something surprising or unusual is happening in a snippet of code (for example, transfer of pointer ownership), leaving textual hints for the reader at the point of use is valuable (`std::unique_ptr` demonstrates the ownership transfer unambiguously at the call site).

1.1.3 Be consistent with existing code

Using one style consistently through our codebase lets us focus on other (more important) issues. Consistency also allows for automation: tools that format your code or adjust your `#include` s only work properly when your code is consistent with the expectations of the tooling. In many cases, rules that are attributed to "Be Consistent" boil down to "Just pick one and stop worrying about it"; the potential value of allowing flexibility on these points is outweighed by the cost of having people argue over them. However, there are limits to consistency; it is a good tie breaker when there is no clear technical argument, nor a long-term direction. It applies more heavily locally (per file, or for a tightly-related set of interfaces). Consistency should not generally be used as a justification to do things in an old style without considering the benefits of the new style, or the tendency of the codebase to converge on newer styles over time.

1.1.4 Be consistent with the broader C++ community when appropriate

Consistency with the way other organizations use C++ has value for the same reasons as consistency within our code base. If a feature in the C++ standard solves a problem, or if some idiom is widely known and accepted, that's an argument for using it. However, sometimes standard features and idioms are flawed, or were just designed without our codebase's needs in mind. In those cases (as described below) it's appropriate to constrain or ban standard features. In some cases we prefer a homegrown or third-party library over a library defined in the C++ Standard, either out of perceived superiority or insufficient value to transition the codebase to the standard interface.

1.1.5 Avoid surprising or dangerous constructs

C++ has features that are more surprising or dangerous than one might think at a glance. Some style guide restrictions are in place to prevent falling into these pitfalls. There is a high bar for style guide waivers on such restrictions, because waiving such rules often directly risks compromising program correctness.

1.1.6 Avoid constructs that our average C++ programmer would find tricky or hard to maintain

C++ has features that may not be generally appropriate because of the complexity they introduce to the code. In widely used code, it may be more acceptable to use trickier language constructs, because any benefits of more complex implementation are multiplied widely by usage, and the cost in understanding the complexity does not need to be paid again when working with new portions of the codebase. When in doubt, waivers to rules of this type can be sought by asking your project leads. This is specifically important for our codebase because code ownership and team membership changes over time: even if everyone that works with some piece of code currently understands it, such understanding is not guaranteed to hold a few years from now.

1.1.7 Be mindful of our scale

With a codebase of 100+ million lines and thousands of engineers, some mistakes and simplifications for one engineer can become costly for many. For instance it's particularly important to avoid polluting the global namespace: name collisions across a codebase of hundreds of millions of lines are difficult to work with and hard to avoid if everyone puts things into the global namespace.

1.1.8 Concede to optimization when necessary

Performance optimizations can sometimes be necessary and appropriate, even when they conflict with the other principles of this document.

1.1.9 Summary

The intent of this document is to provide maximal guidance with reasonable restriction. As always, common sense and good taste should prevail. By this we specifically refer to the established conventions of the entire Google C++ community, not just your personal preferences or those of your team. Be skeptical about and reluctant to use clever or unusual constructs: the absence of a prohibition is not the same as a license to proceed. Use your judgment, and if you are unsure, please don't hesitate to ask your project leads to get additional input.

Chapter 2
C++ Version

Currently, code should target C++17, i.e., should not use C++2x features, with the exception of designated initializers. The C++ version targeted by this guide will advance (aggressively) over time.

Do not use non-standard extensions.

Consider portability to other environments before using features from C++14 and C++17 in your project.

Chapter 3
Header Files

Introduction

❑ *Self-contained Headers*
❑ *The `#define` Guard*
❑ *Include What You Use*
❑ *Forward Declarations*
❑ *Inline Functions*
❑ *Names and Order of Includes*

In general, every `.cc` file should have an associated `.h` file. There are some common exceptions, such as unit tests and small `.cc` files containing just a `main()` function.

Correct use of header files can make a huge difference to the readability, size and performance of your code.

The following rules will guide you through the various pitfalls of using header files.

3.1 Self-contained Headers

Header files should be self-contained (compile on their own) and end in `.h`. Non-header files that are meant for inclusion should end in `.inc` and be used sparingly.

All header files should be self-contained. Users and refactoring tools should not have to adhere to special conditions to include the header. Specifically, a header should have header guards and include all other headers it needs.

When a header declares inline functions or templates that clients of the header will instantiate, the inline functions and templates must also have definitions in the header, either directly or in files it includes. Do not move these definitions to separately included header (`-inl.h`) files; this practice was common in the past, but is no longer allowed. When all instantiations of a template occur in one `.cc` file, either because they're explicit or because the definition is accessible to only the `.cc` file, the template definition can be kept in that file.

There are rare cases where a file designed to be included is not self-contained. These are typically intended to be included at unusual locations, such as the middle of another file. They might not use header guards, and might not include their prerequisites. Name such files with the `.inc` extension. Use sparingly, and prefer self-contained headers when possible.

3.2 The `# define` Guard

All header files should have `#define` guards to prevent multiple inclusion. The format of the symbol name should be `<PROJECT>_<PATH>_<FILE>_H_`.

To guarantee uniqueness, they should be based on the full path in a project's source tree. For example, the file `foo/src/bar/baz.h` in project `foo` should have the following guard:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

```

```
...

#endif  // FOO_BAR_BAZ_H_
```

3.3 Include What You Use

If a source or header file refers to a symbol defined elsewhere, the file should directly include a header file which properly intends to provide a declaration or definition of that symbol. It should not include header files for any other reason.

Do not rely on transitive inclusions. This allows people to remove no-longer-needed `#include` statements from their headers without breaking clients. This also applies to related headers - `foo.cc` should include `bar.h` if it uses a symbol from it even if `foo.h` includes `bar.h`.

3.4 Forward Declarations

Avoid using forward declarations where possible. Instead, include the headers you need.

3.4.1 Definition

A "forward declaration" is a declaration of an entity without an associated definition.

```
// In a C++ source file:
class B;
void FuncInB();
extern int variable_in_b;
ABSL_DECLARE_FLAG(flag_in_b);
```

3.4.2 Pros

- Forward declarations can save compile time, as `#include`s force the compiler to open more files and process more input.
- Forward declarations can save on unnecessary recompilation. `#include`s can force your code to be recompiled more often, due to unrelated changes in the header.

3.4.3 Cons

- Forward declarations can hide a dependency, allowing user code to skip necessary recompilation when headers change.
- A forward declaration as opposed to an `#include` statement makes it difficult for automatic tooling to discover the module defining the symbol.

- A forward declaration may be broken by subsequent changes to the library. Forward declarations of functions and templates can prevent the header owners from making otherwise-compatible changes to their APIs, such as widening a parameter type, adding a template parameter with a default value, or migrating to a new namespace.
- Forward declaring symbols from namespace `std::` yields undefined behavior.
- It can be difficult to determine whether a forward declaration or a full `#include` is needed. Replacing an `#include` with a forward declaration can silently change the meaning of code:

```
// b.h:
struct B {};
struct D : B {};

// good_user.cc:
#include "b.h"
void f(B*);
void f(void*);
void test(D* x) { f(x); }  // Calls f(B*)
```

  If the `#include` was replaced with forward decls for `B` and `D`, `test()` would call `f(void*)`.
- Forward declaring multiple symbols from a header can be more verbose than simply `#include`ing the header.
- Structuring code to enable forward declarations (e.g., using pointer members instead of object members) can make the code slower and more complex.

3.4.4 Decision

Try to avoid forward declarations of entities defined in another project.

3.5 Inline Functions

Define functions inline only when they are small, say, 10 lines or fewer.

3.5.1 Definition

You can declare functions in a way that allows the compiler to expand them inline rather than calling them through the usual function call mechanism.

3.5.2 Pros

Inlining a function can generate more efficient object code, as long as the inlined function is small. Feel free to inline accessors and mutators, and other short, performance-critical functions.

3.5.3 Cons

Overuse of inlining can actually make programs slower. Depending on a function's size, inlining it can cause the code size to increase or decrease. Inlining a very small accessor function will usually decrease code size while inlining a very large function can dramatically increase code size. On modern processors smaller code usually runs faster due to better use of the instruction cache.

3.5.4 Decision

A decent rule of thumb is to not inline a function if it is more than 10 lines long. Beware of destructors, which are often longer than they appear because of implicit member- and base-destructor calls!

Another useful rule of thumb: it's typically not cost effective to inline functions with loops or switch statements (unless, in the common case, the loop or switch statement is never executed).

It is important to know that functions are not always inlined even if they are declared as such; for example, virtual and recursive functions are not normally inlined. Usually recursive functions should not be inline. The main reason for making a virtual function inline is to place its definition in the class, either for convenience or to document its behavior, e.g., for accessors and mutators.

3.6 Names and Order of Includes

Include headers in the following order: Related header, C system headers, C++ standard library headers, other libraries' headers, your project's headers.

All of a project's header files should be listed as descendants of the project's source directory without use of UNIX directory aliases `.` (the current directory) or `..` (the parent directory). For example, `google-awesome-project/s` should be included as:

```
#include "base/logging.h"
```

In `dir/foo.cc` or `dir/foo_test.cc`, whose main purpose is to implement or test the stuff in `dir2/foo2.h`, order your includes as follows:

1. `dir2/foo2.h`.
2. A blank line
3. C system headers (more precisely: headers in angle brackets with the `.h` extension), e.g., `<unistd.h>`, `<stdlib.h>`.
4. A blank line
5. C++ standard library headers (without file extension), e.g., `<algorithm>`, `<cstddef>`.
6. A blank line
7. Other libraries' `.h` files.
8. A blank line
9. Your project's `.h` files.

Separate each non-empty group with one blank line.

With the preferred ordering, if the related header `dir2/foo2.h` omits any necessary includes, the build of `dir/foo.cc` or `dir/foo_test.cc` will break. Thus, this rule ensures that build breaks show up first for the people working on these files, not for innocent people in other packages.

`dir/foo.cc` and `dir2/foo2.h` are usually in the same directory (e.g., `base/basictypes_test.cc` and `base/basictypes.h`), but may sometimes be in different directories too.

Note that the C headers such as `stddef.h` are essentially interchangeable with their C++ counterparts (`cstddef`). Either style is acceptable, but prefer consistency with existing code.

Within each section the includes should be ordered alphabetically. Note that older code might not conform to this rule and should be fixed when convenient.

For example, the includes in `google-awesome-project/src/foo/internal/fooserver.cc` might look like this:

```
#include "foo/server/fooserver.h"

#include <sys/types.h>
#include <unistd.h>

#include <string>
#include <vector>

#include "base/basictypes.h"
#include "foo/server/bar.h"
#include "third_party/absl/flags/flag.h"
```

3.6.1 Exception

Sometimes, system-specific code needs conditional includes. Such code can put conditional includes after other includes. Of course, keep your system-specific code small and localized. Example:

```
#include "foo/public/fooserver.h"

#include "base/port.h"  // For LANG_CXX11.

#ifdef LANG_CXX11
#include <initializer_list>
#endif  // LANG_CXX11
```

Chapter 4
Scoping

Introduction

- ❏ *Namespaces*
- ❏ *Internal Linkage*
- ❏ *Nonmember, Static Member, and Global Functions*
- ❏ *Local Variables*
- ❏ *Static and Global Variables*
- ❏ `thread_local` *Variables*

4.1 Namespaces

With few exceptions, place code in a namespace. Namespaces should have unique names based on the project name, and possibly its path. Do not use *using-directives* (e.g., `using namespace foo`). Do not use inline namespaces. For unnamed namespaces, see Internal Linkage.

4.1.1 Definition

Namespaces subdivide the global scope into distinct, named scopes, and so are useful for preventing name collisions in the global scope.

4.1.2 Pros

Namespaces provide a method for preventing name conflicts in large programs while allowing most code to use reasonably short names.

For example, if two different projects have a class `Foo` in the global scope, these symbols may collide at compile time or at runtime. If each project places their code in a namespace, `project1::Foo` and `project2::Foo` are now distinct symbols that do not collide, and code within each project's namespace can continue to refer to `Foo` without the prefix.

Inline namespaces automatically place their names in the enclosing scope. Consider the following snippet, for example:

```
namespace outer {
inline namespace inner {
  void foo();
}  // namespace inner
}  // namespace outer
```

The expressions `outer::inner::foo()` and `outer::foo()` are interchangeable. Inline namespaces are primarily intended for ABI compatibility across versions.

4.1.3 Cons

Namespaces can be confusing, because they complicate the mechanics of figuring out what definition a name refers to.

Inline namespaces, in particular, can be confusing because names aren't actually restricted to the namespace where they are declared. They are only useful as part of some larger versioning policy.

In some contexts, it's necessary to repeatedly refer to symbols by their fully-qualified names. For deeply-nested namespaces, this can add a lot of clutter.

4.1.4 Decision

Namespaces should be used as follows:

- Follow the rules on Namespace Names.
- Terminate multi-line namespaces with comments as shown in the given examples.
- Namespaces wrap the entire source file after includes, gflags definitions/declarations and forward declarations of classes from other namespaces.

```
// In the .h file
namespace mynamespace {

// All declarations are within the namespace scope.
// Notice the lack of indentation.
class MyClass {
 public:
  ...
  void Foo();
};

}  // namespace mynamespace
```

```
// In the .cc file
namespace mynamespace {

// Definition of functions is within scope of the namespace.
void MyClass::Foo() {
  ...
}

}  // namespace mynamespace
```

More complex `.cc` files might have additional details, like flags or using-declarations.

```
#include "a.h"

ABSL_FLAG(bool, someflag, false, "a flag");

namespace mynamespace {

using ::foo::Bar;

...code for mynamespace...    // Code goes against the left margin.

}  // namespace mynamespace
```

- To place generated protocol message code in a namespace, use the `package` specifier in the `.proto` file. See Protocol Buffer Packages for details.
- Do not declare anything in namespace `std`, including forward declarations of standard library classes. Declaring entities in namespace `std` is undefined behavior, i.e., not portable. To declare entities from the standard library, include the appropriate header file.
- You may not use a *using-directive* to make all names from a namespace available.

```
// Forbidden -- This pollutes the namespace.
using namespace foo;
```

- Do not use *Namespace aliases* at namespace scope in header files except in explicitly marked internal-only namespaces, because anything imported into a namespace in a header file becomes part of the public API exported by that file.

```
// Shorten access to some commonly used names in .cc files.
namespace baz = ::foo::bar::baz;
```

```
// Shorten access to some commonly used names (in a .h file).
namespace librarian {
namespace impl {  // Internal, not part of the API.
namespace sidetable = ::pipeline_diagnostics::sidetable;
}  // namespace impl

inline void my_inline_function() {
  // namespace alias local to a function (or method).
  namespace baz = ::foo::bar::baz;
  ...
}
}  // namespace librarian
```

- Do not use inline namespaces.

4.2 Internal Linkage

When definitions in a `.cc` file do not need to be referenced outside that file, give them internal linkage by placing them in an unnamed namespace or declaring them `static`. Do not use either of these constructs in `.h` files.

4.2.1 Definition

All declarations can be given internal linkage by placing them in unnamed namespaces. Functions and variables can also be given internal linkage by declaring them `static`. This means that anything you're declaring can't be accessed from another file. If a different file declares something with the same name, then the two entities are completely independent.

4.2.2 Decision

Use of internal linkage in `.cc` files is encouraged for all code that does not need to be referenced elsewhere. Do not use internal linkage in `.h` files.

Format unnamed namespaces like named namespaces. In the terminating comment, leave the namespace name empty:

```
namespace {
...
}  // namespace
```

4.3 Nonmember, Static Member, and Global Functions

Prefer placing nonmember functions in a namespace; use completely global functions rarely. Do not use a class simply to group static members. Static methods of a class should generally be closely related to instances of the class or the class's static data.

4.3.1 Pros

Nonmember and static member functions can be useful in some situations. Putting nonmember functions in a namespace avoids polluting the global namespace.

4.3.2 Cons

Nonmember and static member functions may make more sense as members of a new class, especially if they access external resources or have significant dependencies.

4.3.3 Decision

Sometimes it is useful to define a function not bound to a class instance. Such a function can be either a static member or a nonmember function. Nonmember functions should not depend on external variables, and should nearly always exist in a namespace. Do not create classes only to group static members; this is no different than just giving the names a common prefix, and such grouping is usually unnecessary anyway.

If you define a nonmember function and it is only needed in its `.cc` file, use internal linkage to limit its scope.

4.4 Local Variables

Place a function's variables in the narrowest scope possible, and initialize variables in the declaration.

C++ allows you to declare variables anywhere in a function. We encourage you to declare them in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to. In particular, initialization should be used instead of declaration and assignment, e.g.,:

```
int i;
i = f();      // Bad -- initialization separate from declaration.
```

```
int j = g();  // Good -- declaration has initialization.
```

```
std::vector<int> v;
v.push_back(1);  // Prefer initializing using brace initialization.
v.push_back(2);
```

```
std::vector<int> v = {1, 2};  // Good -- v starts initialized.
```

Variables needed for `if`, `while` and `for` statements should normally be declared within those statements, so that such variables are confined to those scopes. E.g.:

```
while (const char* p = strchr(str, '/')) str = p + 1;
```

There is one caveat: if the variable is an object, its constructor is invoked every time it enters scope and is created, and its destructor is invoked every time it goes out of scope.

```
// Inefficient implementation:
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // My ctor and dtor get called 1000000 times each.
  f.DoSomething(i);
}
```

It may be more efficient to declare such a variable used in a loop outside that loop:

```
Foo f;  // My ctor and dtor get called once each.
for (int i = 0; i < 1000000; ++i) {
  f.DoSomething(i);
}
```

4.5 Static and Global Variables

Objects with static storage duration are forbidden unless they are trivially destructible. Informally this means that the destructor does not do anything, even taking member and base destructors into account. More formally it means that the type has no user-defined or virtual destructor and that all bases and non-static members are trivially destructible. Static function-local variables may use dynamic initialization. Use of dynamic initialization for static class member variables or variables at namespace scope is discouraged, but allowed in limited circumstances; see below for details.

As a rule of thumb: a global variable satisfies these requirements if its declaration, considered in isolation, could be `constexpr`.

4.5.1 Definition

Every object has a *storage duration*, which correlates with its lifetime. Objects with static storage duration live from the point of their initialization until the end of the program. Such objects appear as variables at namespace scope ("global variables"), as static data members of classes, or as function-local variables that are declared with the `static` specifier. Function-local static variables are initialized when control first passes through their declaration; all other objects with static storage duration are initialized as part of program start-up. All objects with static storage duration are destroyed at program exit (which happens before unjoined threads are terminated).

Initialization may be *dynamic*, which means that something non-trivial happens during initialization. (For example, consider a constructor that allocates memory, or a variable that is initialized with the current process ID.) The other kind of initialization is *static* initialization. The two aren't quite opposites, though: `static` initialization *always* happens to objects with static storage duration (initializing the object either to a given constant or to a representation consisting of all bytes set to zero), whereas dynamic initialization happens after that, if required.

4.5.2 Pros

Global and static variables are very useful for a large number of applications: named constants, auxiliary data structures internal to some translation unit, command-line flags, logging, registration mechanisms, background infrastructure, etc.

4.5.3 Cons

Global and static variables that use dynamic initialization or have non-trivial destructors create complexity that can easily lead to hard-to-find bugs. Dynamic initialization is not ordered across translation units, and neither is destruction (except that destruction happens in reverse order of initialization). When one initialization refers to another variable with static storage duration, it is possible that this causes an object to be accessed before its lifetime has begun (or after its lifetime has ended). Moreover, when a program starts threads that are not joined at exit, those threads may attempt to access objects after their lifetime has ended if their destructor has already run.

4.5.4 Decision

4.5.4.1 Decision on destruction

When destructors are trivial, their execution is not subject to ordering at all (they are effectively not "run"); otherwise we are exposed to the risk of accessing objects after the end of their lifetime. Therefore, we only allow objects with static storage duration if they are trivially destructible. Fundamental types (like pointers and `int`) are trivially destructible, as are arrays of trivially destructible types. Note that variables marked with `constexpr` are trivially destructible.

```
const int kNum = 10;  // Allowed

struct X { int n; };
const X kX[] = {{1}, {2}, {3}};  // Allowed

void foo() {
  static const char* const kMessages[] = {"hello", "world"};  // Allowed
}

// Allowed: constexpr guarantees trivial destructor.
constexpr std::array<int, 3> kArray = {1, 2, 3};
```

```
// bad: non-trivial destructor
const std::string kFoo = "foo";

// Bad for the same reason, even though kBar is a reference (the
// rule also applies to lifetime-extended temporary objects).
```

```cpp
const std::string& kBar = StrCat("a", "b", "c");

void bar() {
  // Bad: non-trivial destructor.
  static std::map<int, int> kData = {{1, 0}, {2, 0}, {3, 0}};
}
```

Note that references are not objects, and thus they are not subject to the constraints on destructibility. The constraint on dynamic initialization still applies, though. In particular, a function-local static reference of the form `static T& t = *new T;` is allowed.

4.5.4.2 Decision on initialization

Initialization is a more complex topic. This is because we must not only consider whether class constructors execute, but we must also consider the evaluation of the initializer:

```cpp
int n = 5;    // Fine
int m = f();  // ? (Depends on f)
Foo x;        // ? (Depends on Foo::Foo)
Bar y = g();  // ? (Depends on g and on Bar::Bar)
```

All but the first statement expose us to indeterminate initialization ordering.

The concept we are looking for is called *constant initialization* in the formal language of the C++ standard. It means that the initializing expression is a constant expression, and if the object is initialized by a constructor call, then the constructor must be specified as `constexpr`, too:

```cpp
struct Foo { constexpr Foo(int) {} };

int n = 5;  // Fine, 5 is a constant expression.
Foo x(2);   // Fine, 2 is a constant expression and the chosen constructor is
↪  constexpr.
Foo a[] = { Foo(1), Foo(2), Foo(3) };  // Fine
```

Constant initialization is always allowed. Constant initialization of static storage duration variables should be marked with `constexpr` or where possible the ABSL_CONST_INIT attribute. Any non-local static storage duration variable that is not so marked should be presumed to have dynamic initialization, and reviewed very carefully.

By contrast, the following initializations are problematic:

```cpp
// Some declarations used below.
time_t time(time_t*);      // Not constexpr!
int f();                   // Not constexpr!
```

```
struct Bar { Bar() {} };

// Problematic initializations.
time_t m = time(nullptr);  // Initializing expression not a constant expression.
Foo y(f());                // Ditto
Bar b;                     // Chosen constructor Bar::Bar() not constexpr.
```

Dynamic initialization of nonlocal variables is discouraged, and in general it is forbidden. However, we do permit it if no aspect of the program depends on the sequencing of this initialization with respect to all other initializations. Under those restrictions, the ordering of the initialization does not make an observable difference. For example:

```
int p = getpid();  // Allowed, as long as no other static variable
                   // uses p in its own initialization.
```

Dynamic initialization of static local variables is allowed (and common).

4.5.4.3 Common patterns

- Global strings: if you require a named global or static string constant, consider using a `constexpr` variable of `string_view`, character array, or character pointer, pointing to a string literal. String literals have static storage duration already and are usually sufficient. See TotW #140.
- Maps, sets, and other dynamic containers: if you require a static, fixed collection, such as a set to search against or a lookup table, you cannot use the dynamic containers from the standard library as a static variable, since they have non-trivial destructors. Instead, consider a simple array of trivial types, e.g., an array of arrays of ints (for a “map from int to int”), or an array of pairs (e.g., pairs of `int` and `const char*`). For small collections, linear search is entirely sufficient (and efficient, due to memory locality); consider using the facilities from absl/algorithm/container.h for the standard operations. If necessary, keep the collection in sorted order and use a binary search algorithm. If you do really prefer a dynamic container from the standard library, consider using a function-local static pointer, as described below .
- Smart pointers (`unique_ptr`, `shared_ptr`): smart pointers execute cleanup during destruction and are therefore forbidden. Consider whether your use case fits into one of the other patterns described in this section. One simple solution is to use a plain pointer to a dynamically allocated object and never delete it (see last item).
- Static variables of custom types: if you require static, constant data of a type that you need to define yourself, give the type a trivial destructor and a `constexpr` constructor.
- If all else fails, you can create an object dynamically and never delete it by using a function-local static pointer or reference (e.g., `static const auto& impl = *new T(args...);`).

4.6 `thread_local` Variables

`thread_local` variables that aren’t declared inside a function must be initialized with a true compile-time constant, and this must be enforced by using the ABSL_CONST_INIT attribute. Prefer `thread_local` over other ways of defining thread-local data.

4.6.1 Definition

Variables can be declared with the `thread_local` specifier:

```
thread_local Foo foo = ...;
```

Such a variable is actually a collection of objects, so that when different threads access it, they are actually accessing different objects. `thread_local` variables are much like static storage duration variables in many respects. For instance, they can be declared at namespace scope, inside functions, or as static class members, but not as ordinary class members.

`thread_local` variable instances are initialized much like static variables, except that they must be initialized separately for each thread, rather than once at program startup. This means that `thread_local` variables declared within a function are safe, but other `thread_local` variables are subject to the same initialization-order issues as static variables (and more besides).

`thread_local` variable instances are not destroyed before their thread terminates, so they do not have the destruction-order issues of static variables.

4.6.2 Pros

- Thread-local data is inherently safe from races (because only one thread can ordinarily access it), which makes `thread_local` useful for concurrent programming.
- `thread_local` is the only standard-supported way of creating thread-local data.

4.6.3 Cons

- Accessing a `thread_local` variable may trigger execution of an unpredictable and uncontrollable amount of other code.
- `thread_local` variables are effectively global variables, and have all the drawbacks of global variables other than lack of thread-safety.
- The memory consumed by a `thread_local` variable scales with the number of running threads (in the worst case), which can be quite large in a program.
- Non-static data members cannot be `thread_local`.
- `thread_local` may not be as efficient as certain compiler intrinsics.

4.6.4 Decision

`thread_local` variables inside a function have no safety concerns, so they can be used without restriction. Note that you can use a function-scope `thread_local` to simulate a class- or namespace-scope `thread_local` by defining a function or static method that exposes it:

```
Foo& MyThreadLocalFoo() {
  thread_local Foo result = ComplicatedInitialization();
  return result;
}
```

thread_local variables at class or namespace scope must be initialized with a true compile-time constant (i.e., they must have no dynamic initialization). To enforce this, thread_local variables at class or namespace scope must be annotated with ABSL_CONST_INIT (or constexpr, but that should be rare):

```
ABSL_CONST_INIT thread_local Foo foo = ...;
```

thread_local should be preferred over other mechanisms for defining thread-local data.

Chapter 5
Classes

Introduction

- ❑ *Doing Work in Constructors*
- ❑ *Implicit Conversions*
- ❑ *Copyable and Movable Types*
- ❑ *Structs vs. Classes*
- ❑ *Structs vs. Pairs and Tuples*
- ❑ *Inheritance*
- ❑ *Operator Overloading*
- ❑ *Access Control*
- ❑ *Declaration Order*

Classes are the fundamental unit of code in C++. Naturally, we use them extensively. This section lists the main dos and don’ts you should follow when writing a class.

5.1 Doing Work in Constructors

Avoid virtual method calls in constructors, and avoid initialization that can fail if you can’t signal an error.

5.1.1 Definition

It is possible to perform arbitrary initialization in the body of the constructor.

5.1.2 Pros

- No need to worry about whether the class has been initialized or not.
- Objects that are fully initialized by constructor call can be `const` and may also be easier to use with standard containers or algorithms.

5.1.3 Cons

- If the work calls virtual functions, these calls will not get dispatched to the subclass implementations. Future modification to your class can quietly introduce this problem even if your class is not currently subclassed, causing much confusion.
- There is no easy way for constructors to signal errors, short of crashing the program (not always appropriate) or using exceptions (which are forbidden).
- If the work fails, we now have an object whose initialization code failed, so it may be an unusual state requiring a `bool IsValid()` state checking mechanism (or similar) which is easy to forget to call.
- You cannot take the address of a constructor, so whatever work is done in the constructor cannot easily be handed off to, for example, another thread.

5.1.4 Decision

Constructors should never call virtual functions. If appropriate for your code , terminating the program may be an appropriate error handling response. Otherwise, consider a factory function or `Init()` method as described in TotW #42. Avoid `Init()` methods on objects with no other states that affect which public methods may be called (semi-constructed objects of this form are particularly hard to work with correctly).

5.2 Implicit Conversions

Do not define implicit conversions. Use the `explicit` keyword for conversion operators and single-argument constructors.

5.2.1 Definition

Implicit conversions allow an object of one type (called the *source type*) to be used where a different type (called the *destination type*) is expected, such as when passing an `int` argument to a function that takes a `double` parameter.

In addition to the implicit conversions defined by the language, users can define their own, by adding appropriate members to the class definition of the source or destination type. An implicit conversion in the source type is defined by a type conversion operator named after the destination type (e.g., `operator bool()`). An implicit conversion in the destination type is defined by a constructor that can take the source type as its only argument (or only argument with no default value).

The `explicit` keyword can be applied to a constructor or a conversion operator, to ensure that it can only be used when the destination type is explicit at the point of use, e.g., with a cast. This applies not only to implicit conversions, but to list initialization syntax:

```
class Foo {
  explicit Foo(int x, double y);
  ...
};

void Func(Foo f);
```

```
Func({42, 3.14});  // Error
```

This kind of code isn't technically an implicit conversion, but the language treats it as one as far as `explicit` is concerned.

5.2.2 Pros

- Implicit conversions can make a type more usable and expressive by eliminating the need to explicitly name a type when it's obvious.
- Implicit conversions can be a simpler alternative to overloading, such as when a single function with a `string_view` parameter takes the place of separate overloads for `std::string` and `const char*`.
- List initialization syntax is a concise and expressive way of initializing objects.

5.2.3 Cons

- Implicit conversions can hide type-mismatch bugs, where the destination type does not match the user's expectation, or the user is unaware that any conversion will take place.
- Implicit conversions can make code harder to read, particularly in the presence of overloading, by making it less obvious what code is actually getting called.
- Constructors that take a single argument may accidentally be usable as implicit type conversions, even if they are not intended to do so.
- When a single-argument constructor is not marked `explicit`, there's no reliable way to tell whether it's intended to define an implicit conversion, or the author simply forgot to mark it.
- Implicit conversions can lead to call-site ambiguities, especially when there are bidirectional implicit conversions. This can be caused either by having two types that both provide an implicit conversion, or by a single type that has both an implicit constructor and an implicit type conversion operator.
- List initialization can suffer from the same problems if the destination type is implicit, particularly if the list has only a single element.

5.2.4 Decision

Type conversion operators, and constructors that are callable with a single argument, must be marked `explicit` in the class definition. As an exception, copy and move constructors should not be `explicit`, since they do not perform type conversion.

Implicit conversions can sometimes be necessary and appropriate for types that are designed to be interchangeable, for example when objects of two types are just different representations of the same underlying value. In that case, contact your project leads to request a waiver of this rule.

Constructors that cannot be called with a single argument may omit `explicit`. Constructors that take a single `std::initializer_list` parameter should also omit `explicit`, in order to support copy-initialization (e.g., `MyType m = {1, 2};`).

5.3 Copyable and Movable Types

A class's public API must make clear whether the class is copyable, move-only, or neither copyable nor movable. Support copying and/or moving if these operations are clear and meaningful for your type.

5.3.1 Definition

A movable type is one that can be initialized and assigned from temporaries.

A copyable type is one that can be initialized or assigned from any other object of the same type (so is also movable by definition), with the stipulation that the value of the source does not change. `std::unique_ptr<int>` is an example of a movable but not copyable type (since the value of the source `std::unique_ptr<int>` must be modified during assignment to the destination). `int` and `std::string` are examples of movable types that are also copyable. (For `int`, the move and copy operations are the same; for `std::string`, there exists a move operation that is less expensive than a copy.)

For user-defined types, the copy behavior is defined by the copy constructor and the copy-assignment operator. Move behavior is defined by the move constructor and the move-assignment operator, if they exist, or by the copy constructor and the copy-assignment operator otherwise.

The copy/move constructors can be implicitly invoked by the compiler in some situations, e.g., when passing objects by value.

5.3.2 Pros

Objects of copyable and movable types can be passed and returned by value, which makes APIs simpler, safer, and more general. Unlike when passing objects by pointer or reference, there's no risk of confusion over ownership, lifetime, mutability, and similar issues, and no need to specify them in the contract. It also prevents non-local interactions between the client and the implementation, which makes them easier to understand, maintain, and optimize by the compiler. Further, such objects can be used with generic APIs that require pass-by-value, such as most containers, and they allow for additional flexibility in e.g., type composition.

Copy/move constructors and assignment operators are usually easier to define correctly than alternatives like `Clone()`, `CopyFrom()` or `Swap()`, because they can be generated by the compiler, either implicitly or with `= default`. They are concise, and ensure that all data members are copied. Copy and move constructors are also generally more efficient, because they don't require heap allocation or separate initialization and assignment steps, and they're eligible for optimizations such as copy elision.

Move operations allow the implicit and efficient transfer of resources out of rvalue objects. This allows a plainer coding style in some cases.

5.3.3 Cons

Some types do not need to be copyable, and providing copy operations for such types can be confusing, nonsensical, or outright incorrect. Types representing singleton objects (`Registerer`), objects tied to a specific scope (`Cleanup`), or closely coupled to object identity (`Mutex`) cannot be copied meaningfully. Copy operations for base class types that are to be used polymorphically are hazardous, because use of them can lead to object slicing. Defaulted or carelessly-implemented copy operations can be incorrect, and the resulting bugs can be confusing and difficult to diagnose.

Copy constructors are invoked implicitly, which makes the invocation easy to miss. This may cause confusion for programmers used to languages where pass-by-reference is conventional or mandatory. It may also encourage excessive copying, which can cause performance problems.

5.3.4 Decision

Every class's public interface must make clear which copy and move operations the class supports. This should usually take the form of explicitly declaring and/or deleting the appropriate operations in the `public` section of the declaration.

Specifically, a copyable class should explicitly declare the copy operations, a move-only class should explicitly declare the move operations, and a non-copyable/movable class should explicitly delete the copy operations. A copyable class may also declare move operations in order to support efficient moves. Explicitly declaring or deleting all four copy/move operations is permitted, but not required. If you provide a copy or move assignment operator, you must also provide the corresponding constructor.

```
class Copyable {
 public:
  Copyable(const Copyable& other) = default;
  Copyable& operator=(const Copyable& other) = default;

  // The implicit move operations are suppressed by the declarations above.
  // You may explicitly declare move operations to support efficient moves.
};

class MoveOnly {
 public:
  MoveOnly(MoveOnly&& other) = default;
  MoveOnly& operator=(MoveOnly&& other) = default;

  // The copy operations are implicitly deleted, but you can
  // spell that out explicitly if you want:
  MoveOnly(const MoveOnly&) = delete;
  MoveOnly& operator=(const MoveOnly&) = delete;
};

class NotCopyableOrMovable {
 public:
  // Not copyable or movable
  NotCopyableOrMovable(const NotCopyableOrMovable&) = delete;
  NotCopyableOrMovable& operator=(const NotCopyableOrMovable&)
      = delete;

  // The move operations are implicitly disabled, but you can
  // spell that out explicitly if you want:
  NotCopyableOrMovable(NotCopyableOrMovable&&) = delete;
  NotCopyableOrMovable& operator=(NotCopyableOrMovable&&)
      = delete;
};
```

These declarations/deletions can be omitted only if they are obvious:

- If the class has no `private` section, like a struct or an interface-only base class, then the copyability/movability can be determined by the copyability/movability of any public data members.
- If a base class clearly isn't copyable or movable, derived classes naturally won't be either. An interface-only base class that leaves these operations implicit is not sufficient to make concrete subclasses clear.
- Note that if you explicitly declare or delete either the constructor or assignment operation for copy, the other copy operation is not obvious and must be declared or deleted. Likewise for move operations.

A type should not be copyable/movable if the meaning of copying/moving is unclear to a casual user, or if it incurs unexpected costs. Move operations for copyable types are strictly a performance optimization and are a potential source of bugs and complexity, so avoid defining them unless they are significantly more efficient than the corresponding copy operations. If your type provides copy operations, it is recommended that you design your class so that the default implementation of those operations is correct. Remember to review the correctness of any defaulted operations as you would any other code.

To eliminate the risk of slicing, prefer to make base classes abstract, by making their constructors protected, by declaring their destructors protected, or by giving them one or more pure virtual member functions. Prefer to avoid deriving from concrete classes.

5.4 Structs vs. Classes

Use a `struct` only for passive objects that carry data; everything else is a `class`.

The `struct` and `class` keywords behave almost identically in C++. We add our own semantic meanings to each keyword, so you should use the appropriate keyword for the data-type you're defining.

`struct`s should be used for passive objects that carry data, and may have associated constants. All fields must be public. The struct must not have invariants that imply relationships between different fields, since direct user access to those fields may break those invariants. Constructors, destructors, and helper methods may be present; however, these methods must not require or enforce any invariants.

If more functionality or invariants are required, a `class` is more appropriate. If in doubt, make it a `class`.

For consistency with STL, you can use `struct` instead of `class` for stateless types, such as traits, template metafunctions, and some functors.

Note that member variables in structs and classes have different naming rules.

5.5 Structs vs. Pairs and Tuples

Prefer to use a `struct` instead of a pair or a tuple whenever the elements can have meaningful names.

While using pairs and tuples can avoid the need to define a custom type, potentially saving work when *writing* code, a meaningful field name will almost always be much clearer when *reading* code than `.first`, `.second`, or `std::get<X>`. While C++14's introduction of `std::get<Type>` to access a tuple element by type rather than index (when the type is unique) can sometimes partially mitigate this, a field name is usually substantially clearer and more informative than a type.

Pairs and tuples may be appropriate in generic code where there are not specific meanings for the elements of the pair or tuple. Their use may also be required in order to interoperate with existing code or APIs.

5.6 Inheritance

Composition is often more appropriate than inheritance. When using inheritance, make it `public`.

5.6.1 Definition

When a sub-class inherits from a base class, it includes the definitions of all the data and operations that the base class defines. "Interface inheritance" is inheritance from a pure abstract base class (one with no state or defined methods); all other inheritance is "implementation inheritance".

5.6.2 Pros

Implementation inheritance reduces code size by re-using the base class code as it specializes an existing type. Because inheritance is a compile-time declaration, you and the compiler can understand the operation and detect errors. Interface inheritance can be used to programmatically enforce that a class expose a particular API. Again, the compiler can detect errors, in this case, when a class does not define a necessary method of the API.

5.6.3 Cons

For implementation inheritance, because the code implementing a sub-class is spread between the base and the sub-class, it can be more difficult to understand an implementation. The sub-class cannot override functions that are not virtual, so the sub-class cannot change implementation.

Multiple inheritance is especially problematic, because it often imposes a higher performance overhead (in fact, the performance drop from single inheritance to multiple inheritance can often be greater than the performance drop from ordinary to virtual dispatch), and because it risks leading to "diamond" inheritance patterns, which are prone to ambiguity, confusion, and outright bugs.

5.6.4 Decision

- All inheritance should be `public`. If you want to do private inheritance, you should be including an instance of the base class as a member instead. You may use `final` on classes when you don't intend to support using them as base classes.
- Do not overuse implementation inheritance. Composition is often more appropriate. Try to restrict use of inheritance to the "is-a" case: `Bar` subclasses `Foo` if it can reasonably be said that `Bar` "s a kind of" `Foo`.
- Limit the use of `protected` to those member functions that might need to be accessed from subclasses. Note that data members should be private.
- Explicitly annotate overrides of virtual functions or virtual destructors with exactly one of an `override` or (less frequently) `final` specifier. Do not use `virtual` when declaring an `override`. Rationale: A function or destructor marked `override` or `final` that is not an override of a base class virtual function will not compile, and this helps catch common errors. The specifiers serve as documentation; if no specifier is present, the reader has to check all ancestors of the class in question to determine if the function or destructor is virtual or not.
- Multiple inheritance is permitted, but multiple *implementation* inheritance is strongly discouraged.

5.7 Operator Overloading

Overload operators judiciously. Do not use user-defined literals.

5.7.1 Definition

C++ permits user code to declare overloaded versions of the built-in operators using the `operator` keyword, so long as one of the parameters is a user-defined type. The `operator` keyword also permits user code to define new kinds of literals using `operator""`, and to define type-conversion functions such as `operator bool()`.

5.7.2 Pros

Operator overloading can make code more concise and intuitive by enabling user-defined types to behave the same as built-in types. Overloaded operators are the idiomatic names for certain operations (e.g., `==`, `<`, `=`, and `<<`), and adhering to those conventions can make user-defined types more readable and enable them to interoperate with libraries that expect those names.

User-defined literals are a very concise notation for creating objects of user-defined types.

5.7.3 Cons

- Providing a correct, consistent, and unsurprising set of operator overloads requires some care, and failure to do so can lead to confusion and bugs.
- Overuse of operators can lead to obfuscated code, particularly if the overloaded operator's semantics don't follow convention.
- The hazards of function overloading apply just as much to operator overloading, if not more so.
- Operator overloads can fool our intuition into thinking that expensive operations are cheap, built-in operations.
- Finding the call sites for overloaded operators may require a search tool that's aware of C++ syntax, rather than e.g., grep.
- If you get the argument type of an overloaded operator wrong, you may get a different overload rather than a compiler error. For example, `foo < bar` may do one thing, while `&foo < &bar` does something totally different.
- Certain operator overloads are inherently hazardous. Overloading unary `&` can cause the same code to have different meanings depending on whether the overload declaration is visible. Overloads of `&&`, `||`, and `,` (comma) cannot match the evaluation-order semantics of the built-in operators.
- Operators are often defined outside the class, so there's a risk of different files introducing different definitions of the same operator. If both definitions are linked into the same binary, this results in undefined behavior, which can manifest as subtle run-time bugs.
- User-defined literals (UDLs) allow the creation of new syntactic forms that are unfamiliar even to experienced C++ programmers, such as `"Hello World"sv` as a shorthand for `std::string_view("Hello World")`. Existing notations are clearer, though less terse.
- Because they can't be namespace-qualified, uses of UDLs also require use of either using-directives (which we ban) or using-declarations (which we ban in header files except when the imported names are part of the interface exposed by the header file in question). Given that header files would have to avoid UDL suffixes, we prefer to avoid having conventions for literals differ between header files and source files.

5.7.4 Decision

Define overloaded operators only if their meaning is obvious, unsurprising, and consistent with the corresponding built-in operators. For example, use `|` as a bitwise- or logical-or, not as a shell-style pipe.

Define operators only on your own types. More precisely, define them in the same headers, `.cc` files, and namespaces as the types they operate on. That way, the operators are available wherever the type is, minimizing the risk of multiple definitions. If possible, avoid defining operators as templates, because they must satisfy this rule for any possible template arguments. If you define an operator, also define any related operators that make sense, and make sure they are defined consistently. For example, if you overload `<`, overload all the comparison operators, and make sure `<` and `>` never return true for the same arguments.

Prefer to define non-modifying binary operators as non-member functions. If a binary operator is defined as a class member, implicit conversions will apply to the right-hand argument, but not the left-hand one. It will confuse your users if `a < b` compiles but `b < a` doesn't.

Don't go out of your way to avoid defining operator overloads. For example, prefer to define `==`, `=`, and `<<`, rather than `Equals()`, `CopyFrom()`, and `PrintTo()`. Conversely, don't define operator overloads just because other libraries expect them. For example, if your type doesn't have a natural ordering, but you want to store it in a `std::set`, use a custom comparator rather than overloading `<`.

Do not overload `&&`, `||`, `,` (comma), or unary `&`. Do not overload `operator""`, i.e., do not introduce user-defined literals. Do not use any such literals provided by others (including the standard library).

Type conversion operators are covered in the section on implicit conversions. The `=` operator is covered in the section on copy constructors. Overloading `<<` for use with streams is covered in the section on streams. See also the rules on function overloading, which apply to operator overloading as well.

5.8 Access Control

Make classes' data members `private`, unless they are constants. This simplifies reasoning about invariants, at the cost of some easy boilerplate in the form of accessors (usually `const`) if necessary.

For technical reasons, we allow data members of a test fixture class defined in a `.cc` file to be `protected` when using Google Test. If a test fixture class is defined outside of the `.cc` file it is used in, for example in a `.h` file, make data members `private`.

5.9 Declaration Order

Group similar declarations together, placing public parts earlier.

A class definition should usually start with a `public:` section, followed by `protected:`, then `private:`. Omit sections that would be empty.

Within each section, prefer grouping similar kinds of declarations together, and prefer the following order:

1. Types and type aliases (`typedef`, `using`, `enum`, nested structs and classes, and `friend` types)
2. Static constants
3. Factory functions
4. Constructors and assignment operators
5. Destructor
6. All other functions (`static` and non-`static` member functions, and `friend` functions)
7. Data members (`static` and non-`static`)

Do not put large method definitions inline in the class definition. Usually, only trivial or performance-critical, and very short, methods may be defined inline. See Inline Functions for more details.

Chapter 6
Functions

Introduction

- ❏ *Inputs and Outputs*
- ❏ *Write Short Functions*
- ❏ *Function Overloading*
- ❏ *Default Arguments*
- ❏ *Trailing Return Type Syntax*

6.1 Inputs and Outputs

The output of a C++ function is naturally provided via a return value and sometimes via output parameters (or in/out parameters).

Prefer using return values over output parameters: they improve readability, and often provide the same or better performance.

Prefer to return by value or, failing that, return by reference. Avoid returning a pointer unless it can be `null`.

Parameters are either inputs to the function, outputs from the function, or both. Non-optional input parameters should usually be values or `const` references, while non-optional output and input/output parameters should usually be references (which cannot be `null`). Generally, use `std::optional` to represent optional by-value inputs, and use a `const` pointer when the non-optional form would have used a reference. Use non-`const` pointers to represent optional outputs and optional input/output parameters.

Avoid defining functions that require a `const` reference parameter to outlive the call, because `const` reference parameters bind to temporaries. Instead, find a way to eliminate the lifetime requirement (for example, by copying the parameter), or pass it by `const` pointer and document the lifetime and non-null requirements.

When ordering function parameters, put all input-only parameters before any output parameters. In particular, do not add new parameters to the end of the function just because they are new; place new input-only parameters before the output parameters. This is not a hard-and-fast rule. Parameters that are both input and output muddy the waters, and, as always, consistency with related functions may require you to bend the rule. Variadic functions may also require unusual parameter ordering.

6.2 Write Short Functions

Prefer small and focused functions.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code. Small functions are also easier to test.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

6.3 Function Overloading

Use overloaded functions (including constructors) only if a reader looking at a call site can get a good idea of what is happening without having to first figure out exactly which overload is being called.

6.3.1 Definition

You may write a function that takes a const `std::string&` and overload it with another that takes `const char*`. However, in this case consider `std::string_view` instead.

```
class MyClass {
 public:
  void Analyze(const std::string &text);
  void Analyze(const char *text, size_t textlen);
};
```

6.3.2 Pros

Overloading can make code more intuitive by allowing an identically-named function to take different arguments. It may be necessary for templatized code, and it can be convenient for Visitors.

Overloading based on const or ref qualification may make utility code more usable, more efficient, or both. (See TotW #148 for more.)

6.3.3 Cons

If a function is overloaded by the argument types alone, a reader may have to understand C++'s complex matching rules in order to tell what's going on. Also many people are confused by the semantics of inheritance if a derived class overrides only some of the variants of a function.

6.3.4 Decision

You may overload a function when there are no semantic differences between variants. These overloads may vary in types, qualifiers, or argument count. However, a reader of such a call must not need to know which member of the overload set is chosen, only that **something** from the set is being called. If you can document all entries in the overload set with a single comment in the header, that is a good sign that it is a well-designed overload set.

6.4 Default Arguments

Default arguments are allowed on non-virtual functions when the default is guaranteed to always have the same value. Follow the same restrictions as for function overloading, and prefer overloaded functions if the readability gained with default arguments doesn't outweigh the downsides below.

6.4.1 Pros

Often you have a function that uses default values, but occasionally you want to override the defaults. Default parameters allow an easy way to do this without having to define many functions for the rare exceptions. Compared to overloading the function, default arguments have a cleaner syntax, with less boilerplate and a clearer distinction between 'required' and 'optional' arguments.

6.4.2 Cons

Defaulted arguments are another way to achieve the semantics of overloaded functions, so all the reasons not to overload functions apply.

The defaults for arguments in a virtual function call are determined by the static type of the target object, and there's no guarantee that all overrides of a given function declare the same defaults.

Default parameters are re-evaluated at each call site, which can bloat the generated code. Readers may also expect the default's value to be fixed at the declaration instead of varying at each call.

Function pointers are confusing in the presence of default arguments, since the function signature often doesn't match the call signature. Adding function overloads avoids these problems.

6.4.3 Decision

Default arguments are banned on virtual functions, where they don't work properly, and in cases where the specified default might not evaluate to the same value depending on when it was evaluated. (For example, don't write `void f(int n = counter++);`.)

In some other cases, default arguments can improve the readability of their function declarations enough to overcome the downsides above, so they are allowed. When in doubt, use overloads.

6.5 Trailing Return Type Syntax

Use trailing return types only where using the ordinary syntax (leading return types) is impractical or much less readable.

6.5.1 Definition

C++ allows two different forms of function declarations. In the older form, the return type appears before the function name. For example:

```
int foo(int x);
```

The newer form uses the `auto` keyword before the function name and a trailing return type after the argument list. For example, the declaration above could equivalently be written:

```
auto foo(int x) -> int;
```

The trailing return type is in the function's scope. This doesn't make a difference for a simple case like `int` but it matters for more complicated cases, like types declared in class scope or types written in terms of the function parameters.

6.5.2 Pros

Trailing return types are the only way to explicitly specify the return type of a lambda expression. In some cases the compiler is able to deduce a lambda's return type, but not in all cases. Even when the compiler can deduce it automatically, sometimes specifying it explicitly would be clearer for readers.

Sometimes it's easier and more readable to specify a return type after the function's parameter list has already appeared. This is particularly true when the return type depends on template parameters. For example:

```
template <typename T, typename U>
auto add(T t, U u) -> decltype(t + u);
```

versus

```
template <typename T, typename U>
decltype(declval<T&>() + declval<U&>()) add(T t, U u);
```

6.5.3 Cons

Trailing return type syntax is relatively new and it has no analogue in C++-like languages such as C and Java, so some readers may find it unfamiliar.

Existing code bases have an enormous number of function declarations that aren't going to get changed to use the new syntax, so the realistic choices are using the old syntax only or using a mixture of the two. Using a single version is better for uniformity of style.

6.5.4 Decision

In most cases, continue to use the older style of function declaration where the return type goes before the function name. Use the new trailing-return-type form only in cases where it's required (such as lambdas) or where, by putting the type after the function's parameter list, it allows you to write the type in a much more readable way. The latter case should be rare; it's mostly an issue in fairly complicated template code, which is discouraged in most cases.

Chapter 7
Google-Specific Magic

Introduction

- *Ownership and Smart Pointers*
- *cpplint*

There are various tricks and utilities that we use to make C++ code more robust, and various ways we use C++ that may differ from what you see elsewhere.

7.1 Ownership and Smart Pointers

Prefer to have single, fixed owners for dynamically allocated objects. Prefer to transfer ownership with smart pointers.

7.1.1 Definition

“Ownership” is a bookkeeping technique for managing dynamically allocated memory (and other resources). The owner of a dynamically allocated object is an object or function that is responsible for ensuring that it is deleted when no longer needed. Ownership can sometimes be shared, in which case the last owner is typically responsible for deleting it. Even when ownership is not shared, it can be transferred from one piece of code to another.

“Smart” pointers are classes that act like pointers, e.g., by overloading the `*` and `->` operators. Some smart pointer types can be used to automate ownership bookkeeping, to ensure these responsibilities are met. std::unique_ptr is a smart pointer type which expresses exclusive ownership of a dynamically allocated object; the object is deleted when the `std::unique_ptr` goes out of scope. It cannot be copied, but can be moved to represent ownership transfer. std::shared_ptr is a smart pointer type that expresses shared ownership of a dynamically allocated object. `std::shared_ptr` s can be copied; ownership of the object is shared among all copies, and the object is deleted when the last `std::shared_ptr` is destroyed.

7.1.2 Pros

- It’s virtually impossible to manage dynamically allocated memory without some sort of ownership logic.
- Transferring ownership of an object can be cheaper than copying it (if copying it is even possible).
- Transferring ownership can be simpler than ‘borrowing’ a pointer or reference, because it reduces the need to coordinate the lifetime of the object between the two users.
- Smart pointers can improve readability by making ownership logic explicit, self-documenting, and unambiguous.
- Smart pointers can eliminate manual ownership bookkeeping, simplifying the code and ruling out large classes of errors.
- For const objects, shared ownership can be a simple and efficient alternative to deep copying.

7.1.3 Cons

- Ownership must be represented and transferred via pointers (whether smart or plain). Pointer semantics are more complicated than value semantics, especially in APIs: you have to worry not just about ownership, but also aliasing, lifetime, and mutability, among other issues.
- The performance costs of value semantics are often overestimated, so the performance benefits of ownership transfer might not justify the readability and complexity costs.
- APIs that transfer ownership force their clients into a single memory management model.
- Code using smart pointers is less explicit about where the resource releases take place.
- `std::unique_ptr` expresses ownership transfer using move semantics, which are relatively new and may confuse some programmers.
- Shared ownership can be a tempting alternative to careful ownership design, obfuscating the design of a system.
- Shared ownership requires explicit bookkeeping at run-time, which can be costly.
- In some cases (e.g., cyclic references), objects with shared ownership may never be deleted.
- Smart pointers are not perfect substitutes for plain pointers.

7.1.4 Decision

If dynamic allocation is necessary, prefer to keep ownership with the code that allocated it. If other code needs access to the object, consider passing it a copy, or passing a pointer or reference without transferring ownership. Prefer to use `std::unique_ptr` to make ownership transfer explicit. For example:

```
std::unique_ptr<Foo> FooFactory();
void FooConsumer(std::unique_ptr<Foo> ptr);
```

Do not design your code to use shared ownership without a very good reason. One such reason is to avoid expensive copy operations, but you should only do this if the performance benefits are significant, and the underlying object is immutable (i.e., `std::shared_ptr<const Foo>`). If you do use shared ownership, prefer to use `std::shared_ptr`.

Never use `std::auto_ptr`. Instead, use `std::unique_ptr`.

7.2 cpplint

Use `cpplint.py` to detect style errors.

`cpplint.py` is a tool that reads a source file and identifies many style errors. It is not perfect, and has both false positives and false negatives, but it is still a valuable tool.

Some projects have instructions on how to run `cpplint.py` from their project tools. If the project you are contributing to does not, you can download cpplint.py separately.

Chapter 8
Other C++ Features

Introduction

- *Rvalue References*
- *Friends*
- *Exceptions*
- `noexcept`
- *Run-Time Type Information (RTTI)*
- *Casting*
- *Streams*
- *Preincrement and Predecrement*
- *Use of const*
- *Use of constexpr*
- *Integer Types*
- *64-bit Portability*
- *Preprocessor Macros*
- *0 and nullptr/NULL*
- *sizeof*
- *Type Deduction (including* `auto` *)*
- *Class Template Argument Deduction*
- *Designated Initializers*
- *Lambda Expressions*
- *Template Metaprogramming*
- *Boost*
- *Other C++ Features*
- *Nonstandard Extensions*
- *Aliases*

8.1 Rvalue References

Use rvalue references only in certain special cases listed below.

8.1.1 Definition

Rvalue references are a type of reference that can only bind to temporary objects. The syntax is similar to traditional reference syntax. For example, `void f(std::string&& s);` declares a function whose argument is an rvalue reference to a `std::string`.

When the token `&&` is applied to an unqualified template argument in a function parameter, special template argument deduction rules apply. Such a reference is called forwarding reference.

8.1.2 Pros

- Defining a move constructor (a constructor taking an rvalue reference to the class type) makes it possible to move a value instead of copying it. If `v1` is a `std::vector<std::string>`, for example, then `auto v2(std::move(v1))` will probably just result in some simple pointer manipulation instead of copying a large amount of data. In many cases this can result in a major performance improvement.
- Rvalue references make it possible to implement types that are movable but not copyable, which can be useful for types that have no sensible definition of copying but where you might still want to pass them as function arguments, put them in containers, etc.
- `std::move` is necessary to make effective use of some standard-library types, such as `sstd::unique_ptr`.
- Forwarding references which use the rvalue reference token, make it possible to write a generic function wrapper that forwards its arguments to another function, and works whether or not its arguments are

temporary objects and/or const. This is called “perfect forwarding”.

8.1.3 Cons

- Rvalue references are not yet widely understood. Rules like reference collapsing and the special deduction rule for forwarding references are somewhat obscure.
- Rvalue references are often misused. Using rvalue references is counter-intuitive in signatures where the argument is expected to have a valid specified state after the function call, or where no move operation is performed.

8.1.4 Decision

Do not use rvalue references (or apply the `&&` qualifier to methods), except as follows:

- You may use them to define move constructors and move assignment operators (as described in Copyable and Movable Types).
- You may use them to define `&&`-qualified methods that logically “consume” `*this`, leaving it in an unusable or empty state. Note that this applies only to method qualifiers (which come after the closing parenthesis of the function signature); if you want to “consume” an ordinary function parameter, prefer to pass it by value.
- You may use forwarding references in conjunction with `std::forward`, to support perfect forwarding.
- You may use them to define pairs of overloads, such as one taking `Foo&&` and the other taking `const Foo&`. Usually the preferred solution is just to pass by value, but an overloaded pair of functions sometimes yields better performance and is sometimes necessary in generic code that needs to support a wide variety of types. As always: if you’re writing more complicated code for the sake of performance, make sure you have evidence that it actually helps.

8.2 Friends

We allow use of `friend` classes and functions, within reason.

Friends should usually be defined in the same file so that the reader does not have to look in another file to find uses of the private members of a class. A common use of `friend` is to have a `FooBuilder` class be a friend of `Foo` so that it can construct the inner state of `Foo` correctly, without exposing this state to the world. In some cases it may be useful to make a unittest class a friend of the class it tests.

Friends extend, but do not break, the encapsulation boundary of a class. In some cases this is better than making a member public when you want to give only one other class access to it. However, most classes should interact with other classes solely through their public members.

8.3 Exceptions

We do not use C++ exceptions.

8.3.1 Pros

- Exceptions allow higher levels of an application to decide how to handle “can’t happen” failures in deeply nested functions, without the obscuring and error-prone bookkeeping of error codes.

- Exceptions are used by most other modern languages. Using them in C++ would make it more consistent with Python, Java, and the C++ that others are familiar with.
- Some third-party C++ libraries use exceptions, and turning them off internally makes it harder to integrate with those libraries.
- Exceptions are the only way for a constructor to fail. We can simulate this with a factory function or an `Init()` method, but these require heap allocation or a new "invalid" state, respectively.
- Exceptions are really handy in testing frameworks.

8.3.2 Cons

- When you add a `throw` statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result. For instance, if `f()` calls `g()` calls `h()`, and `h` throws an exception that `f` catches, `g` has to be careful or it may not clean up properly.
- More generally, exceptions make the control flow of programs difficult to evaluate by looking at code: functions may return in places you don't expect. This causes maintainability and debugging difficulties. You can minimize this cost via some rules on how and where exceptions can be used, but at the cost of more that a developer needs to know and understand.
- Exception safety requires both RAII and different coding practices. Lots of supporting machinery is needed to make writing correct exception-safe code easy. Further, to avoid requiring readers to understand the entire call graph, exception-safe code must isolate logic that writes to persistent state into a "commit" phase. This will have both benefits and costs (perhaps where you're forced to obfuscate code to isolate the commit). Allowing exceptions would force us to always pay those costs even when they're not worth it.
- Turning on exceptions adds data to each binary produced, increasing compile time (probably slightly) and possibly increasing address space pressure.
- The availability of exceptions may encourage developers to throw them when they are not appropriate or recover from them when it's not safe to do so. For example, invalid user input should not cause exceptions to be thrown. We would need to make the style guide even longer to document these restrictions!

8.3.3 Decision

On their face, the benefits of using exceptions outweigh the costs, especially in new projects. However, for existing code, the introduction of exceptions has implications on all dependent code. If exceptions can be propagated beyond a new project, it also becomes problematic to integrate the new project into existing exception-free code. Because most existing C++ code at Google is not prepared to deal with exceptions, it is comparatively difficult to adopt new code that generates exceptions.

Given that Google's existing code is not exception-tolerant, the costs of using exceptions are somewhat greater than the costs in a new project. The conversion process would be slow and error-prone. We don't believe that the available alternatives to exceptions, such as error codes and assertions, introduce a significant burden.

Our advice against using exceptions is not predicated on philosophical or moral grounds, but practical ones. Because we'd like to use our open-source projects at Google and it's difficult to do so if those projects use exceptions, we need to advise against exceptions in Google open-source projects as well. Things would probably be different if we had to do it all over again from scratch.

This prohibition also applies to exception handling related features such as `std::exception_ptr` and `std::nested_exception`.

There is an exception to this rule (no pun intended) for Windows code.

8.4 `noexcept`

Specify `noexcept` when it is useful and correct.

8.4.1 Definition

The `noexcept` specifier is used to specify whether a function will throw exceptions or not. If an exception escapes from a function marked noexcept, the program crashes via `std::terminate`.

The `noexcept` operator performs a compile-time check that returns true if an expression is declared to not throw any exceptions.

8.4.2 Pros

- Specifying move constructors as `noexcept` improves performance in some cases, e.g., `std::vector<T>::resize(` moves rather than copies the objects if T's move constructor is `noexcept`.
- Specifying `noexcept` on a function can trigger compiler optimizations in environments where exceptions are enabled, e.g., compiler does not have to generate extra code for stack-unwinding, if it knows that no exceptions can be thrown due to a `noexcept` specifier.

8.4.3 Cons

- In projects following this guide that have exceptions disabled it is hard to ensure that `noexcept` specifiers are correct, and hard to define what correctness even means.
- It's hard, if not impossible, to undo `noexcept` because it eliminates a guarantee that callers may be relying on, in ways that are hard to detect.

8.4.4 Decision

You may use `noexcept` when it is useful for performance if it accurately reflects the intended semantics of your function, i.e., that if an exception is somehow thrown from within the function body then it represents a fatal error. You can assume that `noexcept` on move constructors has a meaningful performance benefit. If you think there is significant performance benefit from specifying `noexcept` on some other function, please discuss it with your project leads.

Prefer unconditional `noexcept` if exceptions are completely disabled (i.e., most Google C++ environments). Otherwise, use conditional `noexcept` specifiers with simple conditions, in ways that evaluate false only in the few cases where the function could potentially throw. The tests might include type traits check on whether the involved operation might throw (e.g., `std::is_nothrow_move_constructible` for move-constructing objects), or on whether allocation can throw (e.g., `absl::default_allocator_is_nothrow` for standard default allocation). Note in many cases the only possible cause for an exception is allocation failure (we believe move constructors should not throw except due to allocation failure), and there are many applications where it's appropriate to treat memory exhaustion as a fatal error rather than an exceptional condition that your program should attempt to recover from. Even for other potential failures you should prioritize interface simplicity over supporting all possible exception throwing scenarios: instead of writing a complicated `noexcept` clause that depends on whether a hash function can throw, for example, simply document that your component doesn't support hash functions throwing and make it unconditionally `noexcept`.

8.5 Run-Time Type Information (RTTI)

Avoid using run-time type information (RTTI).

8.5.1 Definition

RTTI allows a programmer to query the C++ class of an object at run-time. This is done by use of `typeid` or `dynamic_cast`.

8.5.2 Pros

The standard alternatives to RTTI (described below) require modification or redesign of the class hierarchy in question. Sometimes such modifications are infeasible or undesirable, particularly in widely-used or mature code.

RTTI can be useful in some unit tests. For example, it is useful in tests of factory classes where the test has to verify that a newly created object has the expected dynamic type. It is also useful in managing the relationship between objects and their mocks.

RTTI is useful when considering multiple abstract objects. Consider

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
  Derived* that = dynamic_cast<Derived*>(other);
  if (that == nullptr)
    return false;
  ...
}
```

8.5.3 Cons

Querying the type of an object at run-time frequently means a design problem. Needing to know the type of an object at runtime is often an indication that the design of your class hierarchy is flawed.

Undisciplined use of RTTI makes code hard to maintain. It can lead to type-based decision trees or switch statements scattered throughout the code, all of which must be examined when making further changes.

8.5.4 Decision

RTTI has legitimate uses but is prone to abuse, so you must be careful when using it. You may use it freely in unittests, but avoid it when possible in other code. In particular, think twice before using RTTI in new code.

If you find yourself needing to write code that behaves differently based on the class of an object, consider one of the following alternatives to querying the type:

- Virtual methods are the preferred way of executing different code paths depending on a specific subclass type. This puts the work within the object itself.
- If the work belongs outside the object and instead in some processing code, consider a double-dispatch solution, such as the Visitor design pattern. This allows a facility outside the object itself to determine the type of class using the built-in type system.

When the logic of a program guarantees that a given instance of a base class is in fact an instance of a particular derived class, then a `dynamic_cast` may be used freely on the object. Usually one can use a `static_cast` as an alternative in such situations.

Decision trees based on type are a strong indication that your code is on the wrong track.

```
if (typeid(*data) == typeid(D1)) {
  ...
} else if (typeid(*data) == typeid(D2)) {
  ...
} else if (typeid(*data) == typeid(D3)) {
...
```

Code such as this usually breaks when additional subclasses are added to the class hierarchy. Moreover, when properties of a subclass change, it is difficult to find and modify all the affected code segments.

Do not hand-implement an RTTI-like workaround. The arguments against RTTI apply just as much to workarounds like class hierarchies with type tags. Moreover, workarounds disguise your true intent.

8.6 Casting

Use C++-style casts like `static_cast<float>(double_value)`, or brace initialization for conversion of arithmetic types like `int64_t y = int64_t{1} << 42`. Do not use cast formats like `(int)x` unless the cast is to void. You may use cast formats like `T(x)` only when `T` is a class type.

8.6.1 Definition

C++ introduced a different cast system from C that distinguishes the types of cast operations.

8.6.2 Pros

The problem with C casts is the ambiguity of the operation; sometimes you are doing a *conversion* (e.g., `(int)3.5`) and sometimes you are doing a *cast* (e.g., `(int)"hello"`). Brace initialization and C++ casts can often help avoid this ambiguity. Additionally, C++ casts are more visible when searching for them.

8.6.3 Cons

The C++-style cast syntax is verbose and cumbersome.

8.6.4 Decision

In general, do not use C-style casts. Instead, use these C++-style casts when explicit type conversion is necessary.

- Use brace initialization to convert arithmetic types (e.g., `int64_t{x}`). This is the safest approach because code will not compile if conversion can result in information loss. The syntax is also concise.
- Use `absl::implicit_cast` to safely cast up a type hierarchy, e.g., casting a `Foo*` to a `SuperclassOfFoo*` or casting a `Foo*` to a `const Foo*`. C++ usually does this automatically but some situations need an explicit up-cast, such as use of the `?:` operator.
- Use `static_cast` as the equivalent of a C-style cast that does value conversion, when you need to explicitly up-cast a pointer from a class to its superclass, or when you need to explicitly cast a pointer from a superclass to a subclass. In this last case, you must be sure your object is actually an instance of the subclass.
- Use `const_cast` to remove the `const` qualifier (see const).
- Use `reinterpret_cast` to do unsafe conversions of pointer types to and from integer and other pointer types, including `void*`. Use this only if you know what you are doing and you understand the aliasing issues. Also, consider the alternative `absl::bit_cast`.
- Use `absl::bit_cast` to interpret the raw bits of a value using a different type of the same size (a type pun), such as interpreting the bits of a `double` as `int64_t`.

See the RTTI section for guidance on the use of `dynamic_cast`.

8.7 Streams

Use streams where appropriate, and stick to "simple" usages. Overload `<<` for streaming only for types representing values, and write only the user-visible value, not any implementation details.

8.7.1 Definition

Streams are the standard I/O abstraction in C++, as exemplified by the standard header `<iostream>`. They are widely used in Google code, mostly for debug logging and test diagnostics.

8.7.2 Pros

The `<<` and `>>` stream operators provide an API for formatted I/O that is easily learned, portable, reusable, and extensible. `printf`, by contrast, doesn't even support `std::string`, to say nothing of user-defined types, and is very difficult to use portably. `printf` also obliges you to choose among the numerous slightly different versions of that function, and navigate the dozens of conversion specifiers.

Streams provide first-class support for console I/O via `std::cin`, `std::cout`, `std::cerr`, and `std::clog`. The C APIs do as well, but are hampered by the need to manually buffer the input.

8.7.3 Cons

- Stream formatting can be configured by mutating the state of the stream. Such mutations are persistent, so the behavior of your code can be affected by the entire previous history of the stream, unless you go out of your way to restore it to a known state every time other code might have touched it. User code can not only modify the built-in state, it can add new state variables and behaviors through a registration system.

- It is difficult to precisely control stream output, due to the above issues, the way code and data are mixed in streaming code, and the use of operator overloading (which may select a different overload than you expect).
- The practice of building up output through chains of `<<` operators interferes with internationalization, because it bakes word order into the code, and streams' support for localization is flawed.
- The streams API is subtle and complex, so programmers must develop experience with it in order to use it effectively.
- Resolving the many overloads of `<<` is extremely costly for the compiler. When used pervasively in a large code base, it can consume as much as 20% of the parsing and semantic analysis time.

8.7.4 Decision

Use streams only when they are the best tool for the job. This is typically the case when the I/O is ad-hoc, local, human-readable, and targeted at other developers rather than end-users. Be consistent with the code around you, and with the codebase as a whole; if there's an established tool for your problem, use that tool instead. In particular, logging libraries are usually a better choice than `std::cerr` or `std::clog` for diagnostic output, and the libraries in `absl/strings` or the equivalent are usually a better choice than `std::stringstream`.

Avoid using streams for I/O that faces external users or handles untrusted data. Instead, find and use the appropriate templating libraries to handle issues like internationalization, localization, and security hardening.

If you do use streams, avoid the stateful parts of the streams API (other than error state), such as `imbue()`, `xalloc()`, and `register_callback()`. Use explicit formatting functions (such as `absl::StreamFormat()`) rather than stream manipulators or formatting flags to control formatting details such as number base, precision, or padding.

Overload `<<` as a streaming operator for your type only if your type represents a value, and `<<` writes out a human-readable string representation of that value. Avoid exposing implementation details in the output of `<<`; if you need to print object internals for debugging, use named functions instead (a method named `DebugString()` is the most common convention).

8.8 Preincrement and Predecrement

Use the prefix form (`++i`) of the increment and decrement operators unless you need postfix semantics.

8.8.1 Definition

When a variable is incremented (`++i` or `i++`) or decremented (`--i` or `i--`) and the value of the expression is not used, one must decide whether to preincrement (decrement) or postincrement (decrement).

8.8.2 Pros

A postfix increment/decrement expression evaluates to the value *as it was before it was modified*. This can result in code that is more compact but harder to read. The prefix form is generally more readable, is never less efficient, and can be more efficient because it doesn't need to make a copy of the value as it was before the operation.

8.8.3 Cons

The tradition developed, in C, of using post-increment, even when the expression value is not used, especially in `for` loops.

8.8.4 Decision

Use prefix increment/decrement, unless the code explicitly needs the result of the postfix increment/decrement expression.

8.9 Use of const

In APIs, use `const` whenever it makes sense. `constexpr` is a better choice for some uses of const.

8.9.1 Definition

Declared variables and parameters can be preceded by the keyword `const` to indicate the variables are not changed (e.g., `const int foo`). Class functions can have the `const` qualifier to indicate the function does not change the state of the class member variables (e.g., `class Foo { int Bar(char c) const; };`).

8.9.2 Pros

Easier for people to understand how variables are being used. Allows the compiler to do better type checking, and, conceivably, generate better code. Helps people convince themselves of program correctness because they know the functions they call are limited in how they can modify your variables. Helps people know what functions are safe to use without locks in multi-threaded programs.

8.9.3 Cons

`const` is viral: if you pass a `const` variable to a function, that function must have `const` in its prototype (or the variable will need a `const_cast`). This can be a particular problem when calling library functions.

8.9.4 Decision

We strongly recommend using `const` in APIs (i.e., on function parameters, methods, and non-local variables) wherever it is meaningful and accurate. This provides consistent, mostly compiler-verified documentation of what objects an operation can mutate. Having a consistent and reliable way to distinguish reads from writes is critical to writing thread-safe code, and is useful in many other contexts as well. In particular:

- If a function guarantees that it will not modify an argument passed by reference or by pointer, the corresponding function parameter should be a reference-to-const (`const T&`) or pointer-to-const (`const T*`), respectively.
- For a function parameter passed by value, `const` has no effect on the caller, thus is not recommended in function declarations. See TotW #109.
- Declare methods to be `const` unless they alter the logical state of the object (or enable the user to modify that state, e.g., by returning a non-const reference, but that's rare), or they can't safely be invoked concurrently.

Using `const` on local variables is neither encouraged nor discouraged.

All of a class's `const` operations should be safe to invoke concurrently with each other. If that's not feasible, the class must be clearly documented as "thread-unsafe".

8.9.4.1 Where to put the `const`

Some people favor the form `int const *foo` to `const int* foo`. They argue that this is more readable because it's more consistent: it keeps the rule that `const` always follows the object it's describing. However, this consistency argument doesn't apply in codebases with few deeply-nested pointer expressions since most `const` expressions have only one `const`, and it applies to the underlying value. In such cases, there's no consistency to maintain. Putting the `const` first is arguably more readable, since it follows English in putting the "adjective" (`const`) before the "noun" (`int`).

That said, while we encourage putting `const` first, we do not require it. But be consistent with the code around you!

8.10 Use of constexpr

Use `constexpr` to define true constants or to ensure constant initialization.

8.10.1 Definition

Some variables can be declared `constexpr` to indicate the variables are true constants, i.e., fixed at compilation/link time. Some functions and constructors can be declared `constexpr` which enables them to be used in defining a `constexpr` variable.

8.10.2 Pros

Use of `constexpr` enables definition of constants with floating-point expressions rather than just literals; definition of constants of user-defined types; and definition of constants with function calls.

8.10.3 Cons

Prematurely marking something as `constexpr` may cause migration problems if later on it has to be downgraded. Current restrictions on what is allowed in `constexpr` functions and constructors may invite obscure workarounds in these definitions.

8.10.4 Decision

`constexpr` definitions enable a more robust specification of the constant parts of an interface. Use `constexpr` to specify true constants and the functions that support their definitions. Avoid complexifying function definitions to enable their use with `constexpr`. Do not use `constexpr` to force inlining.

8.11 Integer Types

Of the built-in C++ integer types, the only one used is `int`. If a program needs a variable of a different size, use a precise-width integer type from `<cstdint>`, such as `int16_t`. If your variable represents a value that could ever be greater than or equal to 2^{31} (2GiB), use a 64-bit type such as `int64_t`. Keep in mind that even if your value won't ever be too large for an `int`, it may be used in intermediate calculations which may require a larger type. When in doubt, choose a larger type.

8.11.1 Definition

C++ does not precisely specify the sizes of integer types like `int`. Typically people assume that `short` is 16 bits, `int` is 32 bits, `long` is 32 bits and `long long` is 64 bits.

8.11.2 Pros

Uniformity of declaration.

8.11.3 Cons

The sizes of integral types in C++ can vary based on compiler and architecture.

8.11.4 Decision

The standard library header `<cstdint>` defines types like `int16_t`, `uint32_t`, `int64_t`, etc. You should always use those in preference to `short`, `unsigned long long` and the like, when you need a guarantee on the size of an integer. Of the C integer types, only `int` should be used. When appropriate, you are welcome to use standard types like `size_t` and `ptrdiff_t`.

We use `int` very often, for integers we know are not going to be too big, e.g., loop counters. Use plain old `int` for such things. You should assume that an `int` is at least 32 bits, but don't assume that it has more than 32 bits. If you need a 64-bit integer type, use `int64_t` or `uint64_t`.

For integers we know can be "big", use `int64_t`.

You should not use the unsigned integer types such as `uint32_t`, unless there is a valid reason such as representing a bit pattern rather than a number, or you need defined overflow modulo 2^N. In particular, do not use unsigned types to say a number will never be negative. Instead, use assertions for this.

If your code is a container that returns a size, be sure to use a type that will accommodate any possible usage of your container. When in doubt, use a larger type rather than a smaller type.

Use care when converting integer types. Integer conversions and promotions can cause undefined behavior, leading to security bugs and other problems.

8.11.4.1 On Unsigned Integers

Unsigned integers are good for representing bitfields and modular arithmetic. Because of historical accident, the C++ standard also uses unsigned integers to represent the size of containers - many members of the standards body believe this to be a mistake, but it is effectively impossible to fix at this point. The fact that unsigned arithmetic doesn't model the behavior of a simple integer, but is instead defined by the standard to model modular arithmetic (wrapping around on overflow/underflow), means that a significant class of bugs cannot be diagnosed by the compiler. In other cases, the defined behavior impedes optimization.

That said, mixing signedness of integer types is responsible for an equally large class of problems. The best advice we can provide:

- Try to use iterators and containers rather than pointers and sizes.

- Try not to mix signedness.
- Try to avoid unsigned types (except for representing bitfields or modular arithmetic).
- Do not use an unsigned type merely to assert that a variable is non-negative.

8.12 64-bit Portability

Code should be 64-bit and 32-bit friendly. Bear in mind problems of printing, comparisons, and structure alignment.

- Correct portable `printf()` conversion specifiers for some integral typedefs rely on macro expansions that we find unpleasant to use and impractical to require (the `PRI` macros from `<cinttypes>`). Unless there is no reasonable alternative for your particular case, try to avoid or even upgrade APIs that rely on the `printf` family. Instead use a library supporting typesafe numeric formatting, such as `StrCat` or `Substitute` for fast simple conversions, or std::ostream.
  Unfortunately, the `PRI` macros are the only portable way to specify a conversion for the standard bitwidth typedefs (e.g., `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, etc). Where possible, avoid passing arguments of types specified by bitwidth typedefs to `printf`-based APIs. Note that it is acceptable to use typedefs for which `printf` has dedicated length modifiers, such as `size_t(z)`, `ptrdiff_t(t)`, and `maxint_t(j)`.
- Remember that `sizeof(void *) != sizeof(int)`. Use `intptr_t` if you want a pointer-sized integer.
- You may need to be careful with structure alignments, particularly for structures being stored on disk. Any class/structure with a `int64_t`/`uint64_t` member will by default end up being 8-byte aligned on a 64-bit system. If you have such structures being shared on disk between 32-bit and 64-bit code, you will need to ensure that they are packed the same on both architectures. Most compilers offer a way to alter structure alignment. For gcc, you can use `__attribute__((packed))`. MSVC offers `#pragma pack()` and `__declspec(align())`.
- Use braced-initialization as needed to create 64-bit constants. For example:

```
int64_t my_value{0x123456789};
uint64_t my_mask{uint64_t{3} << 48};
```

8.13 Preprocessor Macros

Avoid defining macros, especially in headers; prefer inline functions, enums, and `const` variables. Name macros with a project-specific prefix. Do not use macros to define pieces of a C++ API.

Macros mean that the code you see is not the same as the code the compiler sees. This can introduce unexpected behavior, especially since macros have global scope.

The problems introduced by macros are especially severe when they are used to define pieces of a C++ API, and still more so for public APIs. Every error message from the compiler when developers incorrectly use that interface now must explain how the macros formed the interface. Refactoring and analysis tools have a dramatically harder time updating the interface. As a consequence, we specifically disallow using macros in this way. For example, avoid patterns like:

```
class WOMBAT_TYPE(Foo) {
  // ...

 public:
  EXPAND_PUBLIC_WOMBAT_API(Foo)

  EXPAND_WOMBAT_COMPARISONS(Foo, ==, <)
};
```

Luckily, macros are not nearly as necessary in C++ as they are in C. Instead of using a macro to inline performance-critical code, use an inline function. Instead of using a macro to store a constant, use a `const` variable. Instead of using a macro to "abbreviate" a long variable name, use a reference. Instead of using a macro to conditionally compile code ... well, don't do that at all (except, of course, for the `#define` guards to prevent double inclusion of header files). It makes testing much more difficult.

Macros can do things these other techniques cannot, and you do see them in the codebase, especially in the lower-level libraries. And some of their special features (like stringifying, concatenation, and so forth) are not available through the language proper. But before using a macro, consider carefully whether there's a non-macro way to achieve the same result. If you need to use a macro to define an interface, contact your project leads to request a waiver of this rule.

The following usage pattern will avoid many problems with macros; if you use macros, follow it whenever possible:

- Don't define macros in a `.h` file.
- `#define` macros right before you use them, and `#undef` them right after.
- Do not just `#undef` an existing macro before replacing it with your own; instead, pick a name that's likely to be unique.
- Try not to use macros that expand to unbalanced C++ constructs, or at least document that behavior well.
- Prefer not using `##` to generate function/class/variable names.

Exporting macros from headers (i.e., defining them in a header without `#undef`ing them before the end of the header) is extremely strongly discouraged. If you do export a macro from a header, it must have a globally unique name. To achieve this, it must be named with a prefix consisting of your project's namespace name (but upper case).

8.14 0 and nullptr/NULL

Use `nullptr` for pointers, and `'\0'` for chars (and not the `0` literal).

For pointers (address values), use `nullptr`, as this provides type-safety.

Use `'\0'` for the null character. Using the correct type makes the code more readable.

8.15 sizeof

Prefer `sizeof(varname)` to `sizeof(type)`.

Use `sizeof(varname)` when you take the size of a particular variable. `sizeof(varname)` will update appropriately if someone changes the variable type either now or later. You may use `sizeof(type)` for code unrelated to any particular variable, such as code that manages an external or internal data format where a variable of an appropriate C++ type is not convenient.

```
MyStruct data;
memset(&data, 0, sizeof(data));
```

```
memset(&data, 0, sizeof(MyStruct));
```

```
if (raw_size < sizeof(int)) {
  LOG(ERROR) << "compressed record not big enough for count: " << raw_size;
  return false;
}
```

8.16 Type Deduction (including `auto`)

Use type deduction only if it makes the code clearer to readers who aren't familiar with the project, or if it makes the code safer. Do not use it merely to avoid the inconvenience of writing an explicit type.

8.16.1 Definition

here are several contexts in which C++ allows (or even requires) types to be deduced by the compiler, rather than spelled out explicitly in the code:

8.16.1.1 Function template argument deduction

A function template can be invoked without explicit template arguments. The compiler deduces those arguments from the types of the function arguments:

```
template <typename T>
void f(T t);

f(0);  // Invokes f<int>(0)
```

8.16.1.2 auto variable declarations

A variable declaration can use the `auto` keyword in place of the type. The compiler deduces the type from the variable's initializer, following the same rules as function template argument deduction with the same initializer (so long as you don't use curly braces instead of parentheses).

```
auto a = 42;  // a is an int
auto& b = a;  // b is an int&
auto c = b;   // c is an int
auto d{42};   // d is an int, not a std::initializer_list<int>
```

`auto` can be qualified with `const`, and can be used as part of a pointer or reference type, but it can't be used as a template argument. A rare variant of this syntax uses `decltype(auto)` instead of `auto`, in which case the deduced type is the result of applying decltype to the initializer.

8.16.1.3 Function return type deduction

`auto` (and `decltype(auto)`) can also be used in place of a function return type. The compiler deduces the return type from the `return` statements in the function body, following the same rules as for variable declarations:

```
auto f() { return 0; }  // The return type of f is int
```

Lambda expression return types can be deduced in the same way, but this is triggered by omitting the return type, rather than by an explicit `auto`. Confusingly, trailing return type syntax for functions also uses `auto` in the return-type position, but that doesn't rely on type deduction; it's just an alternate syntax for an explicit return type.

8.16.1.4 Generic lambdas

A lambda expression can use the `auto` keyword in place of one or more of its parameter types. This causes the lambda's call operator to be a function template instead of an ordinary function, with a separate template parameter for each `auto` function parameter:

```
// Sort `vec` in decreasing order
std::sort(vec.begin(), vec.end(), [](auto lhs, auto rhs) { return lhs > rhs; });
```

8.16.1.5 Lambda init captures

Lambda captures can have explicit initializers, which can be used to declare wholly new variables rather than only capturing existing ones:

```
[x = 42, y = "foo"] { ... }  // x is an int, and y is a const char*
```

This syntax doesn't allow the type to be specified; instead, it's deduced using the rules for `auto` variables.

8.16.1.6 Class template argument deduction

See below.

8.16.1.7 Structured bindings

When declaring a tuple, struct, or array using `auto`, you can specify names for the individual elements instead of a name for the whole object; these names are called "structured bindings", and the whole declaration is called a "structured binding declaration". This syntax provides no way of specifying the type of either the enclosing object or the individual names:

```
auto [iter, success] = my_map.insert({key, value});
if (!success) {
iter->second = value;
}
```

The `auto` can also be qualified with `const`, `&`, and `&&`, but note that these qualifiers technically apply to the anonymous tuple/struct/array, rather than the individual bindings. The rules that determine the types of the bindings are quite complex; the results tend to be unsurprising, except that the binding types typically won't be references even if the declaration declares a reference (but they will usually behave like references anyway).

(These summaries omit many details and caveats; see the links for further information.)

8.16.2 Pros

- C++ type names can be long and cumbersome, especially when they involve templates or namespaces.
- When a C++ type name is repeated within a single declaration or a small code region, the repetition may not be aiding readability.
- It is sometimes safer to let the type be deduced, since that avoids the possibility of unintended copies or type conversions.

8.16.3 Cons

C++ code is usually clearer when types are explicit, especially when type deduction would depend on information from distant parts of the code. In expressions like:

```
auto foo = x.add_foo();
auto i = y.Find(key);
```

it may not be obvious what the resulting types are if the type of `y` isn't very well known, or if `y` was declared many lines earlier.

Programmers have to understand when type deduction will or won't produce a reference type, or they'll get copies when they didn't mean to.

If a deduced type is used as part of an interface, then a programmer might change its type while only intending to change its value, leading to a more radical API change than intended.

8.16.4 Decision

The fundamental rule is: use type deduction only to make the code clearer or safer, and do not use it merely to avoid the inconvenience of writing an explicit type. When judging whether the code is clearer, keep in mind that your readers are not necessarily on your team, or familiar with your project, so types that you and your reviewer experience as unnecessary clutter will very often provide useful information to others. For example, you can assume that the return type of `make_unique<Foo>()` is obvious, but the return type of `MyWidgetFactory()` probably isn't.

These principles apply to all forms of type deduction, but the details vary, as described in the following sections.

8.16.4.1 Function template argument deduction

Function template argument deduction is almost always OK. Type deduction is the expected default way of interacting with function templates, because it allows function templates to act like infinite sets of ordinary function overloads. Consequently, function templates are almost always designed so that template argument deduction is clear and safe, or doesn't compile.

8.16.4.2 Local variable type deduction

For local variables, you can use type deduction to make the code clearer by eliminating type information that is obvious or irrelevant, so that the reader can focus on the meaningful parts of the code:

```
std::unique_ptr<WidgetWithBellsAndWhistles> widget_ptr =
std::make_unique<WidgetWithBellsAndWhistles>(arg1, arg2);
absl::flat_hash_map<std::string,
std::unique_ptr<WidgetWithBellsAndWhistles>>::const_iterator
it = my_map_.find(key);
std::array<int, 6> numbers = {4, 8, 15, 16, 23, 42};
```

```
auto widget_ptr = std::make_unique<WidgetWithBellsAndWhistles>(arg1, arg2);
auto it = my_map_.find(key);
std::array numbers = {4, 8, 15, 16, 23, 42};
```

Types sometimes contain a mixture of useful information and boilerplate, such as `it` in the example above: it's obvious that the type is an iterator, and in many contexts the container type and even the key type aren't relevant, but the type of the values is probably useful. In such situations, it's often possible to define local variables with explicit types that convey the relevant information:

```
if (auto it = my_map_.find(key); it != my_map_.end()) {
WidgetWithBellsAndWhistles& widget = *it->second;
// Do stuff with `widget`
}
```

If the type is a template instance, and the parameters are boilerplate but the template itself is informative, you can use class template argument deduction to suppress the boilerplate. However, cases where this actually provides a meaningful benefit are quite rare. Note that class template argument deduction is also subject to a separate style rule.

Do not use `decltype(auto)` if a simpler option will work, because it's a fairly obscure feature, so it has a high cost in code clarity.

8.16.4.3 Return type deduction

Use return type deduction (for both functions and lambdas) only if the function body has a very small number of return statements, and very little other code, because otherwise the reader may not be able to tell at a glance what the `return` type is. Furthermore, use it only if the function or lambda has a very narrow scope, because functions with deduced return types don't define abstraction boundaries: the implementation is the interface. In particular, public functions in header files should almost never have deduced return types.

8.16.4.4 Parameter type deduction

`auto` parameter types for lambdas should be used with caution, because the actual type is determined by the code that calls the lambda, rather than by the definition of the lambda. Consequently, an explicit type will almost always be clearer unless the lambda is explicitly called very close to where it's defined (so that the reader can easily see both), or the lambda is passed to an interface so well-known that it's obvious what arguments it will eventually be called with (e.g., the `std::sort` example above).

8.16.4.5 Lambda init captures

Init captures are covered by a more specific style rule, which largely supersedes the general rules for type deduction.

8.16.5 Structured bindings

Unlike other forms of type deduction, structured bindings can actually give the reader additional information, by giving meaningful names to the elements of a larger object. This means that a structured binding declaration may provide a net readability improvement over an explicit type, even in cases where `auto` would not. Structured bindings are especially beneficial when the object is a pair or tuple (as in the `insert` example above), because they don't have meaningful field names to begin with, but note that you generally shouldn't use pairs or tuples unless a pre-existing API like `insert` forces you to.

If the object being bound is a struct, it may sometimes be helpful to provide names that are more specific to your usage, but keep in mind that this may also mean the names are less recognizable to your reader than the field names. We recommend using a comment to indicate the name of the underlying field, if it doesn't match the name of the binding, using the same syntax as for function parameter comments:

```
auto [/*field_name1=*/bound_name1, /*field_name2=*/bound_name2] = ...
```

As with function parameter comments, this can enable tools to detect if you get the order of the fields wrong.

8.17 Class Template Argument Deduction

Use class template argument deduction only with templates that have explicitly opted into supporting it.

8.17.1 Definition

Class template argument deduction (often abbreviated "CTAD") occurs when a variable is declared with a type that names a template, and the template argument list is not provided (not even empty angle brackets):

```
std::array a = {1, 2, 3};  // `a` is a std::array<int, 3>
```

The compiler deduces the arguments from the initializer using the template's "deduction guides", which can be explicit or implicit.

Explicit deduction guides look like function declarations with trailing return types, except that there's no leading `auto`, and the function name is the name of the template. For example, the above example relies on this deduction guide for `std::array`:

```
namespace std {
template <class T, class... U>
array(T, U...) -> std::array<T, 1 + sizeof...(U)>;
}
```

Constructors in a primary template (as opposed to a template specialization) also implicitly define deduction guides.

When you declare a variable that relies on CTAD, the compiler selects a deduction guide using the rules of constructor overload resolution, and that guide's return type becomes the type of the variable.

8.17.2 Pros

CTAD can sometimes allow you to omit boilerplate from your code.

8.17.3 Cons

The implicit deduction guides that are generated from constructors may have undesirable behavior, or be outright incorrect. This is particularly problematic for constructors written before CTAD was introduced in

C++17, because the authors of those constructors had no way of knowing about (much less fixing) any problems that their constructors would cause for CTAD. Furthermore, adding explicit deduction guides to fix those problems might break any existing code that relies on the implicit deduction guides.

CTAD also suffers from many of the same drawbacks as `auto`, because they are both mechanisms for deducing all or part of a variable's type from its initializer. CTAD does give the reader more information than `auto`, but it also doesn't give the reader an obvious cue that information has been omitted.

8.17.4 Decision

Do not use CTAD with a given template unless the template's maintainers have opted into supporting use of CTAD by providing at least one explicit deduction guide (all templates in the `std` namespace are also presumed to have opted in). This should be enforced with a compiler warning if available.

Uses of CTAD must also follow the general rules on Type deduction.

8.18 Designated Initializers

Use designated initializers only in their C++20-compliant form.

8.18.1 Definition

Designated initializers are a syntax that allows for initializing an aggregate ("plain old struct") by naming its fields explicitly:

```
struct Point {
float x = 0.0;
float y = 0.0;
float z = 0.0;
};

Point p = {
.x = 1.0,
.y = 2.0,
// z will be 0.0
};
```

The explicitly listed fields will be initialized as specified, and others will be initialized in the same way they would be in a traditional aggregate initialization expression like `Point{1.0, 2.0}`.

8.18.2 Pros

Designated initializers can make for convenient and highly readable aggregate expressions, especially for structs with less straightforward ordering of fields than the `Point` example above.

8.18.3 Cons

While designated initializers have long been part of the C standard and supported by C++ compilers as an extension, only recently have they made it into the C++ standard, being added as part of C++20.

The rules in the C++ standard are stricter than in C and compiler extensions, requiring that the designated initializers appear in the same order as the fields appear in the struct definition. So in the example above, it is legal according to C++20 to initialize `x` and then `z`, but not `y` and then `x`.

8.18.4 Decision

Use designated initializers only in the form that is compatible with the C++20 standard: with initializers in the same order as the corresponding fields appear in the struct definition.

8.19 Lambda Expressions

Use lambda expressions where appropriate. Prefer explicit captures when the lambda will escape the current scope.

8.19.1 Definition

Lambda expressions are a concise way of creating anonymous function objects. They're often useful when passing functions as arguments. For example:

```
std::sort(v.begin(), v.end(), [](int x, int y) {
return Weight(x) < Weight(y);
});
```

They further allow capturing variables from the enclosing scope either explicitly by name, or implicitly using a default capture. Explicit captures require each variable to be listed, as either a value or reference capture:

```
int weight = 3;
int sum = 0;
// Captures `weight` by value and `sum` by reference.
std::for_each(v.begin(), v.end(), [weight, &sum](int x) {
sum += weight * x;
});
```

Default captures implicitly capture any variable referenced in the lambda body, including `this` if any members are used:

```
const std::vector<int> lookup_table = ...;
std::vector<int> indices = ...;
// Captures `lookup_table` by reference, sorts `indices` by the value
// of the associated element in `lookup_table`.
```

```
std::sort(indices.begin(), indices.end(), [&](int a, int b) {
return lookup_table[a] < lookup_table[b];
});
```

A variable capture can also have an explicit initializer, which can be used for capturing move-only variables by value, or for other situations not handled by ordinary reference or value captures:

```
std::unique_ptr<Foo> foo = ...;
[foo = std::move(foo)] () {
...
}
```

Such captures (often called "init captures" or "generalized lambda captures") need not actually "capture" anything from the enclosing scope, or even have a name from the enclosing scope; this syntax is a fully general way to define members of a lambda object:

```
[foo = std::vector<int>({1, 2, 3})] () {
...
}
```

The type of a capture with an initializer is deduced using the same rules as `auto`.

8.19.2 Pros

- Lambdas are much more concise than other ways of defining function objects to be passed to STL algorithms, which can be a readability improvement.
- Appropriate use of default captures can remove redundancy and highlight important exceptions from the default.
- Lambdas, `std::function`, and `std::bind` can be used in combination as a general purpose callback mechanism; they make it easy to write functions that take bound functions as arguments.

8.19.3 Cons

- Variable capture in lambdas can be a source of dangling-pointer bugs, particularly if a lambda escapes the current scope.
- Default captures by value can be misleading because they do not prevent dangling-pointer bugs. Capturing a pointer by value doesn't cause a deep copy, so it often has the same lifetime issues as capture by reference. This is especially confusing when capturing `this` by value, since the use of `this` is often implicit.
- Captures actually declare new variables (whether or not the captures have initializers), but they look nothing like any other variable declaration syntax in C++. In particular, there's no place for the variable's type, or even an `auto` placeholder (although init captures can indicate it indirectly, e.g., with a cast). This can make it difficult to even recognize them as declarations.

- Init captures inherently rely on type deduction, and suffer from many of the same drawbacks as `auto`, with the additional problem that the syntax doesn't even cue the reader that deduction is taking place.
- It's possible for use of lambdas to get out of hand; very long nested anonymous functions can make code harder to understand.

8.19.4 Decision

- Use lambda expressions where appropriate, with formatting as described below.
- Prefer explicit captures if the lambda may escape the current scope. For example, instead of:

```
{
  Foo foo;
  ...
  executor->Schedule([&] { Frobnicate(foo); })
  ...
}
// BAD! The fact that the lambda makes use of a reference to `foo` and
// possibly `this` (if `Frobnicate` is a member function) may not be
// apparent on a cursory inspection. If the lambda is invoked after
// the function returns, that would be bad, because both `foo`
// and the enclosing object could have been destroyed.
```

  prefer to write:

```
{
  Foo foo;
  ...
  executor->Schedule([&foo] { Frobnicate(foo); })
  ...
}
// BETTER - The compile will fail if `Frobnicate` is a member
// function, and it's clearer that `foo` is dangerously captured by
// reference.
```

- Use default capture by reference (`[&]`) only when the lifetime of the lambda is obviously shorter than any potential captures.
- Use default capture by value (`[=]`) only as a means of binding a few variables for a short lambda, where the set of captured variables is obvious at a glance, and which does not result in capturing `this` implicitly. (That means that a lambda that appears in a non-static class member function and refers to non-static class members in its body must capture `this` explicitly or via `[&]` .) Prefer not to write long or complex lambdas with default capture by value.
- Use captures only to actually capture variables from the enclosing scope. Do not use captures with initializers to introduce new names, or to substantially change the meaning of an existing name. Instead,

declare a new variable in the conventional way and then capture it, or avoid the lambda shorthand and define a function object explicitly.

- See the section on type deduction for guidance on specifying the parameter and return types.

8.20 Template Metaprogramming

Avoid complicated template programming.

8.20.1 Definition

Template metaprogramming refers to a family of techniques that exploit the fact that the C++ template instantiation mechanism is Turing complete and can be used to perform arbitrary compile-time computation in the type domain.

8.20.2 Pros

Template metaprogramming allows extremely flexible interfaces that are type safe and high performance. Facilities like GoogleTest, `std::tuple`, `std::function`, and `Boost.Spirit` would be impossible without it.

8.20.3 Cons

The techniques used in template metaprogramming are often obscure to anyone but language experts. Code that uses templates in complicated ways is often unreadable, and is hard to debug or maintain.

Template metaprogramming often leads to extremely poor compile time error messages: even if an interface is simple, the complicated implementation details become visible when the user does something wrong.

Template metaprogramming interferes with large scale refactoring by making the job of refactoring tools harder. First, the template code is expanded in multiple contexts, and it's hard to verify that the transformation makes sense in all of them. Second, some refactoring tools work with an AST that only represents the structure of the code after template expansion. It can be difficult to automatically work back to the original source construct that needs to be rewritten.

8.20.4 Decision

Template metaprogramming sometimes allows cleaner and easier-to-use interfaces than would be possible without it, but it's also often a temptation to be overly clever. It's best used in a small number of low level components where the extra maintenance burden is spread out over a large number of uses.

Think twice before using template metaprogramming or other complicated template techniques; think about whether the average member of your team will be able to understand your code well enough to maintain it after you switch to another project, or whether a non-C++ programmer or someone casually browsing the code base will be able to understand the error messages or trace the flow of a function they want to call. If you're using recursive template instantiations or type lists or metafunctions or expression templates, or relying on SFINAE or on the `sizeof` trick for detecting function overload resolution, then there's a good chance you've gone too far.

If you use template metaprogramming, you should expect to put considerable effort into minimizing and isolating the complexity. You should hide metaprogramming as an implementation detail whenever possible, so that user-facing headers are readable, and you should make sure that tricky code is especially well commented. You should carefully document how the code is used, and you should say something about what the "generated" code looks like. Pay extra attention to the error messages that the compiler emits when users make mistakes. The error messages are part of your user interface, and your code should be tweaked as necessary so that the error messages are understandable and actionable from a user point of view.

8.21 Boost

Use only approved libraries from the Boost library collection.

8.21.1 Definition

The Boost library collection is a popular collection of peer-reviewed, free, open-source C++ libraries.

8.21.2 Pros

Boost code is generally very high-quality, is widely portable, and fills many important gaps in the C++ standard library, such as type traits and better binders.

8.21.3 Cons

Some Boost libraries encourage coding practices which can hamper readability, such as metaprogramming and other advanced template techniques, and an excessively "functional" style of programming.

8.21.4 Decision

In order to maintain a high level of readability for all contributors who might read and maintain code, we only allow an approved subset of Boost features. Currently, the following libraries are permitted:

- Call Traits from `boost/call_traits.hpp`
- Compressed Pair from `boost/compressed_pair.hpp`
- The Boost Graph Library (BGL) from `boost/graph`, except serialization (`adj_list_serialize.hpp`) and parallel/distributed algorithms and data structures (`boost/graph/parallel/*` and `boost/graph/distribut`
- Property Map from `boost/property_map`, except parallel/distributed property maps (`boost/property_map/para`
- Iterator from `boost/iterator`
- The part of Polygon that deals with Voronoi diagram construction and doesn't depend on the rest of Polygon: `boost/polygon/voronoi_builder.hpp`, `boost/polygon/voronoi_diagram.hpp`, and `boost/polygo`
- Bimap from `boost/bimap`
- Statistical Distributions and Functions from `boost/math/distributions`
- Special Functions from `boost/math/special_functions`
- Root Finding Functions from `boost/math/tools`
- Multi-index from `boost/multi_index`
- Heap from `boost/heap`
- The flat containers from Container: `boost/container/flat_map`, and `boost/container/flat_set`

- Intrusive from `boost/intrusive`.
- The boost/sort library.
- Preprocessor from `boost/preprocessor`.

We are actively considering adding other Boost features to the list, so this list may be expanded in the future.

8.22 Other C++ Features

As with Boost, some modern C++ extensions encourage coding practices that hamper readability—for example by removing checked redundancy (such as type names) that may be helpful to readers, or by encouraging template metaprogramming. Other extensions duplicate functionality available through existing mechanisms, which may lead to confusion and conversion costs.

8.22.1 Decision

In addition to what's described in the rest of the style guide, the following C++ features may not be used:

- Compile-time rational numbers (`<ratio>`), because of concerns that it's tied to a more template-heavy interface style.
- The `<cfenv>` and `<fenv.h>` headers, because many compilers do not support those features reliably.
- The `<filesystem>` header, which does not have sufficient support for testing, and suffers from inherent security vulnerabilities.

8.23 Nonstandard Extensions

Nonstandard extensions to C++ may not be used unless otherwise specified.

8.23.1 Definition

Compilers support various extensions that are not part of standard C++. Such extensions include GCC's `__attribute__`, intrinsic functions such as `__builtin_prefetch`, inline assembly, `__COUNTER__`, `__PRETTY_FUNCT` compound statement expressions (e.g., `foo = (\{ int x; Bar(\&x); x \})`, variable-length arrays and `alloca()`, and the "Elvis Operator" `a?:b`.

8.23.2 Pros

- Nonstandard extensions may provide useful features that do not exist in standard C++.
- Important performance guidance to the compiler can only be specified using extensions.

8.23.3 Cons

- Nonstandard extensions do not work in all compilers. Use of nonstandard extensions reduces portability of code.
- Even if they are supported in all targeted compilers, the extensions are often not well-specified, and there may be subtle behavior differences between compilers.
- Nonstandard extensions add to the language features that a reader must know to understand the code.

8.23.4 Decision

Do not use nonstandard extensions. You may use portability wrappers that are implemented using non-standard extensions, so long as those wrappers are provided by a designated project-wide portability header.

8.24 Aliases

Public aliases are for the benefit of an API's user, and should be clearly documented.

8.24.1 Definition

There are several ways to create names that are aliases of other entities:

```
typedef Foo Bar;
using Bar = Foo;
using other_namespace::Foo;
```

In new code, `using` is preferable to `typedef`, because it provides a more consistent syntax with the rest of C++ and works with templates.

Like other declarations, aliases declared in a header file are part of that header's public API unless they're in a function definition, in the private portion of a class, or in an explicitly-marked internal namespace. Aliases in such areas or in `.cc` files are implementation details (because client code can't refer to them), and are not restricted by this rule.

8.24.2 Pros

- Aliases can improve readability by simplifying a long or complicated name.
- Aliases can reduce duplication by naming in one place a type used repeatedly in an API, which *might* make it easier to change the type later.

8.24.3 Cons

- When placed in a header where client code can refer to them, aliases increase the number of entities in that header's API, increasing its complexity.
- Clients can easily rely on unintended details of public aliases, making changes difficult.
- It can be tempting to create a public alias that is only intended for use in the implementation, without considering its impact on the API, or on maintainability.
- Aliases can create risk of name collisions
- Aliases can reduce readability by giving a familiar construct an unfamiliar name
- Type aliases can create an unclear API contract: it is unclear whether the alias is guaranteed to be identical to the type it aliases, to have the same API, or only to be usable in specified narrow ways

8.24.4 Decision

Don't put an alias in your public API just to save typing in the implementation; do so only if you intend it to be used by your clients.

When defining a public alias, document the intent of the new name, including whether it is guaranteed to always be the same as the type it's currently aliased to, or whether a more limited compatibility is intended. This lets the user know whether they can treat the types as substitutable or whether more specific rules must be followed, and can help the implementation retain some degree of freedom to change the alias.

Don't put namespace aliases in your public API. (See also Namespaces).

For example, these aliases document how they are intended to be used in client code:

```
namespace mynamespace {
// Used to store field measurements. DataPoint may change from Bar* to some
↪  internal type.
// Client code should treat it as an opaque pointer.
using DataPoint = ::foo::Bar*;

// A set of measurements. Just an alias for user convenience.
using TimeSeries = std::unordered_set<DataPoint, std::hash<DataPoint>,
↪  DataPointComparator>;
}  // namespace mynamespace
```

These aliases don't document intended use, and half of them aren't meant for client use:

```
namespace mynamespace {
// Bad: none of these say how they should be used.
using DataPoint = ::foo::Bar*;
using ::std::unordered_set;  // Bad: just for local convenience
using ::std::hash;           // Bad: just for local convenience
typedef unordered_set<DataPoint, hash<DataPoint>, DataPointComparator> TimeSeries;
}  // namespace mynamespace
```

However, local convenience aliases are fine in function definitions, private sections of classes, explicitly marked internal namespaces, and in `.cc` files:

```
// In a .cc file
using ::foo::Bar;
```

Chapter 9
Inclusive Language

In all code, including naming and comments, use inclusive language and avoid terms that other programmers might find disrespectful or offensive (such as “master” and “slave”, “blacklist” and “whitelist”, or “redline”), even if the terms also have an ostensibly neutral meaning. Similarly, use gender-neutral language unless you’re referring to a specific person (and using their pronouns). For example, use “they”/“them”/“their” for people of unspecified gender (even when singular), and “it”/“its” for software, computers, and other things that aren’t people.

Chapter 10
Naming

Introduction

- *General Naming Rules*
- *File Names*
- *Type Names*
- *Variable Names*

The most important consistency rules are those that govern naming. The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring us to search for the declaration of that entity. The pattern-matching engine in our brains relies a great deal on these naming rules.

Naming rules are pretty arbitrary, but we feel that consistency is more important than individual preferences in this area, so regardless of whether you find them sensible or not, the rules are the rules.

10.1 General Naming Rules

Optimize for readability using names that would be clear even to people on a different team.

Use names that describe the purpose or intent of the object. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Minimize the use of abbreviations that would likely be unknown to someone outside your project (especially acronyms and initialisms). Do not abbreviate by deleting letters within a word. As a rule of thumb, an abbreviation is probably OK if it's listed in Wikipedia. Generally speaking, descriptiveness should be proportional to the name's scope of visibility. For example, `n` may be a fine name within a 5-line function, but within the scope of a class, it's likely too vague.

```cpp
class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int n = 0;  // Clear meaning given limited scope and context
    for (const auto& foo : foos) {
      ...
      ++n;
    }
    return n;
  }
  void DoSomethingImportant() {
    std::string fqdn = ...;  // Well-known abbreviation for Fully Qualified Domain
↪   Name
  }
 private:
  const int kMaxAllowedConnections = ...;  // Clear meaning within context
};
```

```
class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int total_number_of_foo_errors = 0;  // Overly verbose given limited scope and
    ↪ context
    for (int foo_index = 0; foo_index < foos.size(); ++foo_index) {  // Use
    ↪ idiomatic `i`
      ...
      ++total_number_of_foo_errors;
    }
    return total_number_of_foo_errors;
  }
  void DoSomethingImportant() {
    int cstmr_id = ...;  // Deletes internal letters
  }
 private:
  const int kNum = ...;  // Unclear meaning within broad scope
};
```

Note that certain universally-known abbreviations are OK, such as i for an iteration variable and T for a template parameter.

For the purposes of the naming rules below, a "word" is anything that you would write in English without internal spaces. This includes abbreviations, such as acronyms and initialisms. For names written in mixed case (also sometimes referred to as "camel case" or "Pascal case"), in which the first letter of each word is capitalized, prefer to capitalize abbreviations as single words, e.g., `StartRpc()` rather than `StartRPC()`.

Template parameters should follow the naming style for their category: type template parameters should follow the rules for type names, and non-type template parameters should follow the rules for variable names.

10.2 File Names

Filenames should be all lowercase and can include underscores (`_`) or dashes (`-`). Follow the convention that your project uses. If there is no consistent local pattern to follow, prefer "`_`".

Examples of acceptable file names:

- `my_useful_class.cc`
- `my-useful-class.cc`
- `myusefulclass.cc`
- `myusefulclass_test.cc // _unittest and _regtest are deprecated.`

C++ files should end in `.cc` and header files should end in `.h`. Files that rely on being textually included at specific points should end in `.inc` (see also the section on self-contained headers).

Do not use filenames that already exist in `/usr/include`, such as `db.h`.

In general, make your filenames very specific. For example, use `http_server_logs.h` rather than `logs.h`. A very common case is to have a pair of files called, e.g., `foo_bar.h` and `foo_bar.cc`, defining a class called `FooBar`.

10.3 Type Names

Type names start with a capital letter and have a capital letter for each new word, with no underscores: `MyExcitingClass`, `MyExcitingEnum`.

The names of all types — classes, structs, type aliases, enums, and type template parameters — have the same naming convention. Type names should start with a capital letter and have a capital letter for each new word. No underscores. For example:

```
// classes and structs
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

// typedefs
typedef hash_map<UrlTableProperties *, std::string> PropertiesMap;

// using aliases
using PropertiesMap = hash_map<UrlTableProperties *, std::string>;

// enums
enum class UrlTableError { ...
```

10.4 Variable Names

The names of variables (including function parameters) and data members are all lowercase, with underscores between words. Data members of classes (but not structs) additionally have trailing underscores. For instance: `a_local_variable`, `a_struct_data_member`, `a_class_data_member_`.

10.4.1 Common Variable names

For example:

```
std::string table_name;
```

```
std::string tableName;   // Bad - mixed case.
```

10.4.2 Class Data Members

Data members of classes, both static and non-static, are named like ordinary nonmember variables, but with a trailing underscore.

```
class TableInfo {
  ...
 private:
  std::string table_name_;  // OK - underscore at end.
  static Pool<TableInfo>* pool_;  // OK.
};
```

10.4.3 Struct Data Members

Data members of structs, both static and non-static, are named like ordinary nonmember variables. They do not have the trailing underscores that data members in classes have.

```
struct UrlTableProperties {
  std::string name;
  int num_entries;
  static Pool<UrlTableProperties>* pool;
};
```

See Structs vs. Classes for a discussion of when to use a struct versus a class.

10.4.4 Constant Names

Variables declared constexpr or const, and whose value is fixed for the duration of the program, are named with a leading "k" followed by mixed case. Underscores can be used as separators in the rare cases where capitalization cannot be used for separation. For example:

```
const int kDaysInAWeek = 7;
const int kAndroid8_0_0 = 24;  // Android 8.0.0
```

All such variables with static storage duration (i.e., statics and globals, see Storage Duration for details) should be named this way. This convention is optional for variables of other storage classes, e.g., automatic variables, otherwise the usual variable naming rules apply.

10.4.5 Function Names

Regular functions have mixed case; accessors and mutators may be named like variables.

Ordinarily, functions should start with a capital letter and have a capital letter for each new word.

```
AddTableEntry()
DeleteUrl()
OpenFileOrDie()
```

(The same naming rule applies to class- and namespace-scope constants that are exposed as part of an API and that are intended to look like functions, because the fact that they're objects rather than functions is an unimportant implementation detail.)

Accessors and mutators (get and set functions) may be named like variables. These often correspond to actual member variables, but this is not required. For example, `int count()` and void `set_count(int count)`.

10.4.6 Namespace Names

Namespace names are all lower-case, with words separated by underscores. Top-level namespace names are based on the project name . Avoid collisions between nested namespaces and well-known top-level namespaces. The name of a top-level namespace should usually be the name of the project or team whose code is contained in that namespace. The code in that namespace should usually be in a directory whose basename matches the namespace name (or in subdirectories thereof).

Keep in mind that the rule against abbreviated names applies to namespaces just as much as variable names. Code inside the namespace seldom needs to mention the namespace name, so there's usually no particular need for abbreviation anyway.

Avoid nested namespaces that match well-known top-level namespaces. Collisions between namespace names can lead to surprising build breaks because of name lookup rules. In particular, do not create any nested `std` namespaces. Prefer unique project identifiers () `websearch::index, websearch::index_util`) over collision-prone names like `websearch::util` . Also avoid overly deep nesting namespaces (TotW #130).

For `internal` namespaces, be wary of other code being added to the same `internal` namespace causing a collision (internal helpers within a team tend to be related and may lead to collisions). In such a situation, using the filename to make a unique internal name is helpful (`websearch::index::frobber_internal` for use in `frobber.h`).

10.4.7 Enumerator Names

Enumerators (for both scoped and unscoped enums) should be named like constants, not like macros. That is, use `kEnumName` not `ENUM_NAME` .

```
enum class UrlTableError {
  kOk = 0,
  kOutOfMemory,
  kMalformedInput,
};
```

```
enum class AlternateUrlTableError {
  OK = 0,
  OUT_OF_MEMORY = 1,
  MALFORMED_INPUT = 2,
};
```

Until January 2009, the style was to name enum values like macros. This caused problems with name collisions between enum values and macros. Hence, the change to prefer constant-style naming was put in place. New code should use constant-style naming.

10.4.8 Macro Names

You're not really going to define a macro, are you? If you do, they're like this: `MY_MACRO_THAT_SCARES_SMALL_CHILD`

Please see the description of macros; in general macros should *not* be used. However, if they are absolutely needed, then they should be named with all capitals and underscores.

```
#define ROUND(x) ...
#define PI_ROUNDED 3.0
```

10.4.9 Exceptions to Naming Rules

If you are naming something that is analogous to an existing C or C++ entity then you can follow the existing naming convention scheme.

- `bigopen()` : function name, follows form of `open()`
- `uint` : `typedef`
- `bigpos` : `struct` or `class`, follows form of `pos`
- `sparse_hash_map` : STL-like entity; follows STL naming conventions
- `LONGLONG_MAX` : a constant, as in `INT_MAX`

Chapter 11
Comments

Introduction

- *Comment Style*
- *File Comments*
- *Class Comments*
- *Function Comments*
- *Variable Comments*
- *Implementation Comments*
- *Function Argument Comments*
- *Don’ts*
- *Punctuation, Spelling, and Grammar*
- *TODO Comments*

Comments are absolutely vital to keeping our code readable. The following rules describe what you should comment and where. But remember: while comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments.

When writing your comments, write for your audience: the next contributor who will need to understand your code. Be generous — the next one may be you!

11.1 Comment Style

Use either the `//` or `/* */` syntax, as long as you are consistent.

You can use either the `//` or the `/* */` syntax; however, `//` is much more common. Be consistent with how you comment and what style you use where.

11.2 File Comments

Start each file with license boilerplate.

File comments describe the contents of a file. If a file declares, implements, or tests exactly one abstraction that is documented by a comment at the point of declaration, file comments are not required. All other files must have file comments.

11.2.1 Legal Notice and Author Line

Every file should contain license boilerplate. Choose the appropriate boilerplate for the license used by the project (for example, Apache 2.0, BSD, LGPL, GPL).

If you make significant changes to a file with an author line, consider deleting the author line. New files should usually not contain copyright notice or author line.

11.2.2 File Contents

If a `.h` declares multiple abstractions, the file-level comment should broadly describe the contents of the file, and how the abstractions are related. A 1 or 2 sentence file-level comment may be sufficient. The detailed documentation about individual abstractions belongs with those abstractions, not at the file level.

Do not duplicate comments in both the `.h` and the `.cc`. Duplicated comments diverge.

11.3 Class Comments

Every non-obvious class or struct declaration should have an accompanying comment that describes what it is for and how it should be used.

```
// Iterates over the contents of a GargantuanTable.
// Example:
//    std::unique_ptr<GargantuanTableIterator> iter = table->NewIterator();
//    for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//      process(iter->key(), iter->value());
//    }
class GargantuanTableIterator {
  ...
};
```

The class comment should provide the reader with enough information to know how and when to use the class, as well as any additional considerations necessary to correctly use the class. Document the synchronization assumptions the class makes, if any. If an instance of the class can be accessed by multiple threads, take extra care to document the rules and invariants surrounding multithreaded use.

The class comment is often a good place for a small example code snippet demonstrating a simple and focused usage of the class.

When sufficiently separated (e.g., `.h` and `.cc` files), comments describing the use of the class should go together with its interface definition; comments about the class operation and implementation should accompany the implementation of the class's methods.

11.4 Function Comments

Declaration comments describe use of the function (when it is non-obvious); comments at the definition of a function describe operation.

11.4.1 Function Declarations

Almost every function declaration should have comments immediately preceding it that describe what the function does and how to use it. These comments may be omitted only if the function is simple and obvious (e.g., simple accessors for obvious properties of the class). Private methods and functions declared in `.cc` files are not exempt. Function comments should be written with an implied subject of *This function* and should start with the verb phrase; for example, "Opens the file", rather than "Open the file". In general, these comments do not describe how the function performs its task. Instead, that should be left to comments in the function definition.

Types of things to mention in comments at the function declaration:

- What the inputs and outputs are. If function argument names are provided in “backticks”, then code-indexing tools may be able to present the documentation better.
- For class member functions: whether the object remembers reference or pointer arguments beyond the duration of the method call. This is quite common for pointer/reference arguments to constructors.
- For each pointer argument, whether it is allowed to be null and what happens if it is.
- For each output or input/output argument, what happens to any state that argument is in. (E.g. is the state appended to or overwritten?).
- If there are any performance implications of how a function is used.

Here is an example:

```
// Returns an iterator for this table, positioned at the first entry
// lexically greater than or equal to `start_word`. If there is no
// such entry, returns a null pointer. The client must not use the
// iterator after the underlying GargantuanTable has been destroyed.
//
// This method is equivalent to:
//    std::unique_ptr<Iterator> iter = table->NewIterator();
//    iter->Seek(start_word);
//    return iter;
std::unique_ptr<Iterator> GetIterator(absl::string_view start_word) const;
```

However, do not be unnecessarily verbose or state the completely obvious.

When documenting function overrides, focus on the specifics of the override itself, rather than repeating the comment from the overridden function. In many of these cases, the override needs no additional documentation and thus no comment is required.

When commenting constructors and destructors, remember that the person reading your code knows what constructors and destructors are for, so comments that just say something like “destroys this object” are not useful. Document what constructors do with their arguments (for example, if they take ownership of pointers), and what cleanup the destructor does. If this is trivial, just skip the comment. It is quite common for destructors not to have a header comment.

11.4.2 Function Definitions

If there is anything tricky about how a function does its job, the function definition should have an explanatory comment. For example, in the definition comment you might describe any coding tricks you use, give an overview of the steps you go through, or explain why you chose to implement the function in the way you did rather than using a viable alternative. For instance, you might mention why it must acquire a lock for the first half of the function but why it is not needed for the second half.

Note you should *not* just repeat the comments given with the function declaration, in the `.h` file or wherever. It’s okay to recapitulate briefly what the function does, but the focus of the comments should be on how it does it.

11.5 Variable Comments

In general the actual name of the variable should be descriptive enough to give a good idea of what the variable is used for. In certain cases, more comments are required.

11.5.1 Class Data Members

The purpose of each class data member (also called an instance variable or member variable) must be clear. If there are any invariants (special values, relationships between members, lifetime requirements) not clearly expressed by the type and name, they must be commented. However, if the type and name suffice (`int num_events_;`), no comment is needed.

In particular, add comments to describe the existence and meaning of sentinel values, such as nullptr or -1, when they are not obvious. For example:

```
private:
 // Used to bounds-check table accesses. -1 means
 // that we don't yet know how many entries the table has.
 int num_total_entries_;
```

11.5.2 Global Variables

All global variables should have a comment describing what they are, what they are used for, and (if unclear) why they need to be global. For example:

```
// The total number of test cases that we run through in this regression test.
const int kNumTestCases = 6;
```

11.6 Implementation Comments

In your implementation you should have comments in tricky, non-obvious, interesting, or important parts of your code.

11.6.1 Explanatory Comments

Tricky or complicated code blocks should have comments before them.

11.7 Function Argument Comments

When the meaning of a function argument is nonobvious, consider one of the following remedies:

- If the argument is a literal constant, and the same constant is used in multiple function calls in a way that tacitly assumes they're the same, you should use a named constant to make that constraint explicit, and to guarantee that it holds.

- Consider changing the function signature to replace a `bool` argument with an `enum` argument. This will make the argument values self-describing.
- For functions that have several configuration options, consider defining a single class or struct to hold all the options, and pass an instance of that. This approach has several advantages. Options are referenced by name at the call site, which clarifies their meaning. It also reduces function argument count, which makes function calls easier to read and write. As an added benefit, you don’t have to change call sites when you add another option.
- Replace large or complex nested expressions with named variables.
- As a last resort, use comments to clarify argument meanings at the call site.

Consider the following example:

```cpp
// What are these arguments?
const DecimalNumber product = CalculateProduct(values, 7, false, nullptr);
```

versus:

```cpp
ProductOptions options;
options.set_precision_decimals(7);
options.set_use_cache(ProductOptions::kDontUseCache);
const DecimalNumber product =
    CalculateProduct(values, options, /*completion_callback=*/nullptr);
```

11.8 Don’ts

Do not state the obvious. In particular, don’t literally describe what code does, unless the behavior is nonobvious to a reader who understands C++ well. Instead, provide higher level comments that describe *why* the code does what it does, or make the code self describing.

Compare this:

```cpp
// Find the element in the vector.  <-- Bad: obvious!
if (std::find(v.begin(), v.end(), element) != v.end()) {
  Process(element);
}
```

To this:

```cpp
// Process "element" unless it was already processed.
if (std::find(v.begin(), v.end(), element) != v.end()) {
  Process(element);
}
```

Self-describing code doesn't need a comment. The comment from the example above would be obvious:

```
if (!IsAlreadyProcessed(element)) {
  Process(element);
}
```

11.9 Punctuation, Spelling, and Grammar

Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.

Comments should be as readable as narrative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the end of a line of code, can sometimes be less formal, but you should be consistent with your style.

Although it can be frustrating to have a code reviewer point out that you are using a comma when you should be using a semicolon, it is very important that source code maintain a high level of clarity and readability. Proper punctuation, spelling, and grammar help with that goal.

11.10 TODO Comments

Use `TODO` comments for code that is temporary, a short-term solution, or good-enough but not perfect.

`TODO`s should include the string `TODO` in all caps, followed by the name, e-mail address, bug ID, or other identifier of the person or issue with the best context about the problem referenced by the `TODO`. The main purpose is to have a consistent `TODO` that can be searched to find out how to get more details upon request. A `TODO` is not a commitment that the person referenced will fix the problem. Thus when you create a `TODO` with a name, it is almost always your name that is given.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations.
// TODO(bug 12345): remove the "Last visitors" feature.
```

If your `TODO` is of the form "At a future date do something" make sure that you either include a very specific date ("Fix by November 2005") or a very specific event ("Remove this code when all clients can handle XML responses.").

Chapter 12
Formatting

Introduction

- ❏ *Line Length*
- ❏ *Non-ASCII Characters*
- ❏ *Spaces vs. Tabs*
- ❏ *Function Declarations and Definitions*
- ❏ *Lambda Expressions*
- ❏ *Floating-point Literals*
- ❏ *Function Calls*
- ❏ *Braced Initializer List Format*
- ❏ *Conditionals*
- ❏ *Loops and Switch Statements*
- ❏ *Pointer and Reference Expressions*
- ❏ *Boolean Expressions*
- ❏ *Return Values*
- ❏ *Variable and Array Initialization*
- ❏ *Preprocessor Directives*
- ❏ *Class Format*
- ❏ *Constructor Initializer Lists*
- ❏ *Namespace Formatting*
- ❏ *Horizontal Whitespace*

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

To help you format code correctly, we've created a settings file for emacs.

12.1 Line Length

Each line of text in your code should be at most 80 characters long.

We recognize that this rule is controversial, but so much existing code already adheres to it, and we feel that consistency is important.

12.1.1 Pros

Those who favor this rule argue that it is rude to force them to resize their windows and there is no need for anything longer. Some folks are used to having several code windows side-by-side, and thus don't have room to widen their windows in any case. People set up their work environment assuming a particular maximum window width, and 80 columns has been the traditional standard. Why change it?

12.1.2 Cons

Proponents of change argue that a wider line can make code more readable. The 80-column limit is an hidebound throwback to 1960s mainframes; modern equipment has wide screens that can easily show longer lines.

12.1.3 Decision

80 characters is the maximum.

A line may exceed 80 characters if it is

- a comment line which is not feasible to split without harming readability, ease of cut and paste or auto-linking – e.g., if a line contains an example command or a literal URL longer than 80 characters.
- a string literal that cannot easily be wrapped at 80 columns. This may be because it contains URIs or other semantically-critical pieces, or because the literal contains an embedded language, or a multiline literal whose newlines are significant like help messages. In these cases, breaking up the literal would reduce readability, searchability, ability to click links, etc. Except for test code, such literals should appear at namespace scope near the top of a file. If a tool like Clang-Format doesn't recognize the unsplittable content, disable the tool around the content as necessary.
  (We must balance between usability/searchability of such literals and the readability of the code around them.)
- an include statement.
- a header guard
- a using-declaration

12.2 Non-ASCII Characters

Non-ASCII characters should be rare, and must use UTF-8 formatting.

You shouldn't hard-code user-facing text in source, even English, so use of non-ASCII characters should be rare. However, in certain cases it is appropriate to include such words in your code. For example, if your code parses data files from foreign sources, it may be appropriate to hard-code the non-ASCII string(s) used in those data files as delimiters. More commonly, unittest code (which does not need to be localized) might contain non-ASCII strings. In such cases, you should use UTF-8, since that is an encoding understood by most tools able to handle more than just ASCII.

Hex encoding is also OK, and encouraged where it enhances readability — for example, `"\xEF\xBB\xBF"`, or, even more simply, `u8"\uFEFF"`, is the Unicode zero-width no-break space character, which would be invisible if included in the source as straight UTF-8.

Use the `u8` prefix to guarantee that a string literal containing `"\uXXXX"` escape sequences is encoded as UTF-8. Do not use it for strings containing non-ASCII characters encoded as UTF-8, because that will produce incorrect output if the compiler does not interpret the source file as UTF-8.

You shouldn't use `char16_t` and `char32_t` character types, since they're for non-UTF-8 text. For similar reasons you also shouldn't use `wchar_t` (unless you're writing code that interacts with the Windows API, which uses `wchar_t` extensively).

12.3 Spaces vs. Tabs

Use only spaces, and indent 2 spaces at a time.

We use spaces for indentation. Do not use tabs in your code. You should set your editor to emit spaces when you hit the tab key.

12.4 Function Declarations and Definitions

Return type on the same line as function name, parameters on the same line if they fit. Wrap parameter lists which do not fit on a single line as you would wrap arguments in a function call.

Functions look like this:

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
  DoSomething();
  ...
}
```

If you have too much text to fit on one line:

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1, Type par_name2,
                                             Type par_name3) {
  DoSomething();
  ...
}
```

or if you cannot fit even the first parameter:

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,  // 4 space indent
    Type par_name2,
    Type par_name3) {
  DoSomething();  // 2 space indent
  ...
}
```

Some points to note:

- Choose good parameter names.
- A parameter name may be omitted only if the parameter is not used in the function's definition.
- If you cannot fit the return type and the function name on a single line, break between them.
- If you break after the return type of a function declaration or definition, do not indent.
- The open parenthesis is always on the same line as the function name.
- There is never a space between the function name and the open parenthesis.
- There is never a space between the parentheses and the parameters.
- The open curly brace is always on the end of the last line of the function declaration, not the start of the next line.
- The close curly brace is either on the last line by itself or on the same line as the open curly brace.
- There should be a space between the close parenthesis and the open curly brace.
- All parameters should be aligned if possible.

- Default indentation is 2 spaces.
- Wrapped parameters have a 4 space indent.

Unused parameters that are obvious from context may be omitted:

```
class Foo {
 public:
  Foo(const Foo&) = delete;
  Foo& operator=(const Foo&) = delete;
};
```

Unused parameters that might not be obvious should comment out the variable name in the function definition:

```
class Shape {
 public:
  virtual void Rotate(double radians) = 0;
};

class Circle : public Shape {
 public:
  void Rotate(double radians) override;
};

void Circle::Rotate(double /*radians*/) {}
```

```
// Bad - if someone wants to implement later, it's not clear what the
// variable means.
void Circle::Rotate(double) {}
```

Attributes, and macros that expand to attributes, appear at the very beginning of the function declaration or definition, before the return type:

```
ABSL_ATTRIBUTE_NOINLINE void ExpensiveFunction();
[[nodiscard]] bool IsOk();
```

12.5 Lambda Expressions

Format parameters and bodies as for any other function, and capture lists like other comma-separated lists. For by-reference captures, do not leave a space between the ampersand (`&`) and the variable name.

```
int x = 0;
auto x_plus_n = [&x](int n) -> int { return x + n; }
```

Short lambdas may be written inline as function arguments.

```
std::set<int> to_remove = {7, 8, 9};
std::vector<int> digits = {3, 9, 1, 8, 4, 7, 1};
digits.erase(std::remove_if(digits.begin(), digits.end(), [&to_remove](int i) {
               return to_remove.find(i) != to_remove.end();
             }),
             digits.end());
```

12.6 Floating-point Literals

Floating-point literals should always have a radix point, with digits on both sides, even if they use exponential notation. Readability is improved if all floating-point literals take this familiar form, as this helps ensure that they are not mistaken for integer literals, and that the `E`/`e` of the exponential notation is not mistaken for a hexadecimal digit. It is fine to initialize a floating-point variable with an integer literal (assuming the variable type can exactly represent that integer), but note that a number in exponential notation is never an integer literal.

```
float f = 1.f;
long double ld = -.5L;
double d = 1248e6;
```

```
float f = 1.0f;
float f2 = 1;   // Also OK
long double ld = -0.5L;
double d = 1248.0e6;
```

12.7 Function Calls

Either write the call all on a single line, wrap the arguments at the parenthesis, or start the arguments on a new line indented by four spaces and continue at that 4 space indent. In the absence of other considerations, use the minimum number of lines, including placing multiple arguments on each line where appropriate.

Function calls have the following format:

```
bool result = DoSomething(argument1, argument2, argument3);
```

If the arguments do not all fit on one line, they should be broken up onto multiple lines, with each subsequent line aligned with the first argument. Do not add spaces after the open paren or before the close paren:

```
bool result = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
```

Arguments may optionally all be placed on subsequent lines with a four space indent:

```
if (...) {
  ...
  ...
  if (...) {
    bool result = DoSomething(
        argument1, argument2,  // 4 space indent
        argument3, argument4);
    ...
  }
```

Put multiple arguments on a single line to reduce the number of lines necessary for calling a function unless there is a specific readability problem. Some find that formatting with strictly one argument on each line is more readable and simplifies editing of the arguments. However, we prioritize for the reader over the ease of editing arguments, and most readability problems are better addressed with the following techniques.

If having multiple arguments in a single line decreases readability due to the complexity or confusing nature of the expressions that make up some arguments, try creating variables that capture those arguments in a descriptive name:

```
int my_heuristic = scores[x] * y + bases[x];
bool result = DoSomething(my_heuristic, x, y, z);
```

Or put the confusing argument on its own line with an explanatory comment:

```
bool result = DoSomething(scores[x] * y + bases[x],  // Score heuristic.
                          x, y, z);
```

If there is still a case where one argument is significantly more readable on its own line, then put it on its own line. The decision should be specific to the argument which is made more readable rather than a general policy.

Sometimes arguments form a structure that is important for readability. In those cases, feel free to format the arguments according to that structure:

```
// Transform the widget by a 3x3 matrix.
my_widget.Transform(x1, x2, x3,
                    y1, y2, y3,
                    z1, z2, z3);
```

12.8 Braced Initializer List Format

Format a braced initializer list exactly like you would format a function call in its place.

If the braced list follows a name (e.g., a type or variable name), format as if the `{}` were the parentheses of a function call with that name. If there is no name, assume a zero-length name.

```
// Examples of braced init list on a single line.
return {foo, bar};
functioncall({foo, bar});
std::pair<int, int> p{foo, bar};

// When you have to wrap.
SomeFunction(
    {"assume a zero-length name before {"},
    some_other_function_parameter);
SomeType variable{
    some, other, values,
    {"assume a zero-length name before {"},
    SomeOtherType{
        "Very long string requiring the surrounding breaks.",
        some, other, values},
    SomeOtherType{"Slightly shorter string",
                  some, other, values}};
SomeType variable{
    "This is too long to fit all in one line"};
MyType m = {  // Here, you could also break before {.
    superlongvariablename1,
    superlongvariablename2,
    {short, interior, list},
    {interiorwrappinglist,
     interiorwrappinglist2}};
```

12.9 Conditionals

In an `if` statement, including its optional `else if` and `else` clauses, put one space between the `if` and the opening parenthesis, and between the closing parenthesis and the curly brace (if any), but no spaces between the parentheses and the condition or initializer. If the optional initializer is present, put a space or newline after the semicolon, but not before.

```
if(condition) {              // Bad - space missing after IF
if ( condition ) {           // Bad - space between the parentheses and the
↪  condition
if (condition){              // Bad - space missing before {
if(condition){               // Doubly bad

if (int a = f();a == 10) {   // Bad - space missing after the semicolon
```

Use curly braces for the controlled statements following `if`, `else if` and `else`. Break the line immediately after the opening brace, and immediately before the closing brace. A subsequent `else`, if any, appears on the same line as the preceding closing brace, separated by a space.

```
if (condition) {                   // no spaces inside parentheses, space before
↪  brace
  DoOneThing();                    // two space indent
  DoAnotherThing();
} else if (int a = f(); a != 3) {  // closing brace on new line, else on same line
  DoAThirdThing(a);
} else {
  DoNothing();
}
```

For historical reasons, we allow one exception to the above rules: if an `if` statement has no `else` or `else if` clauses, then the curly braces for the controlled statement or the line breaks inside the curly braces may be omitted if as a result the entire `if` statement appears on either a single line (in which case there is a space between the closing parenthesis and the controlled statement) or on two lines (in which case there is a line break after the closing parenthesis and there are no braces). For example, the following forms are allowed under this exception.

```
if (x == kFoo) return new Foo();

if (x == kBar)
  return new Bar(arg1, arg2, arg3);

if (x == kQuz) { return new Quz(1, 2, 3); }
```

Use this style only when the statement is brief, and consider that conditional statements with complex conditions or controlled statements may be more readable with curly braces. Some projects require curly braces always.

Finally, these are not permitted:

```
// Bad - IF statement with ELSE is missing braces
if (x) DoThis();
else DoThat();

// Bad - IF statement with ELSE does not have braces everywhere
if (condition)
  foo;
else {
  bar;
}

// Bad - IF statement is too long to omit braces
if (condition)
  // Comment
  DoSomething();

// Bad - IF statement is too long to omit braces
if (condition1 &&
    condition2)
  DoSomething();
```

12.10 Loops and Switch Statements

Switch statements may use braces for blocks. Annotate non-trivial fall-through between cases. Braces are optional for single-statement loops. Empty loop bodies should use either empty braces or `continue`.

`case` blocks in `switch` statements can have curly braces or not, depending on your preference. If you do include curly braces they should be placed as shown below.

If not conditional on an enumerated value, switch statements should always have a `default` case (in the case of an enumerated value, the compiler will warn you if any values are not handled). If the default case should never execute, treat this as an error. For example:

```
switch (var) {
  case 0: {  // 2 space indent
    ...      // 4 space indent
    break;
  }
  case 1: {
    ...
    break;
  }
  default: {
    assert(false);
  }
}
```

Fall-through from one case label to another must be annotated using the `[[fallthrough]];` attribute. `[[fallthrough]]` should be placed at a point of execution where a fall-through to the next case label occurs. A common exception is consecutive case labels without intervening code, in which case no annotation is needed.

```
switch (x) {
  case 41:  // No annotation needed here.
  case 43:
    if (dont_be_picky) {
      // Use this instead of or along with annotations in comments.
      [[fallthrough]];
    } else {
      CloseButNoCigar();
      break;
    }
  case 42:
    DoSomethingSpecial();
    [[fallthrough]];
  default:
    DoSomethingGeneric();
    break;
}
```

Braces are optional for single-statement loops.

```
for (int i = 0; i < kSomeNumber; ++i)
  printf("I love you\n");

for (int i = 0; i < kSomeNumber; ++i) {
  printf("I take it back\n");
}
```

Empty loop bodies should use either an empty pair of braces or `continue` with no braces, rather than a single semicolon.

```
while (condition) {
  // Repeat test until it returns false.
}
for (int i = 0; i < kSomeNumber; ++i) {}  // Good - one newline is also OK.
while (condition) continue;  // Good - continue indicates no logic.
```

```
while (condition);  // Bad - looks like part of do/while loop.
```

12.11 Pointer and Reference Expressions

No spaces around period or arrow. Pointer operators do not have trailing spaces.

The following are examples of correctly-formatted pointer and reference expressions:

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

Note that:

- There are no spaces around the period or arrow when accessing a member.
- Pointer operators have no space after the `*` or `&`.

When referring to a pointer or reference (variable declarations or definitions, arguments, return types, template parameters, etc), you may place the space before or after the asterisk/ampersand. In the trailing-space style, the space is elided in some cases (template parameters, etc).

```
// These are fine, space preceding.
char *c;
const std::string &str;
int *GetPointer();
std::vector<char *>

// These are fine, space following (or elided).
char* c;
const std::string& str;
int* GetPointer();
std::vector<char*>  // Note no space between '*' and '>'
```

You should do this consistently within a single file. When modifying an existing file, use the style in that file.

It is allowed (if unusual) to declare multiple variables in the same declaration, but it is disallowed if any of those have pointer or reference decorations. Such declarations are easily misread.

```cpp
// Fine if helpful for readability.
int x, y;
```

```cpp
int x, *y;  // Disallowed - no & or * in multiple declaration
int* x, *y;  // Disallowed - no & or * in multiple declaration; inconsistent
↪  spacing
char * c;  // Bad - spaces on both sides of *
const std::string & str;  // Bad - spaces on both sides of &
```

12.12 Boolean Expressions

When you have a boolean expression that is longer than the standard line length, be consistent in how you break up the lines.

In this example, the logical AND operator is always at the end of the lines:

```cpp
if (this_one_thing > this_other_thing &&
    a_third_thing == a_fourth_thing &&
    yet_another && last_one) {
  ...
}
```

Note that when the code wraps in this example, both of the `&&` logical AND operators are at the end of the line. This is more common in Google code, though wrapping all operators at the beginning of the line is also allowed. Feel free to insert extra parentheses judiciously because they can be very helpful in increasing readability when used appropriately, but be careful about overuse. Also note that you should always use the punctuation operators, such as `&&` and `~`, rather than the word operators, such as `and` and `compl`.

12.13 Return Values

Do not needlessly surround the `return` expression with parentheses.

Use parentheses in `return expr;` only where you would use them in `x = expr;`.

```cpp
return result;                  // No parentheses in the simple case.
// Parentheses OK to make a complex expression more readable.
return (some_long_condition &&
        another_condition);
```

```
return (value);                // You wouldn't write var = (value);
return(result);                // return is not a function!
```

12.14 Variable and Array Initialization

You may choose between `=`, `()`, and `{}`; the following are all correct:

```
int x = 3;
int x(3);
int x{3};
std::string name = "Some Name";
std::string name("Some Name");
std::string name{"Some Name"};
```

Be careful when using a braced initialization list `{...}` on a type with an `std::initializer_list` constructor. A nonempty *braced-init-list* prefers the `std::initializer_list` constructor whenever possible. Note that empty braces `{}` are special, and will call a default constructor if available. To force the non-`std::initializer_list` constructor, use parentheses instead of braces.

```
std::vector<int> v(100, 1);  // A vector containing 100 items: All 1s.
std::vector<int> v{100, 1};  // A vector containing 2 items: 100 and 1.
```

Also, the brace form prevents narrowing of integral types. This can prevent some types of programming errors.

```
int pi(3.14);  // OK -- pi == 3.
int pi{3.14};  // Compile error: narrowing conversion.
```

12.15 Preprocessor Directives

The hash mark that starts a preprocessor directive should always be at the beginning of the line.

Even when preprocessor directives are within the body of indented code, the directives should start at the beginning of the line.

```
// Good - directives at beginning of line
  if (lopsided_score) {
#if DISASTER_PENDING      // Correct -- Starts at beginning of line
    DropEverything();
# if NOTIFY               // OK but not required -- Spaces after #
    NotifyClient();
# endif
#endif
    BackToNormal();
  }
```

```
// Bad - indented directives
  if (lopsided_score) {
    #if DISASTER_PENDING  // Wrong!  The "#if" should be at beginning of line
    DropEverything();
    #endif                // Wrong!  Do not indent "#endif"
    BackToNormal();
  }
```

12.16 Class Format

Sections in `public`, `protected` and `private` order, each indented one space.

The basic format for a class definition (lacking the comments, see Class Comments for a discussion of what comments are needed) is:

```
class MyClass : public OtherClass {
 public:      // Note the 1 space indent!
  MyClass();  // Regular 2 space indent.
  explicit MyClass(int var);
  ~MyClass() {}

  void SomeFunction();
  void SomeFunctionThatDoesNothing() {
  }

  void set_some_var(int var) { some_var_ = var; }
  int some_var() const { return some_var_; }

 private:
  bool SomeInternalFunction();

  int some_var_;
  int some_other_var_;
};
```

Things to note:

- Any base class name should be on the same line as the subclass name, subject to the 80-column limit.
- The `public:`, `protected:`, and `private:` keywords should be indented one space.
- Except for the first instance, these keywords should be preceded by a blank line. This rule is optional in small classes.
- Do not leave a blank line after these keywords.
- The public section should be first, followed by the `protected` and finally the `private` section.
- See Declaration Order for rules on ordering declarations within each of these sections.

12.17 Constructor Initializer Lists

Constructor initializer lists can be all on one line or with subsequent lines indented four spaces.

The acceptable formats for initializer lists are:

```
// When everything fits on one line:
MyClass::MyClass(int var) : some_var_(var) {
  DoSomething();
}

// If the signature and initializer list are not all on one line,
// you must wrap before the colon and indent 4 spaces:
MyClass::MyClass(int var)
    : some_var_(var), some_other_var_(var + 1) {
  DoSomething();
}

// When the list spans multiple lines, put each member on its own line
// and align them:
MyClass::MyClass(int var)
    : some_var_(var),             // 4 space indent
      some_other_var_(var + 1) {  // lined up
  DoSomething();
}

// As with any other code block, the close curly can be on the same
// line as the open curly, if it fits.
MyClass::MyClass(int var)
    : some_var_(var) {}
```

12.18 Namespace Formatting

The contents of namespaces are not indented.

Namespaces do not add an extra level of indentation. For example, use:

```
namespace {

void foo() {  // Correct.  No extra indentation within namespace.
  ...
}

}  // namespace
```

Do not indent within a namespace:

```
namespace {

  // Wrong!  Indented when it should not be.
  void foo() {
    ...
  }

}  // namespace
```

12.19 Horizontal Whitespace

Use of horizontal whitespace depends on location. Never put trailing whitespace at the end of a line.

12.19.1 General

```
int i = 0;  // Two spaces before end-of-line comments.

void f(bool b) {  // Open braces should always have a space before them.
  ...
int i = 0;  // Semicolons usually have no space before them.
// Spaces inside braces for braced-init-list are optional.  If you use them,
// put them on both sides!
int x[] = { 0 };
int x[] = {0};

// Spaces around the colon in inheritance and initializer lists.
class Foo : public Bar {
 public:
  // For inline function implementations, put spaces between the braces
  // and the implementation itself.
  Foo(int b) : Bar(), baz_(b) {}  // No spaces inside empty braces.
  void Reset() { baz_ = 0; }  // Spaces separating braces from implementation.
  ...
```

Adding trailing whitespace can cause extra work for others editing the same file, when they merge, as can removing existing trailing whitespace. So: Don't introduce trailing whitespace. Remove it if you're already changing that line, or do it in a separate clean-up operation (preferably when no-one else is working on the file).

12.19.2 Loops and Conditionals

```
if (b) {          // Space after the keyword in conditions and loops.
} else {          // Spaces around else.
}
while (test) {}   // There is usually no space inside parentheses.
switch (i) {
for (int i = 0; i < 5; ++i) {
// Loops and conditions may have spaces inside parentheses, but this
// is rare.  Be consistent.
switch ( i ) {
if ( test ) {
for ( int i = 0; i < 5; ++i ) {
// For loops always have a space after the semicolon.  They may have a space
// before the semicolon, but this is rare.
for ( ; i < 5 ; ++i) {
  ...

// Range-based for loops always have a space before and after the colon.
for (auto x : counts) {
  ...
}
switch (i) {
  case 1:         // No space before colon in a switch case.
    ...
  case 2: break;  // Use a space after a colon if there's code after it.
```

12.19.3 Operators

```
// Assignment operators always have spaces around them.
x = 0;
```

```

// Other binary operators usually have spaces around them, but it's
// OK to remove spaces around factors.  Parentheses should have no
// internal padding.
v = w * x + y / z;
v = w*x + y/z;
v = w * (x + z);

// No spaces separating unary operators and their arguments.
x = -5;
++x;
if (x && !y)
  ...

```

12.19.4 Templates and Casts

```
// No spaces inside the angle brackets (< and >), before
// <, or between >( in a cast
std::vector<std::string> x;
y = static_cast<char*>(x);

// Spaces between type and pointer are OK, but be consistent.
std::vector<char *> x;
```

12.19.5 Vertical Whitespace

Minimize use of vertical whitespace.

This is more a principle than a rule: don't use blank lines when you don't have to. In particular, don't put more than one or two blank lines between functions, resist starting functions with a blank line, don't end functions with a blank line, and be sparing with your use of blank lines. A blank line within a block of code serves like a paragraph break in prose: visually separating two thoughts.

The basic principle is: The more code that fits on one screen, the easier it is to follow and understand the control flow of the program. Use whitespace purposefully to provide separation in that flow.

Some rules of thumb to help when blank lines may be useful:

- Blank lines at the beginning or end of a function do not help readability.
- Blank lines inside a chain of if-else blocks may well help readability.
- A blank line before a comment line usually helps readability — the introduction of a new comment suggests the start of a new thought, and the blank line makes it clear that the comment goes with the following thing instead of the preceding.

- Blank lines immediately inside a declaration of a namespace or block of namespaces may help readability by visually separating the load-bearing content from the (largely non-semantic) organizational wrapper. Especially when the first declaration inside the namespace(s) is preceded by a comment, this becomes a special case of the previous rule, helping the comment to “attach” to the subsequent declaration.

Chapter 13
Exceptions to the Rules

Introduction

❑ *Existing Non-conformant Code* ❑ *Windows Code*

The coding conventions described above are mandatory. However, like all good rules, these sometimes have exceptions, which we discuss here.

13.1 Existing Non-conformant Code

You may diverge from the rules when dealing with code that does not conform to this style guide.

If you find yourself modifying code that was written to specifications other than those presented by this guide, you may have to diverge from these rules in order to stay consistent with the local conventions in that code. If you are in doubt about how to do this, ask the original author or the person currently responsible for the code. Remember that consistency includes local consistency, too.

13.2 Windows Code

Windows programmers have developed their own set of coding conventions, mainly derived from the conventions in Windows headers and other Microsoft code. We want to make it easy for anyone to understand your code, so we have a single set of guidelines for everyone writing C++ on any platform.

It is worth reiterating a few of the guidelines that you might forget if you are used to the prevalent Windows style:

- Do not use Hungarian notation (for example, naming an integer `iNum`). Use the Google naming conventions, including the `.cc` extension for source files.
- Windows defines many of its own synonyms for primitive types, such as `DWORD`, `HANDLE`, etc. It is perfectly acceptable, and encouraged, that you use these types when calling Windows API functions. Even so, keep as close as you can to the underlying C++ types. For example, use `const TCHAR *` instead of `LPCTSTR`.
- When compiling with Microsoft Visual C++, set the compiler to warning level 3 or higher, and treat all warnings as errors.
- Do not use `#pragma once`; instead use the standard Google include guards. The path in the include guards should be relative to the top of your project tree.
- In fact, do not use any nonstandard extensions, like `#pragma` and `__declspec`, unless you absolutely must. Using `__declspec(dllimport)` and `__declspec(dllexport)` is allowed; however, you must use them through macros such as `DLLIMPORT` and `DLLEXPORT`, so that someone can easily disable the extensions if they share the code.

However, there are just a few rules that we occasionally need to break on Windows:

- Normally we strongly discourage the use of multiple implementation inheritance; however, it is required when using COM and some ATL/WTL classes. You may use multiple implementation inheritance to implement COM or ATL/WTL classes and interfaces.

- Although you should not use exceptions in your own code, they are used extensively in the ATL and some STLs, including the one that comes with Visual C++. When using the ATL, you should define `_ATL_NO_EXCEPTIONS` to disable exceptions. You should investigate whether you can also disable exceptions in your STL, but if not, it is OK to turn on exceptions in the compiler. (Note that this is only to get the STL to compile. You should still not write exception handling code yourself.)
- The usual way of working with precompiled headers is to include a header file at the top of each source file, typically with a name like `StdAfx.h` or `precompile.h`. To make your code easier to share with other projects, avoid including this file explicitly (except in `precompile.cc`), and use the `/FI` compiler option to include the file automatically.
- Resource headers, which are usually named `resource.h` and contain only macros, do not need to conform to these style guidelines.

Appendix A
Singular “They”

The singular “they” is a generic third-person singular pronoun in English. Use of the singular “they” is endorsed as part of APA Style because it is inclusive of all people and helps writers avoid making assumptions about gender. Although usage of the singular “they” was once discouraged in academic writing, many advocacy groups and publishers have accepted and endorsed it, including Merriam-Webster’s Dictionary.

Always use a person’s self-identified pronoun, including when a person uses the singular “they” as their pronoun.Also use “they” as a generic third-person singular pronoun to refer to a person whose gender is unknown or irrelevant to the context of the usage.Do not use “he” or “she” alone as generic third-person singular pronouns. Use combination forms such as “he or she” and “she or he” only if you know that these pronouns match the people being described.Do not use combination forms such as “(s)he” and “s/he”. If you do not know the pronouns of the person being described, reword the sentence to avoid a pronoun or use the pronoun “they”.

A.1 Forms of the singular “they”

Use following forms of the singular “they”:

Form	Example
they	Casey is a gender-fluid person. They are from Texas and enjoy tacos.
them	Every client got a care package delivered to them.
their	Each child played with their parent.
theirs	The cup of coffee is theirs.
themselves (or themself)	A private person usually keeps to themselves [or themself].

Here are some tips to help you use the proper forms:

- Use a plural verb form with the singular pronoun “they” (i.e., write “they are” not “they is”).
- Use a singular verb form with a singular noun (i.e., write “Casey is” or “a person is”, not “Casey are” or “a person are”).
- Both “themselves” and “themself” are acceptable as reflexive singular pronouns; however, “themselves” is currently the more common usage.

A.2 Alternatives to the generic singular “they”

If using the singular “they” as a generic third-person pronoun seems awkward, try rewording the sentence or using the plural.

Strategy	Example
Rewording the sentence	I delivered a care package to the client.
Using the plural	Private people usually keep to themselves.

However, do not use alternatives when people use “they” as their pronoun—always use the pronouns that people use to refer to themselves.

Appendix B
Misc Use After Move

The Clang Team

Warns if an object is used after it has been moved, for example:

```
std::string str = "Hello, world!\n";
std::vector<std::string> messages;
messages.emplace_back(std::move(str));
std::cout << str;
```

The last line will trigger a warning that `str` is used after it has been moved.

The check does not trigger a warning if the object is reinitialized after the move and before the use. For example, no warning will be output for this code:

```
messages.emplace_back(std::move(str));
str = "Greetings, stranger!\n";
std::cout << str;
```

The check takes control flow into account. A warning is only emitted if the use can be reached from the move. This means that the following code does not produce a warning:

```
if (condition) {
  messages.emplace_back(std::move(str));
} else {
  std::cout << str;
}
```

On the other hand, the following code does produce a warning:

```
for (int i = 0; i < 10; ++i) {
  std::cout << str;
  messages.emplace_back(std::move(str));
}
```

(The use-after-move happens on the second iteration of the loop.)

In some cases, the check may not be able to detect that two branches are mutually exclusive. For example (assuming that `i` is an int):

```
if (i == 1) {
  messages.emplace_back(std::move(str));
}
if (i == 2) {
  std::cout << str;
}
```

In this case, the check will erroneously produce a warning, even though it is not possible for both the move and the use to be executed.

An erroneous warning can be silenced by reinitializing the object after the move:

```
if (i == 1) {
  messages.emplace_back(std::move(str));
  str = "";
}
if (i == 2) {
  std::cout << str;
}
```

Subsections below explain more precisely what exactly the check considers to be a move, use, and reinitialization.

B.1 Unsequenced moves, uses, and reinitializations

In many cases, C++ does not make any guarantees about the order in which sub-expressions of a statement are evaluated. This means that in code like the following, it is not guaranteed whether the use will happen before or after the move:

```
void f(int i, std::vector<int> v);
std::vector<int> v = { 1, 2, 3 };
f(v[1], std::move(v));
```

In this kind of situation, the check will note that the use and move are unsequenced.

The check will also take sequencing rules into account when reinitializations occur in the same statement as moves or uses. A reinitialization is only considered to reinitialize a variable if it is guaranteed to be evaluated after the move and before the use.

B.2 Move

The check currently only considers calls of `std::move` on local variables or function parameters. It does not check moves of member variables or global variables.

Any call of `std::move` on a variable is considered to cause a move of that variable, even if the result of `std::move` is not passed to an rvalue reference parameter.

This means that the check will flag a use-after-move even on a type that does not define a move constructor or move assignment operator. This is intentional. Developers may use `std::move` on such a type in the expectation that the type will add move semantics in the future. If such a `std::move` has the potential to cause a use-after-move, we want to warn about it even if the type does not implement move semantics yet.

Furthermore, if the result of `std::move` is passed to an rvalue reference parameter, this will always be considered to cause a move, even if the function that consumes this parameter does not move from it, or if it does so only conditionally. For example, in the following situation, the check will assume that a move always takes place:

```
std::vector<std::string> messages;
void f(std::string &&str) {
  // Only remember the message if it isn't empty.
  if (!str.empty()) {
    messages.emplace_back(std::move(str));
  }
}
std::string str = "";
f(std::move(str));
```

The check will assume that the last line causes a move, even though, in this particular case, it does not. Again, this is intentional.

When analyzing the order in which moves, uses and reinitializations happen (see section Unsequenced moves, uses, and reinitializations), the move is assumed to occur in whichever function the result of the `std::move` is passed to.

B.3 Use

Any occurrence of the moved variable that is not a reinitialization (see below) is considered to be a use.

An exception to this are objects of type `std::unique_ptr`, `std::shared_ptr` and `std::weak_ptr`, which have defined move behavior (objects of these classes are guaranteed to be empty after they have been moved from). Therefore, an object of these classes will only be considered to be used if it is dereferenced, i.e. if `operator*`, `operator->` or `operator[]` (in the case of `std::unique_ptr<T []>`) is called on it.

If multiple uses occur after a move, only the first of these is flagged.

B.4 Reinitialization

The check considers a variable to be reinitialized in the following cases:

- The variable occurs on the left-hand side of an assignment.
- The variable is passed to a function as a non-const pointer or non-const lvalue reference. (It is assumed that the variable may be an out-parameter for the function.)

- `clear()` or `assign()` is called on the variable and the variable is of one of the standard container types `basic_string`, `vector`, `deque`, `forward_list`, `list`, `set`, `map`, `multiset`, `multimap`, `unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap`.
- `reset()` is called on the variable and the variable is of type `std::unique_ptr`, `std::shared_ptr` or `std::weak_ptr`.

If the variable in question is a struct and an individual member variable of that struct is written to, the check does not consider this to be a reinitialization – even if, eventually, all member variables of the struct are written to. For example:

```
struct S {
  std::string str;
  int i;
};
S s = { "Hello, world!\n", 42 };
S s_other = std::move(s);
s.str = "Lorem ipsum";
s.i = 99;
```

The check will not consider `s` to be reinitialized after the last line; instead, the line that assigns to `s.str` will be flagged as a use-after-move. This is intentional as this pattern of reinitializing a struct is error-prone. For example, if an additional member variable is added to `S`, it is easy to forget to add the reinitialization for this additional member. Instead, it is safer to assign to the entire struct in one go, and this will also avoid the use-after-move warning.

Appendix C
absl/memory/memory.h

COMMIT 4E4E8D285490C122B22E4BA1C025C453E556DD1E

```cpp
// Copyright 2017 The Abseil Authors.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      https://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
//
// -----------------------------------------------------------------------------
// File: memory.h
// -----------------------------------------------------------------------------
//
// This header file contains utility functions for managing the creation and
// conversion of smart pointers. This file is an extension to the C++
// standard <memory> library header file.

#ifndef ABSL_MEMORY_MEMORY_H_
#define ABSL_MEMORY_MEMORY_H_

#include <cstddef>
#include <limits>
#include <memory>
#include <new>
#include <type_traits>
#include <utility>

#include "absl/base/macros.h"
#include "absl/meta/type_traits.h"

namespace absl {
```

```
ABSL_NAMESPACE_BEGIN

// -----------------------------------------------------------------------------
// Function Template: WrapUnique()
// -----------------------------------------------------------------------------
//
// Adopts ownership from a raw pointer and transfers it to the returned
// `std::unique_ptr`, whose type is deduced. Because of this deduction, *do not*
// specify the template type `T` when calling `WrapUnique`.
//
// Example:
//   X* NewX(int, int);
//   auto x = WrapUnique(NewX(1, 2));  // 'x' is std::unique_ptr<X>.
//
// Do not call WrapUnique with an explicit type, as in
// `WrapUnique<X>(NewX(1, 2))`.  The purpose of WrapUnique is to automatically
// deduce the pointer type. If you wish to make the type explicit, just use
// `std::unique_ptr` directly.
//
//   auto x = std::unique_ptr<X>(NewX(1, 2));
//                  - or -
//   std::unique_ptr<X> x(NewX(1, 2));
//
// While `absl::WrapUnique` is useful for capturing the output of a raw
// pointer factory, prefer 'absl::make_unique<T>(args...)' over
// 'absl::WrapUnique(new T(args...))'.
//
//   auto x = WrapUnique(new X(1, 2));  // works, but nonideal.
//   auto x = make_unique<X>(1, 2);     // safer, standard, avoids raw 'new'.
//
// Note that `absl::WrapUnique(p)` is valid only if `delete p` is a valid
// expression. In particular, `absl::WrapUnique()` cannot wrap pointers to
// arrays, functions or void, and it must not be used to capture pointers
// obtained from array-new expressions (even though that would compile!).
template <typename T>
std::unique_ptr<T> WrapUnique(T* ptr) {
  static_assert(!std::is_array<T>::value, "array types are unsupported");
  static_assert(std::is_object<T>::value, "non-object types are unsupported");
  return std::unique_ptr<T>(ptr);
}

// -----------------------------------------------------------------------------
// Function Template: make_unique<T>()
// -----------------------------------------------------------------------------
//
// Creates a `std::unique_ptr<>`, while avoiding issues creating temporaries
// during the construction process. `absl::make_unique<>` also avoids redundant
// type declarations, by avoiding the need to explicitly use the `new` operator.
//
// https://en.cppreference.com/w/cpp/memory/unique_ptr/make_unique
//
// For more background on why `std::unique_ptr<T>(new T(a,b))` is problematic,
// see Herb Sutter's explanation on
// (Exception-Safe Function Calls)[https://herbsutter.com/gotw/_102/].
// (In general, reviewers should treat `new T(a,b)` with scrutiny.)
//
// Historical note: Abseil once provided a C++11 compatible implementation of
// the C++14's `std::make_unique`. Now that C++11 support has been sunsetted,
// `absl::make_unique` simply uses the STL-provided implementation. New code
// should use `std::make_unique`.
using std::make_unique;

// -----------------------------------------------------------------------------
// Function Template: RawPtr()
// -----------------------------------------------------------------------------
//
// Extracts the raw pointer from a pointer-like value `ptr`. `absl::RawPtr` is
// useful within templates that need to handle a complement of raw pointers,
// `std::nullptr_t`, and smart pointers.
template <typename T>
auto RawPtr(T&& ptr) -> decltype(std::addressof(*ptr)) {
  // ptr is a forwarding reference to support Ts with non-const operators.
  return (ptr != nullptr) ? std::addressof(*ptr) : nullptr;
}
inline std::nullptr_t RawPtr(std::nullptr_t) { return nullptr; }

// -----------------------------------------------------------------------------
// Function Template: ShareUniquePtr()
// -----------------------------------------------------------------------------
//
// Adopts a `std::unique_ptr` rvalue and returns a `std::shared_ptr` of deduced
// type. Ownership (if any) of the held value is transferred to the returned
// shared pointer.
//
```

```
// Example:
//
//     auto up = absl::make_unique<int>(10);
//     auto sp = absl::ShareUniquePtr(std::move(up));  // shared_ptr<int>
//     CHECK_EQ(*sp, 10);
//     CHECK(up == nullptr);
//
// Note that this conversion is correct even when T is an array type, and more
// generally it works for *any* deleter of the `unique_ptr` (single-object
// deleter, array deleter, or any custom deleter), since the deleter is adopted
// by the shared pointer as well. The deleter is copied (unless it is a
// reference).
//
// Implements the resolution of [LWG 2415](http://wg21.link/lwg2415), by which a
// null shared pointer does not attempt to call the deleter.
template <typename T, typename D>
std::shared_ptr<T> ShareUniquePtr(std::unique_ptr<T, D>&& ptr) {
  return ptr ? std::shared_ptr<T>(std::move(ptr)) : std::shared_ptr<T>();
}

// -----------------------------------------------------------------------------
// Function Template: WeakenPtr()
// -----------------------------------------------------------------------------
//
// Creates a weak pointer associated with a given shared pointer. The returned
// value is a `std::weak_ptr` of deduced type.
//
// Example:
//
//    auto sp = std::make_shared<int>(10);
//    auto wp = absl::WeakenPtr(sp);
//    CHECK_EQ(sp.get(), wp.lock().get());
//    sp.reset();
//    CHECK(wp.lock() == nullptr);
//
template <typename T>
std::weak_ptr<T> WeakenPtr(const std::shared_ptr<T>& ptr) {
  return std::weak_ptr<T>(ptr);
}

namespace memory_internal {

```

```
// ExtractOr<E, O, D>::type evaluates to E<O> if possible. Otherwise, D.
template <template <typename> class Extract, typename Obj, typename Default,
          typename>
struct ExtractOr {
  using type = Default;
};

template <template <typename> class Extract, typename Obj, typename Default>
struct ExtractOr<Extract, Obj, Default, void_t<Extract<Obj>>> {
  using type = Extract<Obj>;
};

template <template <typename> class Extract, typename Obj, typename Default>
using ExtractOrT = typename ExtractOr<Extract, Obj, Default, void>::type;

// Extractors for the features of allocators.
template <typename T>
using GetPointer = typename T::pointer;

template <typename T>
using GetConstPointer = typename T::const_pointer;

template <typename T>
using GetVoidPointer = typename T::void_pointer;

template <typename T>
using GetConstVoidPointer = typename T::const_void_pointer;

template <typename T>
using GetDifferenceType = typename T::difference_type;

template <typename T>
using GetSizeType = typename T::size_type;

template <typename T>
using GetPropagateOnContainerCopyAssignment =
    typename T::propagate_on_container_copy_assignment;

template <typename T>
using GetPropagateOnContainerMoveAssignment =
    typename T::propagate_on_container_move_assignment;

```

```
template <typename T>
using GetPropagateOnContainerSwap = typename T::propagate_on_container_swap;

template <typename T>
using GetIsAlwaysEqual = typename T::is_always_equal;

template <typename T>
struct GetFirstArg;

template <template <typename...> class Class, typename T, typename... Args>
struct GetFirstArg<Class<T, Args...>> {
  using type = T;
};

template <typename Ptr, typename = void>
struct ElementType {
  using type = typename GetFirstArg<Ptr>::type;
};

template <typename T>
struct ElementType<T, void_t<typename T::element_type>> {
  using type = typename T::element_type;
};

template <typename T, typename U>
struct RebindFirstArg;

template <template <typename...> class Class, typename T, typename... Args,
          typename U>
struct RebindFirstArg<Class<T, Args...>, U> {
  using type = Class<U, Args...>;
};

template <typename T, typename U, typename = void>
struct RebindPtr {
  using type = typename RebindFirstArg<T, U>::type;
};

template <typename T, typename U>
struct RebindPtr<T, U, void_t<typename T::template rebind<U>>> {
  using type = typename T::template rebind<U>;
};
```

```

template <typename T, typename U>
constexpr bool HasRebindAlloc(...) {
  return false;
}

template <typename T, typename U>
constexpr bool HasRebindAlloc(typename T::template rebind<U>::other*) {
  return true;
}

template <typename T, typename U, bool = HasRebindAlloc<T, U>(nullptr)>
struct RebindAlloc {
  using type = typename RebindFirstArg<T, U>::type;
};

template <typename T, typename U>
struct RebindAlloc<T, U, true> {
  using type = typename T::template rebind<U>::other;
};

}  // namespace memory_internal

// --------------------------------------------------------------------------
// Class Template: pointer_traits
// --------------------------------------------------------------------------
//
// An implementation of C++11's std::pointer_traits.
//
// Provided for portability on toolchains that have a working C++11 compiler,
// but the standard library is lacking in C++11 support. For example, some
// version of the Android NDK.
//

template <typename Ptr>
struct pointer_traits {
  using pointer = Ptr;

  // element_type:
  // Ptr::element_type if present. Otherwise T if Ptr is a template
  // instantiation Template<T, Args...>
  using element_type = typename memory_internal::ElementType<Ptr>::type;

  // difference_type:
  // Ptr::difference_type if present, otherwise std::ptrdiff_t
  using difference_type =
      memory_internal::ExtractOrT<memory_internal::GetDifferenceType, Ptr,
                                  std::ptrdiff_t>;

  // rebind:
  // Ptr::rebind<U> if exists, otherwise Template<U, Args...> if Ptr is a
  // template instantiation Template<T, Args...>
  template <typename U>
  using rebind = typename memory_internal::RebindPtr<Ptr, U>::type;

  // pointer_to:
  // Calls Ptr::pointer_to(r)
  static pointer pointer_to(element_type& r) {  // NOLINT(runtime/references)
    return Ptr::pointer_to(r);
  }
};

// Specialization for T*.
template <typename T>
struct pointer_traits<T*> {
  using pointer = T*;
  using element_type = T;
  using difference_type = std::ptrdiff_t;

  template <typename U>
  using rebind = U*;

  // pointer_to:
  // Calls std::addressof(r)
  static pointer pointer_to(
      element_type& r) noexcept {  // NOLINT(runtime/references)
    return std::addressof(r);
  }
};

// --------------------------------------------------------------------------------
// Class Template: allocator_traits
// --------------------------------------------------------------------------------
//
```

```cpp
// A C++11 compatible implementation of C++17's std::allocator_traits.
//
#if __cplusplus >= 201703L || (defined(_MSVC_LANG) && _MSVC_LANG >= 201703L)
using std::allocator_traits;
#else  // __cplusplus >= 201703L
template <typename Alloc>
struct allocator_traits {
  using allocator_type = Alloc;

  // value_type:
  // Alloc::value_type
  using value_type = typename Alloc::value_type;

  // pointer:
  // Alloc::pointer if present, otherwise value_type*
  using pointer = memory_internal::ExtractOrT<memory_internal::GetPointer,
                                              Alloc, value_type*>;

  // const_pointer:
  // Alloc::const_pointer if present, otherwise
  // absl::pointer_traits<pointer>::rebind<const value_type>
  using const_pointer =
      memory_internal::ExtractOrT<memory_internal::GetConstPointer, Alloc,
                                  typename absl::pointer_traits<pointer>::
                                      template rebind<const value_type>>;

  // void_pointer:
  // Alloc::void_pointer if present, otherwise
  // absl::pointer_traits<pointer>::rebind<void>
  using void_pointer = memory_internal::ExtractOrT<
      memory_internal::GetVoidPointer, Alloc,
      typename absl::pointer_traits<pointer>::template rebind<void>>;

  // const_void_pointer:
  // Alloc::const_void_pointer if present, otherwise
  // absl::pointer_traits<pointer>::rebind<const void>
  using const_void_pointer = memory_internal::ExtractOrT<
      memory_internal::GetConstVoidPointer, Alloc,
      typename absl::pointer_traits<pointer>::template rebind<const void>>;

  // difference_type:
  // Alloc::difference_type if present, otherwise
  // absl::pointer_traits<pointer>::difference_type
  using difference_type = memory_internal::ExtractOrT<
      memory_internal::GetDifferenceType, Alloc,
      typename absl::pointer_traits<pointer>::difference_type>;

  // size_type:
  // Alloc::size_type if present, otherwise
  // std::make_unsigned<difference_type>::type
  using size_type = memory_internal::ExtractOrT<
      memory_internal::GetSizeType, Alloc,
      typename std::make_unsigned<difference_type>::type>;

  // propagate_on_container_copy_assignment:
  // Alloc::propagate_on_container_copy_assignment if present, otherwise
  // std::false_type
  using propagate_on_container_copy_assignment = memory_internal::ExtractOrT<
      memory_internal::GetPropagateOnContainerCopyAssignment, Alloc,
      std::false_type>;

  // propagate_on_container_move_assignment:
  // Alloc::propagate_on_container_move_assignment if present, otherwise
  // std::false_type
  using propagate_on_container_move_assignment = memory_internal::ExtractOrT<
      memory_internal::GetPropagateOnContainerMoveAssignment, Alloc,
      std::false_type>;

  // propagate_on_container_swap:
  // Alloc::propagate_on_container_swap if present, otherwise std::false_type
  using propagate_on_container_swap =
      memory_internal::ExtractOrT<memory_internal::GetPropagateOnContainerSwap,
                                  Alloc, std::false_type>;

  // is_always_equal:
  // Alloc::is_always_equal if present, otherwise std::is_empty<Alloc>::type
  using is_always_equal =
      memory_internal::ExtractOrT<memory_internal::GetIsAlwaysEqual, Alloc,
                                  typename std::is_empty<Alloc>::type>;

  // rebind_alloc:
  // Alloc::rebind<T>::other if present, otherwise Alloc<T, Args> if this Alloc
  // is Alloc<U, Args>
  template <typename T>
```

```
using rebind_alloc = typename memory_internal::RebindAlloc<Alloc, T>::type;

// rebind_traits:
// absl::allocator_traits<rebind_alloc<T>>
template <typename T>
using rebind_traits = absl::allocator_traits<rebind_alloc<T>>;

// allocate(Alloc& a, size_type n):
// Calls a.allocate(n)
static pointer allocate(Alloc& a,  // NOLINT(runtime/references)
                        size_type n) {
  return a.allocate(n);
}

// allocate(Alloc& a, size_type n, const_void_pointer hint):
// Calls a.allocate(n, hint) if possible.
// If not possible, calls a.allocate(n)
static pointer allocate(Alloc& a, size_type n,  // NOLINT(runtime/references)
                        const_void_pointer hint) {
  return allocate_impl(0, a, n, hint);
}

// deallocate(Alloc& a, pointer p, size_type n):
// Calls a.deallocate(p, n)
static void deallocate(Alloc& a, pointer p,  // NOLINT(runtime/references)
                       size_type n) {
  a.deallocate(p, n);
}

// construct(Alloc& a, T* p, Args&&... args):
// Calls a.construct(p, std::forward<Args>(args)...) if possible.
// If not possible, calls
//   ::new (static_cast<void*>(p)) T(std::forward<Args>(args)...)
template <typename T, typename... Args>
static void construct(Alloc& a, T* p,  // NOLINT(runtime/references)
                      Args&&... args) {
  construct_impl(0, a, p, std::forward<Args>(args)...);
}

// destroy(Alloc& a, T* p):
// Calls a.destroy(p) if possible. If not possible, calls p->~T().
template <typename T>
```

```
  static void destroy(Alloc& a, T* p) {  // NOLINT(runtime/references)
    destroy_impl(0, a, p);
  }

  // max_size(const Alloc& a):
  // Returns a.max_size() if possible. If not possible, returns
  //   std::numeric_limits<size_type>::max() / sizeof(value_type)
  static size_type max_size(const Alloc& a) { return max_size_impl(0, a); }

  // select_on_container_copy_construction(const Alloc& a):
  // Returns a.select_on_container_copy_construction() if possible.
  // If not possible, returns a.
  static Alloc select_on_container_copy_construction(const Alloc& a) {
    return select_on_container_copy_construction_impl(0, a);
  }

 private:
  template <typename A>
  static auto allocate_impl(int, A& a,  // NOLINT(runtime/references)
                            size_type n, const_void_pointer hint)
      -> decltype(a.allocate(n, hint)) {
    return a.allocate(n, hint);
  }
  static pointer allocate_impl(char, Alloc& a,  // NOLINT(runtime/references)
                               size_type n, const_void_pointer) {
    return a.allocate(n);
  }

  template <typename A, typename... Args>
  static auto construct_impl(int, A& a,  // NOLINT(runtime/references)
                             Args&&... args)
      -> decltype(a.construct(std::forward<Args>(args)...)) {
    a.construct(std::forward<Args>(args)...);
  }

  template <typename T, typename... Args>
  static void construct_impl(char, Alloc&, T* p, Args&&... args) {
    ::new (static_cast<void*>(p)) T(std::forward<Args>(args)...);
  }

  template <typename A, typename T>
  static auto destroy_impl(int, A& a,  // NOLINT(runtime/references)
                            T* p) -> decltype(a.destroy(p)) {
    a.destroy(p);
  }
  template <typename T>
  static void destroy_impl(char, Alloc&, T* p) {
    p->~T();
  }

  template <typename A>
  static auto max_size_impl(int, const A& a) -> decltype(a.max_size()) {
    return a.max_size();
  }
  static size_type max_size_impl(char, const Alloc&) {
    return (std::numeric_limits<size_type>::max)() / sizeof(value_type);
  }

  template <typename A>
  static auto select_on_container_copy_construction_impl(int, const A& a)
      -> decltype(a.select_on_container_copy_construction()) {
    return a.select_on_container_copy_construction();
  }
  static Alloc select_on_container_copy_construction_impl(char,
                                                          const Alloc& a) {
    return a;
  }
};
#endif  // __cplusplus >= 201703L

namespace memory_internal {

// This template alias transforms Alloc::is_nothrow into a metafunction with
// Alloc as a parameter so it can be used with ExtractOrT<>.
template <typename Alloc>
using GetIsNothrow = typename Alloc::is_nothrow;

}  // namespace memory_internal

// ABSL_ALLOCATOR_NOTHROW is a build time configuration macro for user to
// specify whether the default allocation function can throw or never throws.
// If the allocation function never throws, user should define it to a non-zero
// value (e.g. via `-DABSL_ALLOCATOR_NOTHROW`).
// If the allocation function can throw, user should leave it undefined or
// define it to zero.
//
// allocator_is_nothrow<Alloc> is a traits class that derives from
// Alloc::is_nothrow if present, otherwise std::false_type. It's specialized
// for Alloc = std::allocator<T> for any type T according to the state of
// ABSL_ALLOCATOR_NOTHROW.
//
// default_allocator_is_nothrow is a class that derives from std::true_type
// when the default allocator (global operator new) never throws, and
// std::false_type when it can throw. It is a convenience shorthand for writing
// allocator_is_nothrow<std::allocator<T>> (T can be any type).
// NOTE: allocator_is_nothrow<std::allocator<T>> is guaranteed to derive from
// the same type for all T, because users should specialize neither
// allocator_is_nothrow nor std::allocator.
template <typename Alloc>
struct allocator_is_nothrow
    : memory_internal::ExtractOrT<memory_internal::GetIsNothrow, Alloc,
                                  std::false_type> {};

#if defined(ABSL_ALLOCATOR_NOTHROW) && ABSL_ALLOCATOR_NOTHROW
template <typename T>
struct allocator_is_nothrow<std::allocator<T>> : std::true_type {};
struct default_allocator_is_nothrow : std::true_type {};
#else
struct default_allocator_is_nothrow : std::false_type {};
#endif

namespace memory_internal {
template <typename Allocator, typename Iterator, typename... Args>
void ConstructRange(Allocator& alloc, Iterator first, Iterator last,
                    const Args&... args) {
  for (Iterator cur = first; cur != last; ++cur) {
    ABSL_INTERNAL_TRY {
      std::allocator_traits<Allocator>::construct(alloc, std::addressof(*cur),
                                                  args...);
    }
    ABSL_INTERNAL_CATCH_ANY {
      while (cur != first) {
        --cur;
        std::allocator_traits<Allocator>::destroy(alloc, std::addressof(*cur));
      }
      ABSL_INTERNAL_RETHROW;
    }
  }
}

template <typename Allocator, typename Iterator, typename InputIterator>
void CopyRange(Allocator& alloc, Iterator destination, InputIterator first,
               InputIterator last) {
  for (Iterator cur = destination; first != last;
       static_cast<void>(++cur), static_cast<void>(++first)) {
    ABSL_INTERNAL_TRY {
      std::allocator_traits<Allocator>::construct(alloc, std::addressof(*cur),
                                                  *first);
    }
    ABSL_INTERNAL_CATCH_ANY {
      while (cur != destination) {
        --cur;
        std::allocator_traits<Allocator>::destroy(alloc, std::addressof(*cur));
      }
      ABSL_INTERNAL_RETHROW;
    }
  }
}
}  // namespace memory_internal
ABSL_NAMESPACE_END
}  // namespace absl

#endif  // ABSL_MEMORY_MEMORY_H_
```

Appendix D
Hyrum's Law

Hyrum Weight

An observation on Software Engineering

—Hyrum Weight

Put succinctly, the observation is this:

> With a sufficient number of users of an API,
> it does not matter what you promise in the contract:
> all observable behaviors of your system
> will be depended on by somebody.

Over the past couple years of doing low-level infrastructure migrations in one of the most complex software systems on the planet, I've made some observations about the differences between an interface and its implementations. We typically think of the interface as an abstraction for interacting with a system (like the steering wheel and pedals in a car), and the implementation as the way the system does its work (wheels and an engine). This is useful for a number of reasons, foremost among them that most useful systems rapidly become too complex for a single individual or group to completely understand, and abstractions are essential to managing that complexity.

Defining the correct level of abstraction is a completely separate discussion (see Mythical Man-Month), but we like to think that once an abstraction is defined, it is concrete. In other words, an interface should theoretically provide a clear separation between consumers of a system and its implementers. In practice, this theory breaks down as the use of a system grows and its users start to rely upon implementation details intentionally exposed through the interface, or which they divine through regular use. Spolsky's "Law of Leaky Abstractions" embodies consumers' reliance upon internal implementation details.

Taken to its logical extreme, this leads to the following observation, colloquially referred to as "The Law of Implicit Interfaces": Given enough use, there is no such thing as a private implementation. That is, if an interface has enough consumers, they will collectively depend on every aspect of the implementation, intentionally or not. This effect serves to constrain changes to the implementation, which must now conform to both the explicitly documented interface, as well as the implicit interface captured by usage. We often refer to this phenomenon as "bug-for-bug compatibility".

The creation of the implicit interface usually happens gradually, and interface consumers generally aren't aware as it's happening. For example, an interface may make no guarantees about performance, yet consumers often come to expect a certain level of performance from its implementation. Those expectations become part of the implicit interface to a system, and changes to the system must maintain these performance characteristics to continue functioning for its consumers.

Not all consumers depend upon the same implicit interface, but given enough consumers, the implicit interface will eventually exactly match the implementation. At this point, the interface has evaporated: the implementation has become the interface, and any changes to it will violate consumer expectations. With a bit of luck, widespread, comprehensive, and automated testing can detect these new expectations but not ameliorate them.

Implicit interfaces result from the organic growth of large systems, and while we may wish the problem did not exist, designers and engineers would be wise to consider it when building and maintaining complex systems. So be aware of how the implicit interface constrains your system design and evolution, and know that for any reasonably popular system, the interface reaches much deeper than you think.

D.1 Who's Hyrum?

I'm a Software Engineer at Google, working on large-scale code change tooling and infrastructure. Prior to that, I spent five years improving Google's core C++ libraries. The above observation grew out of experiences when even the simplest library change caused failures in some far off system.

While I may have made the observation, credit goes to Titus Winters for actually naming it as "Hyrum's Law" and popularizing the concept more broadly.

Appendix E
Tip of the Week #119: Using-declarations and namespace aliases

Thomas Köppe (tkoeppe@google.com)

This tip gives a simple, robust recipe for writing *using-declarations* and namespace aliases in `.cc` files that avoids subtle pitfalls.

Before we dive into the details, here is an example of the recipe in action:

```
namespace example {
namespace makers {
namespace {

using ::otherlib::BazBuilder;
using ::mylib::BarFactory;
namespace abc = ::applied::bitfiddling::concepts;

// Private helper code here.

}  // namespace

// Interface implementation code here.

}  // namespace makers
}  // namespace example
```

Remember that everything in this tip applies only to `.cc` files, since you should never put convenience aliases in header files. Such aliases are a convenience for the implementer (and the reader of an implementation), not an exported facility. (Names that *are* part of the exported API may of course be declared in headers.)

E.1 Summary

- Never declare namespace aliases or convenience *using-declarations* at namespace scope in header files, only in `.cc` files.
- Declare namespace aliases and *using-declarations* inside the innermost namespace, whether named or anonymous. (Do not add an anonymous namespace just for this purpose.)
- When declaring namespace aliases and *using-declarations*, use fully qualified names (with leading `::`) unless you are referring to a name inside the current namespace.
- For other uses of names, avoid fully qualifying when reasonable, see TotW #130.

(Remember that you can always have a local namespace alias or *using-declaration* in a block scope, which can be handy in header-only libraries.)

E.2 Background

C++ organizes names into *namespaces*. This crucial facility allows code bases to scale by keeping ownership of names local avoiding name collisions in other scopes. However, namespaces impose a certain cosmetic burden, since qualified names (`foo::Bar`) are often long and quickly become clutter. We often find it convenient to use *unqualified names* (`Bar`). Additionally, we may wish to introduce a namespace alias for a long but frequently used namespace: `namespace eu = example::v1::util;` We will collectively call *using-declarations* and namespace *aliases* just aliases in this tip.

E.3 The Problem

The purpose of a namespace is to help code authors avoid name collisions, both at the point of name lookup and at the point of linking. Aliases can potentially undermine the protection afforded by namespaces. The problem has two separate aspects: the scope of the alias, and the use of relative qualifiers.

E.3.0.1 Scope of the Alias

The scope at which you place an alias can have subtle effects on code maintainability. Consider the following two variants:

```
using ::foo::Quz;

namespace example {
namespace util {

using ::foo::Bar;
```

It appears that both *using-declarations* are effective at making the names `Bar` and `Quz` available for unqualified lookup inside our working namespace `::example::util`. For `Bar`, everything is working as expected, provided there is no other declaration of `Bar` inside namespace `::example::util`. But this is your namespace, so it is in your power to control this.

On the other hand, if a header is later included that declares a global name `Quz`, then the first *using-declaration* becomes ill-formed, as it attempts to redeclare the name `Quz`. And if another header declares `::example::Quz` or `::example::util::Quz`, then unqualified lookup will find that name rather than your alias.

This brittleness can be avoided if you do not add names to namespaces that you do not own (which includes the global namespace). By placing the aliases inside your own namespace, the unqualified lookup finds your alias first and never continues to search containing namespaces.

More generally, the closer a declaration is to the point of use, the smaller is the set of scopes that can break your code. At the worst end of our example is `Quz`, which can be broken by anyone; `Bar` can only be broken by other code in `::example::util`, and a name that is declared and used inside an unnamed namespace cannot be broken by any other scope. See Unnamed Namespaces for an example.

E.3.0.2 Relative Qualifiers

A *using-declaration* of the form `using foo::Bar` seems innocuous, but it is in fact ambiguous. The problem is that it is safe to rely on the existence of names in a namespace, but it is not safe to rely on the *absence* of names. Consider this code:

```
namespace example {
namespace util {

using foo::Bar;
```

It is perhaps the author's intention to use the name `::foo::Bar`. However, this can break, because the code is relying on the existence of `::foo::Bar` and also on the non-existence of namespaces `::example::foo` and `::example::util::foo`. This brittleness can be avoided by qualifying the used name fully: `using ::foo::Bar`.

The only time that a relative name is unambiguous and cannot possibly be broken by outside declarations is if it refers to a name that is already inside your current namespace:

```
namespace example {
namespace util {
namespace internal {

struct Params { /* ... */ };

}  // namespace internal

using internal::Params;  // OK, same as ::example::util::internal::Params
```

This follows the same logic that we discussed in the previous section.

What if a name lives in a sibling namespace, such as `::example::tools::Thing`? You can say either `tools::Thing` or `::example::tools::Thing`. The fully qualified name is always correct, but it may also be appropriate to use the relative name. Use your own judgment.

A cheap way to avoid a lot of these problems is not to use namespaces in your project that are the same as popular top-level namespaces (such as `util`); the Style Guide recommends this practice explicitly.

E.3.0.3 Demo

The following code shows examples of both failure modes.

`helper.h`:

```
namespace bar {
namespace foo {

// ...

}  // namespace foo
}  // namespace bar
```

`some_feature.h`:

```
extern int f;
```

`Your code`:

```
#include "helper.h"
#include "some_feature.h"

namespace foo {
void f();
}  // namespace foo

// Failure mode #1: Alias at a bad scope.
using foo::f;  // Error: redeclaration (because of "f" declared in some_feature.h)

namespace bar {

// Failure mode #2: Alias badly qualified.
using foo::f;  // Error: No "f" in namespace ::bar::foo (because that namespace was
↪  declared in helper.h)

// The recommended way, robust in the face of unrelated declarations:
using ::foo::f;  // OK

void UseCase() { f(); }

}  // namespace bar
```

E.4 Unnamed Namespaces

A *using-declaration* placed in an unnamed namespace can be accessed from the enclosing namespace and vice versa. If you already have an unnamed namespace at the top of the file, prefer putting all aliases there. From within that unnamed namespace, you gain an extra little bit of robustness against clashing with something declared in the enclosing namespace.

```
namespace example {
namespace util {

namespace {

// Put all using-declarations in here. Don't spread them over the file.
using ::foo::Bar;
using ::foo::Quz;

// In here, Bar and Quz refer inalienably to your aliases.

}  // namespace

// Can use both Bar and Quz here too. (But don't declare any entities called Bar or
↪  Quz yourself now.)
```

E.5 Non-aliased names

So far we have been talking about local aliases for distant names. But what if we want to use names directly and not create aliases at all? Should we say `util::Status` or `::util::Status`?

There is no obvious answer. Unlike the alias declarations that we have been discussing until now, which appear at the top of the file far away from the actual code, the direct use of names affects the local readability of your code. While it is true that relatively qualified names may break in the future, there is a significant cost for using absolute qualifications. The visual clutter caused by the leading `::` may well be distracting and not worth the added robustness. In this case, use your own judgment to decide which style you prefer. See TotW #130.

E.6 Acknowledgments

All credit is due to Roman Perepelitsa (romanp@google.com) who originally suggested this style in a mailing list discussion and who contributed numerous corrections and punchlines. However, all errors are mine.

Appendix F
Tip of the Week #42: Prefer Factory Functions to Initializer Methodsr

Geoffrey Romer (gromer@google.com)

The man who builds a factory builds a temple; the man who works there worships there, and to each is due, not scorn and blame, but reverence and praise.”
—Calvin Coolidge

In environments where exceptions are disallowed (such as within Google), C++ constructors effectively must succeed, because they have no way to report failure to the caller. You can use an `abort()`, of course, but doing so crashes the whole program, which is often unacceptable in production code.

If your class’s initialization logic can’t avoid the possibility of failure, one common approach is to give the class an initializer method (also called an “init method”), which performs any initialization work that might fail, and signals that failure via its return value. The assumption is usually that the user will call this method immediately after construction, and if it fails the user will immediately destroy the object. However, these assumptions are not always documented, nor always obeyed. It’s all too easy for users to start calling other methods before initialization, or after initialization has failed. Sometimes the class actually encourages this behavior, e.g. by providing methods to configure the object before initializing it, or to read errors out of it after initialization fails.

This design commits you to maintaining a class with at least two distinct user-visible states, and often three: initialized, uninitialized, and initialization-failed. Making such a design work requires a lot of discipline: every method of the class has to specify what states it can be called in, and users have to comply with these rules. If this discipline lapses, client developers will tend to write whatever code happens to work, regardless of what you intended to support. When that starts to happen, maintainability nosedives, because your implementation has to support whatever combinations of pre-initialization method calls your clients have started to depend on. In effect, your implementation has become your interface. (See Hyrum’s Law.)

Fortunately, there’s a simple alternative that lacks these drawbacks: provide a factory function which creates and initializes instances of your class, and returns them by pointer or as `absl::optional` (see TotW #123), using null to indicate failure. Here’s a toy example using `unique_ptr<>`:

```
// foo.h
class Foo {
 public:
  // Factory method: creates and returns a Foo.
  // May return null on failure.
  static std::unique_ptr<Foo> Create();

  // Foo is not copyable.
  Foo(const Foo&) = delete;
```

```cpp
  Foo& operator=(const Foo&) = delete;

 private:
  // Clients can't invoke the constructor directly.
  Foo();
};

// foo.c
std::unique_ptr<Foo> Foo::Create() {
  // Note that since Foo's constructor is private, we have to use new.
  return absl::WrapUnique(new Foo());
}
```

In many cases, this pattern gives you the best of both worlds: the factory function `Foo::Create()` exposes only fully-initialized objects like a constructor, but it can indicate failure like an initializer method. Another advantage of factory functions is that they can return instances of any subclass of the return type (though this is not possible if using `absl::optional` as the return type). This allows you to swap in a different implementation without updating user code, or even choose the implementation class dynamically, based on user input.

The primary drawback of this approach is that it returns a pointer to a heap-allocated object, so it's not well-suited for "value-like" classes designed to work on the stack. However, such classes usually don't require complex initialization in the first place. Factory functions also can't be used when a derived class constructor needs to initialize its base, so initializer methods are sometimes necessary in the protected API of a base class. The public API can still use factory functions, though.

Appendix G
Tip of the Week #55:Name Counting and `unique_ptr`

Though we may know Him by a thousand names, He is one and the same to us all.

—Mahatma Gandhi

Colloquially, a "name" for a value is any value-typed variable (not a pointer, nor a reference), in any scope, that holds a particular data value. (For the spec-lawyers, if we say "name" we're essentially talking about lvalues.) Because of `std::unique_ptr`'s specific behavioral requirements, we need to make sure that any value held in a `std::unique_ptr` only has one name.

It's important to note that the C++ language committee picked a very apt name for `std::unique_ptr`. Any non-null pointer value stored in a `std::unique_ptr` must occur in only one `std::unique_ptr` at any time; the standard library is designed to enforce this. Many common problems compiling code that uses `std::unique_ptr` can be resolved by learning to recognize how to count the names for a `std::unique_ptr`: one is OK, but multiple names for the same pointer value are not.

Let's count some names. At each line number, count the number of names alive at that point (whether in scope or not) that refer to a `std::unique_ptr` containing the same pointer. If you find any line with more than one name for the same pointer value, that's an error!

```
 1  std::unique_ptr<Foo> NewFoo() {
 2    return std::unique_ptr<Foo>(new Foo(1));
 3  }
 4
 5  void AcceptFoo(std::unique_ptr<Foo> f) { f->PrintDebugString(); }
 6
 7  void Simple() {
 8    AcceptFoo(NewFoo());
 9  }
10
11  void DoesNotBuild() {
12    std::unique_ptr<Foo> g = NewFoo();
13    AcceptFoo(g); // DOES NOT COMPILE!
14  }
15
16  void SmarterThanTheCompilerButNot() {
17    Foo* j = new Foo(2);
18    // Compiles, BUT VIOLATES THE RULE and will double-delete at runtime.
19    std::unique_ptr<Foo> k(j);
20    std::unique_ptr<Foo> l(j);
21  }
```

In `Simple()`, the unique pointer allocated with `NewFoo()` only ever has one name by which you could refer it: the name "f" inside `AcceptFoo()`.

Contrast this with `DoesNotBuild()`: the unique pointer allocated with `NewFoo()` has two names which refer to it: `DoesNotBuild()`'s "g" and `AcceptFoo()`'s "f".

This is the classic uniqueness violation: at any given point in the execution, any value held by a `std::unique_ptr` (or more generally, any move-only type) can only be referred to by a single distinct name. Anything that looks like a copy introducing an additional name is forbidden and won't compile:

```
scratch.cc: error: call to deleted constructor of std::unique_ptr<Foo>'
  AcceptFoo(g);
```

Even if the compiler doesn't catch you, the runtime behavior of `std::unique_ptr` will. Any time where you "outsmart" the compiler (see `SmarterThanTheCompilerButNot()`) and introduce multiple `std::unique_ptr` names, it may compile (for now) but you'll get a run-time memory problem.

Now the question becomes: how do we remove a name? C++11 provides a solution for that as well, in the form of `std::move()`.

```
void EraseTheName() {
  std::unique_ptr<Foo> h = NewFoo();
  AcceptFoo(std::move(h)); // Fixes DoesNotBuild with std::move
}
```

The call to `std::move()` is effectively a name-eraser: conceptually you can stop counting "h" as a name for the pointer value. This now passes the distinct-names rule: on the unique pointer allocated with `NewFoo()` has a single name ("h"), and within the call to `AcceptFoo()` there is again only a single name ("f"). By using `std::move()` we promise that we will not read from "h" again until we assign a new value to it.

Name counting is a handy trick in modern C++ for those that aren't expert in the subtleties of lvalues, rvalues, etc: it can help you recognize the possibility of unnecessary copies, and it will help you use `std::unique_ptr` properly. After counting, if you discover a point where there are too many names, use `std::move` to erase the no-longer-necessary name.

Appendix H
Tip of the Week #65: Putting Things in their Place

Hyrum Wright (hyrum@hyrumwright.org)

Let me 'splain. No, there is too much. Let me sum up.

—Inigo Montoya

C++11 added a new way to insert elements into standard containers: the `emplace()` family of methods. These methods create an object directly within a container, instead of creating a temporary object and then copying or moving that object into the container. Avoiding these copies is more efficient for almost all objects, and makes it easier to store move-only objects (such as `std::unique_ptr`) in standard containers.

H.1 The Old Way and the New Way

Let's look at a simple example using vectors to contrast the two styles. The first example uses pre-C++11 code:

```
class Foo {
 public:
  Foo(int x, int y);
  ...
};

void addFoo() {
  std::vector<Foo> v1;
  v1.push_back(Foo(1, 2));
}
```

Using the older `push_back()` method, two `Foo` objects are constructed: the temporary argument and the object in the vector that is move-constructed from the temporary.

We can instead use C++11's `emplace_back()` and only one object will be constructed directly within the memory of the vector. Since the "emplace" family of functions forward their arguments to the underlying object's constructor, we can provide the constructor arguments directly, obviating the need to create a temporary `Foo`:

```
void addBetterFoo() {
  std::vector<Foo> v2;
  v2.emplace_back(1, 2);
}
```

H.2 Using Emplace Methods for Move-Only Operations

So far, we've looked at cases where emplace methods improve performance, but they also make previously impossible code feasible, such as storing move-only types like `std::unique_ptr` within containers. Consider this snippet:

```
std::vector<std::unique_ptr<Foo>> v1;
```

How would you insert values into this vector? One way would be to use `push_back()` and construct the value directly within its argument:

```
v1.push_back(std::unique_ptr<Foo>(new Foo(1, 2)));
```

This syntax works, but can be a bit unwieldy. Unfortunately, the traditional way of getting around this confusion is fraught with complexity:

```
Foo *f2 = new Foo(1, 2);
v1.push_back(std::unique_ptr<Foo>(f2));
```

This code compiles, but it leaves ownership of the raw pointer unclear until the insertion. What's worse, the vector now owns the object, but `f2` still remains valid, and could accidentally be deleted later on. To an uninformed reader, this ownership pattern can be confusing, particularly if construction and insertion are not sequential events as above.

Other solutions won't even compile, because `unique_ptr` isn't copyable:

```
std::unique_ptr<Foo> f(new Foo(1, 2));
v1.push_back(f);             // Does not compile!
v1.push_back(new Foo(1, 2)); // Does not compile!
```

Using emplace methods can make it more intuitive to insert the object while it's being created. In other cases, if you need to move the `unique_ptr` into the vector, you can:

```
std::unique_ptr<Foo> f(new Foo(1, 2));
v1.emplace_back(new Foo(1, 2));
v1.push_back(std::move(f));
```

By combining emplace with a standard iterator, you can also insert the object at an arbitrary location in the vector:

```
v1.emplace(v1.begin(), new Foo(1, 2));
```

That said, in practical terms we wouldn't want to see these ways to construct a `unique_ptr` - Use `std::make_unique` (from C++14) or `absl::make_unique` (if you're still on C++11).

H.3 Conclusion

We've used vector as an example in this Tip, but emplace methods are also available for maps, lists and other STL containers. When combined with `unique_ptr`, emplace allows for good encapsulation and makes the ownership semantics of heap-allocated objects clear in ways that weren't possible before. Hopefully this has given you a feel for the power of the new emplace family of container methods, and a desire to use them where appropriate in your own code.

Appendix I
Tip of the Week #77: Temporaries, Moves, and Copies

Titus Winters (titus@google.com)

In the ongoing attempt to figure out how to explain to non-language-lawyers how C++11 changed things, we present yet another entry in the series "When are copies made?" This is part of a general attempt to simplify the subtle rules that have surrounded copies in C++ and replace it with a simpler set of rules.

I.1 Can You Count to 2?

You can? Awesome. Remember that the "name rule" means that each unique name you can assign to a certain resource affects how many copies of the object are in circulation. (See TotW #55 on Name Counting for a refresher.)

I.2 Name Counting, in Brief

If you're worrying about a copy being created, presumably you're worried about some line of code in particular. So, look at that point. How many names exist for the data you think is being copied? There are only 3 cases to consider:

I.3 Two Names: It's a Copy

This one is easy: if you're giving a second name to the same data, it's a copy.

```
std::vector<int> foo;
FillAVectorOfIntsByOutputParameterSoNobodyThinksAboutCopies(&foo);
std::vector<int> bar = foo;     // Yep, this is a copy.

std::map<int, string> my_map;
string forty_two = "42";
my_map[5] = forty_two;          // Also a copy: my_map[5] counts as a name.
```

I.4 One Name: It's a Move

This one is a little surprising: C++11 recognizes that if you can't refer to a name anymore, you also don't care about that data anymore. The language had to be careful not to break cases where you were relying on the destructor (say, `absl::MutexLock`), so `return` is the easy case to identify.

```
std::vector<int> GetSomeInts() {
  std::vector<int> ret = {1, 2, 3, 4};
  return ret;
}

// Just a move: either "ret" or "foo" has the data, but never both at once.
std::vector<int> foo = GetSomeInts();
```

The other way to tell the compiler that you're done with a name (the "name eraser" from TotW #55) is calling `std::move()`.

```
std::vector<int> foo = GetSomeInts();
// Not a copy, move allows the compiler to treat foo as a
// temporary, so this is invoking the move constructor for
// std::vector<int>.
// Note that it isn't the call to std::move that does the moving,
// it's the constructor. The call to std::move just allows foo to
// be treated as a temporary (rather than as an object with a name).
std::vector<int> bar = std::move(foo);
```

I.5 Zero Names: It's a Temporary

Temporaries are also special: if you want to avoid copies, avoid providing names to variables.

```
void OperatesOnVector(const std::vector<int>& v);

// No copies: the values in the vector returned by GetSomeInts()
// will be moved (O(1)) into the temporary constructed between these
// calls and passed by reference into OperatesOnVector().
OperatesOnVector(GetSomeInts());
```

I.6 Beware: Zombies

The above (other than `std::move()` itself) is hopefully pretty intuitive, it's just that we all built up weird notions of copies in the years pre-dating C++11. For a language without garbage collection, this type of accounting gives us an excellent mix of performance and clarity. However, it's not without dangers, and the big one is this: what is left in a value after it has been moved from?

```
T bar = std::move(foo);
CHECK(foo.empty()); // Is this valid? Maybe, but don't count on it.
```

This is one of the major difficulties: what can we say about these leftover values? For most standard library types, such a value is left in a "valid but unspecified state". Non-standard types usually hold to the same rule. The safe approach is to stay away from these objects: you are allowed to re-assign to them, or let them go out of scope, but don't make any other assumptions about their state.

Clang-tidy provides some some static-checking to catch use-after move with the misc-use-after-move check. However, static-analysis won't ever be able to catch all of these - be on the lookout. Call these out in code review, and avoid them in your own code. Stay away from the zombies.

I.7 Wait, `std::move` Doesn't Move?

Yeah, one other thing to watch for is that a call to `std::move()` isn't actually a move itself, it's just a cast to an rvalue-reference. It's only the use of that reference by a move constructor or move assignment that does the work.

```
std::vector<int> foo = GetSomeInts();
std::move(foo); // Does nothing.
// Invokes std::vector<int>'s move-constructor.
std::vector<int> bar = std::move(foo);
```

This should almost never happen, and you probably shouldn't waste a lot of mental storage on it. I really only mention it if the connection between `std::move()` and a move constructor was confusing you.

I.8 Aaaagh! It's All Complicated! Why!?!

First: it's really not so bad. Since we have move operations in the majority of our value types (including protobufs), we can do away with all of the discussions of "Is this a copy? Is this efficient?" and just rely on name counting: two names, a copy. Fewer than that: no copy.

Ignoring the issue of copies, value semantics are clearer and simpler to reason about. Consider these two operations:

```
void Foo(std::vector<string>* paths) {
  ExpandGlob(GenerateGlob(), paths);
}

std::vector<string> Bar() {
  std::vector<string> paths;
  ExpandGlob(GenerateGlob(), &paths);
  return paths;
}
```

re these the same? What about if there is existing data in `*paths`? How can you tell? Value semantics are easier for a reader to reason about than input/output parameters, where you need to think about (and document) what happens to existing data, and potentially whether there is an pointer ownership transfer.

Because of the simpler guarantees about lifetime and usage when dealing with values (instead of pointers), it is easier for the compiler's optimizers to operate on code in that style. Well-managed value semantics also minimizes hits on the allocator (which is cheap but not free). Once we understand how move semantics help rid us of copies, the compiler's optimizers can better reason about object types, lifetimes, virtual dispatch, and a host of other issues that help generate more efficient machine code.

Since most utility code is now move-aware, we should stop worrying about copies and pointer semantics, and focus on writing simple easy-to-follow code. Please make sure you understand the new rules: not all legacy interfaces you encounter may be updated to return by value (instead of by output parameter), so there will always be a mix of styles. It's important that you understand when one is more appropriate than the other.

Appendix J
Tip of the Week #86: Enumerating with Class

Bradley White (bww@google.com)

Show class, … and display character.

—Bear Bryant

An enumeration, or simply an `enum`, is a type that can hold one of a specified set of integers. Some values of this set can be given names, and are called the enumerators.

J.1 Unscoped Enumerations

This concept will be familiar to C++ programmers, but prior to C++11 enumerations had two significant shortcomings: the enumeration names were:

- in the same scope as the enum type, and
- implicitly convertible to values of some integer type.

So, with C++98 …

```
enum CursorDirection { kLeft, kRight, kUp, kDown };
CursorDirection d = kLeft; // OK: enumerator in scope
int i = kRight;            // OK: enumerator converts to int
```

but, …

```
// error: redeclarations of kLeft and kRight
enum PoliticalOrientation { kLeft, kCenter, kRight };
```

C++11 modified the behavior of unscoped enums in one way: the enumerators are now local to the enum, but continue to be exported into the enum's scope for backwards compatibility.

So, with C++11 …

```
CursorDirection d = CursorDirection::kLeft;  // OK in C++11
int i = CursorDirection::kRight;             // OK: still converts to int
```

but the declaration of `PoliticalOrientation` would still elicit errors.

J.2 Scoped Enumerations

The implicit conversion to integer has been observed to be a common source of bugs, while the namespace pollution caused by having the enumerators in the same scope as the enum causes problems in large, multi-library projects. To address both these concerns, C++11 introduced a new concept: the *scoped enum*.

In a scoped enum, introduced by the keywords `enum class`, the enumerators are:

- only local to the enum (they are not exported into the enum's scope), and
- not implicitly convertible to integer types.

So, (note the additional class keyword) …

```
enum class CursorDirection { kLeft, kRight, kUp, kDown };
CursorDirection d = kLeft;                    // error: kLeft not in this scope
CursorDirection d2 = CursorDirection::kLeft;  // OK
int i = CursorDirection::kRight;              // error: no conversion
```

and, …

```
// OK: kLeft and kRight are local to each scoped enum
enum class PoliticalOrientation { kLeft, kCenter, kRight };
```

These simple changes eliminate the problems with plain enumerations, so enum class should be preferred in all new code.

Using a scoped enum does mean that you'll have to explicitly cast to an integer type should you still want such a conversion (e.g., when logging an enumeration value, or when using bitwise operations on flag-like enumerators). Hashing with `std::hash` will continue to work though (e.g., `std::unordered_map<CursorDirection, int`

J.3 Underlying Enumeration Types

C++11 also introduced the ability to specify the underlying type for both varieties of enumeration. Previously the underlying integer type of an enum was determined by examining the sign and magnitude of the enumerators, but now we can be explicit. For example, …

```
// Use "int" as the underlying type for CursorDirection
enum class CursorDirection : int { kLeft, kRight, kUp, kDown };
```

Because this enumerator range is small, and if we wished to avoid wasting space when storing `CursorDirection` values, we could specify `char` instead.

```
// Use "char" as the underlying type for CursorDirection
enum class CursorDirection : char { kLeft, kRight, kUp, kDown };
```

The compiler will issue an error if an enumerator value exceeds the range of the underlying type.

J.4 Conclusion

Prefer using `enum class` in new code. You'll reduce namespace pollution, and you may avoid bugs in implicit conversions.

```
// Use "char" as the underlying type for CursorDirection
enum class Parting { kSoLong, kFarewell, kAufWiedersehen, kAdieu };
```

Appendix K
Tip of the Week #109: Meaningful `const` in Function Declarations

GREG MILLER (JGM@GOOGLE.COM)

This document will explain when `const` is meaningful in function declarations, and when it is meaningless and best omitted. But first, let us briefly explain what is meant by the terms *declaration* and *definition*.

Consider the following code:

```
1 void F(int);                      // 1: declaration of F(int)
2 void F(const int);                // 2: re-declaration of F(int)
3 void F(int) { /* ... */ }         // 3: definition of F(int)
4 void F(const int) { /* ... */ }   // 4: error: re-definition of F(int)
```

The first two lines are function *declaration*s. A function *declaration* tells the compiler the function's signature and return type. In the above example, the function's signature is `F(int)`. The constness of the function's parameter type is ignored, so both declarations are equivalent (See Overloadable declarations.)

Lines 3 and 4 from the above code are both function *definition*s. A function *definition* is also a declaration, but a definition also contains the function's body. Therefore, line 3 is a definition for a function with the signature `F(int)`. Similarly, line 4 is also a definition for the same function, `F(int)`, which will result in an error at link time. Multiple declarations are allowed, but only a single definition is permitted.

Even though the definitions on lines 3 and 4 *declare* and *define* the same function, there is a difference within their function bodies due to the way they are declared. From the definition on line 3, the type of the function-parameter variable within the function will be `int` (i.e., non-const). On the other hand, the definition on line 4 will produce a function-parameter variable within the function whose type is `const int`.

K.1 Meaningful `const` in Function Declarations

Not all `const` qualifications in function declarations are ignored. To quote from Overloadable declarations ([over.load]) in the C++ standard (emphasis added):

"const *type-specifiers* ***buried within a parameter type specification*** *are significant and can be used to distinguish overloaded function declarations*"

The following are examples where `const` is significant and not ignored:

```
void F(const int* x);                  // 1
void F(const int& x);                  // 2
void F(std::unique_ptr<const int> x);  // 3
void F(int* x);                        // 4
```

In the above examples, the `x` parameter itself is never declared `const`. Each of the above functions accepts a parameter named `x` of a different type, thus forming a valid overload set. Line 1 declares a function that accepts a "pointer to an `int` that is `const`". Line 2 declares a function that accepts a "reference to an `int` that is `const`". And line 3 declares a function that accepts a "`std::unique_ptr` to an `int` that is `const`". All of these uses of `const` are important and not ignored because they are part of the parameter type specification and are not top-level `const` qualifications that affect the parameter `x` itself.

Line 4 is interesting because it does not include the `const` keyword at all, and may at first appear to be equivalent to the declaration on line 1 given the reasons cited at the beginning of this document. The reason that this is not true and that line 4 is a valid and distinct declaration is that only top-level, or outermost, `const` qualifications of the parameter type specification are ignored.

To complete this example, let us look at a few more examples where a const is meaningless and ignored.

```
void F(const int x);          // 1: declares F(int)
void F(int* const x);         // 2: declares F(int*)
void F(const int* const x);   // 3: declares F(const int*)
```

K.2 Rules of Thumb

Though few of us will ever truly master all the delightful obscurities of C++, it is important that we do our best to understand the rules of the game. This will help us write code that is understood by other C++ programmers who are following the same rules and playing the same game. For this reason, it is important that we understand when `const` qualification is meaningful in a function declaration and when it is ignored.

Although there is no official guidance from the Google C++ style guide, and there is no single generally accepted opinion, the following is one reasonable set of guidelines:

- Never use top-level `const` on function parameters in *declaration*s that are not definitions (and be careful not to copy/paste a meaningless `const`). It is meaningless and ignored by the compiler, it is visual noise, and it could mislead readers.
- Do use top-level `const` on function parameters in *definition*s at your (or your team's) discretion. You might follow the same rationale as you would for when to declare a function-local variable `const`.

Appendix L

Tip of the Week #112: `emplace` vs. `push_back`

GEOFF ROMER (GROMER@GOOGLE.COM)

The less we use our power, the greater it will be.

—Thomas Jefferson

As you may already know (and if not, TotW #65), C++11 introduced a powerful new way to insert items into containers: `emplace` methods. These let you construct an object in-place inside the container, using any of the object's constructors. That includes the move and copy constructors, so it turns out any time you could use a `push` or `insert` method, you can use an `emplace` method instead, with no other changes:

```
std::vector<string> my_vec;
my_vec.push_back("foo");     // This is OK, so...
my_vec.emplace_back("foo");  // This is also OK, and has the same result

std::set<string> my_set;
my_set.insert("foo");        // Same here: any insert call can be
my_set.emplace("foo");       // rewritten as an emplace call.
```

This raises an obvious question: which one should you use? Should we perhaps just discard `push_back()` and `insert()`, and use `emplace` methods all the time?

Let me answer that question by asking another: what do these two lines of code do?

```
vec1.push_back(1<<20);
vec2.emplace_back(1<<20);
```

The first line is quite straightforward: it adds the number 1048576 to the end of the vector. The second, however, is not so clear. Without knowing the type of the vector, we don't know what constructor it's invoking, so we can't really say what that line is doing. For example, if `vec2` is a `std::vector<int>`, that line merely adds 1048576 to the end, as with the first line, but if `vec2` is a `std::vector<std::vector<int>>`, that second line constructs a `vector` of over a million elements, allocating several megabytes of memory in the process.

Consequently, if you have a choice between `push_back()` and `emplace_back()` with the same arguments, your code will tend to be more readable if you choose `push_back()`, because `push_back()` expresses your intent more specifically. Choosing `push_back()` is also safer: suppose you have a `std::vector<std::vector<int` and you want to append a number to the end of the first `vector`, but you accidentally forget the subscript. If you write `my_vec.push_back(2<<20)`, you'll get a compile error and you'll quickly spot the problem. On the other hand, if you write `my_vec.emplace_back(2<<20)`, the code will compile, and you won't notice any problems until run-time.

Now, it's true that when implicit conversions are involved, `emplace_back()` can be somewhat faster than `push_back()`. For example, in the code that we began with, `my_vec.push_back("foo")` constructs a temporary `string` from the string literal, and then moves that string into the container, whereas `my_vec.emplace_back("foo` just constructs the `string` directly in the container, avoiding the extra move. For more expensive types, this may be a reason to use `emplace_back()` instead of `push_back()`, despite the readability and safety costs, but then again it may not. Very often the performance difference just won't matter. As always, the rule of thumb is that you should avoid "optimizations" that make the code less safe or less clear, unless the performance benefit is big enough to show up in your application benchmarks.

So in general, if both `push_back()` and `emplace_back()` would work with the same arguments, you should prefer `push_back()`, and likewise for `insert()` vs. `emplace()`.

Appendix M
Tip of the Week #117: Copy Elision and Pass-by-value

Geoff Romer, (gromer@google.com)

Everything is so far away, a copy of a copy of a copy. The insomnia distance of everything, you can't touch anything and nothing can touch you.

—Chuck Palahniuk

Suppose you have a class like this:

```
class Widget {
 public:
  ...

 private:
  string name_;
};
```

How would you write its constructor? For years, the answer has been to do it like this:

```
// First constructor version
explicit Widget(const std::string& name) : name_(name) {}
```

However, there is an alternative approach which is becoming more common:

```
// Second constructor version
explicit Widget(std::string name) : name_(std::move(name)) {}
```

(If you're not familiar with `std::move()`, see TotW #77, or pretend I used `std::swap` instead; the same principles apply). What's going on here? Isn't it horribly expensive to pass `std::string` by copy? It turns out that no, sometimes passing by value (as we'll see, it's not really "by copy") can be much more efficient than passing by reference.

To understand why, consider what happens when the callsite looks like this:

```
Widget widget(absl::StrCat(bar, baz));
```

With the first version of the `Widget` constructor, `absl::StrCat()` produces a temporary string containing the concatenated string values, which is passed by reference into `Widget()`, and then the string is copied into `name_`. With the second version of the Widget constructor, the temporary string is passed into `Widget()` by value, which you might think would cause the string to be copied, but this is where the magic happens. When the compiler sees a temporary being used to copy-construct an object, the compiler will[1] simply use the same storage for both the temporary and the new object, so that copying from the one to the other is literally free; this is called *copy elision*. Thanks to this optimization, the string is never copied, but only moved once, which is a cheap constant-time operation.

Consider what happens when the argument isn't a temporary:

```
string local_str;
Widget widget(local_str);
```

In this case, both versions of the code make a copy of the string, but the second version also moves the string. That, again, is a cheap constant-time operation, whereas copying is a linear-time operation, so in many cases that will be a price well worth paying.

The key property of the `name` parameter that makes this technique work is that `name` has to be copied. Indeed, the essence of this technique is to try to make the copy operation happen on the function call boundary, where it can be elided, rather than in the interior of the function. This need not necessarily involve `std::move();` for example, if the function needs to mutate the copy rather than store it, it may just be mutated in place.

M.1 When to Use Copy Elision

Passing parameters by value has several drawbacks which should be borne in mind. First of all, it makes the function body more complex, which creates a maintenance and readability burden. For example, in the code above, we've added a `std::move()` call, which creates a risk of accidentally accessing a moved-from value. In this function that risk is pretty minimal, but if the function were more complex, the risk would be higher.

Second, it can degrade performance, sometimes in surprising ways. It can sometimes be difficult to tell whether it's a net performance win without profiling a specific workload.

- As stated above, this technique applies only to parameters that need to be copied; it will be useless at best and harmful at worst when applied to parameters that don't need to be copied, or only need to be copied conditionally.
- This technique often involves some amount of extra work in the function body, such as the move assignment in the example above. If that extra work imposes too much overhead, the slowdown in the cases where the copy can't be elided may not be worth the speedup in the cases where the copy can be elided. Note that this determination can depend on the particulars of your use case. For example, if the arguments to `Widget()` are almost always very short, or are almost never temporaries, this technique may be harmful on balance. As always when considering optimization tradeoffs, when in doubt, measure.
- When the copy in question is a *copy assignment* (for example if we wanted to add a `set_name()` method to Widget), the pass-by-reference version can sometimes avoid memory allocation by reusing `name_`'s existing buffer, in cases where the pass-by-value version would allocate new memory. In addition, the fact that pass-by-value always replaces `name_`'s allocation can lead to worse allocation behavior down the line: if the `name_` field tends to grow over time after it is set, that growth will require further new

[1]Strictly speaking the compiler is not required to perform copy elision, but it is such a powerful and vitally important optimization that it is extremely unlikely that you will ever encounter a compiler that doesn't do it.

allocations in the pass-by-value case, whereas in the pass-by-reference case we only reallocate when the `name_` field exceeds its historical maximum size.

Generally speaking, you should prefer simpler, safer, more readable code, and only go for something more complex if you have concrete evidence that the complex version performs better and that the difference matters. That principle certainly applies to this technique: passing by const reference is simpler and safer, so it's still a good default choice. However, if you're working in an area that's known to be performance-sensitive, or your benchmarks show that you're spending too much time copying function parameters, passing by value can be a very useful tool to have in your toolbox.

Appendix N
Tip of the Week #123: `absl::optional` and `std::unique_ptr`

Alexey Sokolov (sokolov@google.com) & Etienne Dechamps (edechamps@google.com)

N.1 How to Store Values

This tip discusses several ways of storing values. Here we use class member variables as an example, but many of the points below also apply to local variables.

```cpp
#include <memory>
#include "absl/types/optional.h"
#include ".../bar.h"

class Foo {
  ...
 private:
  Bar val_;
  absl::optional<Bar> opt_;
  std::unique_ptr<Bar> ptr_;
};
```

N.1.0.1 As a Bare Object

This is the simplest way. `val_` is constructed and destroyed at the beginning of `Foo`'s constructor and at the end of `Foo`'s destructor, respectively. If `Bar` has a default constructor, it doesn't even need to be initialized explicitly.

`val_` is very safe to use, because its value can't be null. This removes a class of potential bugs.

But bare objects are not very flexible:

- The lifetime of `val_` is fundamentally tied to the lifetime of its parent `Foo` object, which is sometimes not desirable. If `Bar` supports move or swap operations, the contents of `val_` can be replaced using these operations, while any existing pointers or references to `val_` continue pointing or referring to the same `val_` object (as a container), not to the value stored in it.
- Any arguments that need to be passed to `Bar`'s constructor need to be computed inside the initializer list of `Foo`'s constructor, which can be difficult if complicated expressions are involved.

N.1.0.2 As `absl::optional<Bar>`

This is a good middle ground between the simplicity of bare objects and the flexibility of `std::unique_ptr`. The object is stored inside `Foo` but, unlike bare objects, `absl::optional` can be empty. It can be populated at any time by assignment (`opt_ = ...`) or by constructing the object in place (`opt_.emplace(...)`).

Because the object is stored inline, the usual caveats about allocating large objects on the stack apply, just like for a bare object. Also be aware that an empty `absl::optional` uses as much memory as a populated one.

Compared to a bare object, `absl::optional` has a few downsides:

- It's less obvious for the reader where object construction and destruction occur.
- There is a risk of accessing an object which does not exist.

N.1.0.3 As `std::unique_ptr<Bar>`

This is the most flexible way. The object is stored outside of `Foo`. Just like `absl::optional`, a `std::unique_ptr` can be empty. However, unlike absl::optional, it is possible to transfer ownership of the object to something else (through a move operation), to take ownership of the object from something else (at construction or through assignment), or to assume ownership of a raw pointer (at construction or through `ptr_ = absl::WrapUnique(...)`, see TotW #126.

When `std::unique_ptr` is `null`, it doesn't have the object allocated, and consumes only the size of a pointer[1].

Wrapping an object in a `std::unique_ptr` is necessary if the object may need to outlive the scope of the `std::unique_ptr` (ownership transfer).

This flexibility comes with some costs:

- Increased cognitive load on the reader:
  - It's less obvious what's stored inside (`Bar`, or something derived from `Bar`). However, it may also decrease the cognitive load, as the reader can focus only on the base interface held by the pointer.
  - It's even less obvious than with `absl::optional` where object construction and destruction occur, because ownership of the object can be transferred.
- As with `absl::optional`, there is a risk of accessing an object which does not exist - the famous null pointer dereference.
- The pointer introduces an additional level of indirection, which requires a heap allocation, and is not friendly to CPU caches; Whether this matters or not depends a lot on particular use cases.
- `std::unique_ptr<Bar>` is not copyable even if `Bar` is. This also prevents `Foo` from being copyable.

N.2 Conclusion

As always, strive to avoid unnecessary complexity, and use the simplest thing that works. Prefer bare object, if it works for your case. Otherwise, try `absl::optional`. As a last resort, use `std::unique_ptr`.

[1]In case of a non-empty custom deleter there is also an additional space for that deleter.

[2]Also padding may be added.

[3]Also padding may be added.

[4]Also padding may be added.

[5]Also padding may be added.

[6]Also padding may be added.

[7]Also padding may be added.

[8]Also padding may be added.

Strategy	Bar	`absl::optional<Bar>`	`std::unique_ptr<Bar>`
Supports delayed construction		✓	✓
Always safe to access	✓		
Can transfer ownership of Bar			✓
Can store subclasses of Bar			✓
Movable	If Bar is movable	If Bar is movable	✓
Copyable	If Bar is copyable	If Bar is copyable	
Friendly to CPU caches	✓	✓	
No heap allocation overhead	✓	✓	
Memory usage	`sizeof(Bar)`	`sizeof(Bar)` + `sizeof(bool)` [8]	`sizeof(Bar*)` when `null`, `sizeof(Bar*)` + `sizeof(Bar)` otherwise
Object lifetime	Same as enclosing scope	Restricted to enclosing scope	Unrestricted
Call f(Bar*)	`f(&val_)`	`f(&opt_.value())` or `f(&*opt_)`	`f(ptr_.get())` or `f(&*ptr_)`
Remove value	N/A	`opt_.reset();` or `opt_ = absl::nullopt;`	`ptr_.reset();` or `ptr_ = nullptr;`

Appendix O

Tip of the Week #126: `std::make_unique` is the new `new`

James Dennett (jdennett@google.com) & Titus Winters (titus@google.com)

As a codebase expands it is increasingly difficult to know the details of everything you depend on. Requiring deep knowledge doesn't scale: we have to rely on interfaces and contracts to know that code is correct, both when writing and when reviewing. In many cases the type system can provide those contracts in a common fashion. Consistent use of type system contracts makes for easier authoring and reviewing of code by identifying places where there are potentially risky allocations or ownership transfers for objects allocated on the heap.

While in C++ we can reduce the need for dynamic memory allocation by using plain values, sometimes we need objects to outlive their scope. C++ code should prefer smart pointers (most commonly `std::unique_ptr`) instead of raw pointers when dynamically allocating objects. This provides a consistent story around allocation and ownership transfer, and leaves a clearer visual signal where there's code that needs closer inspection for ownership issues. The side effect of matching how allocation works in the outside world post-C++14 and being exception safe is just icing.

Two key tools for this are `absl::make_unique()` (a C++11 implementation of C++14's `std::make_unique()`, for leak-free dynamic allocation) and `absl::WrapUnique()` (for wrapping owned raw pointers into the corresponding `std::unique_ptr` types). They can be found in absl/memory/memory.h.

O.1 Why Avoid `new`?

Why should code prefer smart pointers and these allocation functions over raw pointers and `new`?

- When possible, ownership is best expressed in the type system. This allows reviewers to verify correctness (absence of leaks and of double-deletes) almost entirely by local inspection. (In code that is exceptionally performance sensitive, this may be excused: while cheap, passing `std::unique_ptr` across function boundaries by value has non-zero overhead because of ABI constraints. That's rarely important enough to justify avoiding it.)
- Somewhat like the reasoning for preferring `push_back()` over `emplace_back()` (TotW #112), `absl::make_uniq` directly expresses the intent and can only do one thing (do the allocation with a public constructor, returning a `std::unique_ptr` of the specified type). There's no type conversion or hidden behavior. `absl::make_unique()` does what it says on the tin.
- The same could be achieved with `std::unique_ptr<T> my_t(new T(args));` but that is redundant (repeating the type name T) and for some people there's value in minimizing calls to `new`. More on this in #5.
- If all allocations are handled via `absl::make_unique()` or factory calls, that leaves `absl::WrapUnique()` for the implementation of those factory calls, for code interacting with legacy methods that don't rely on `std::unique_ptr` for ownership transfer, and for rare cases that need to dynamically allocate with aggregate initialization (`absl::WrapUnique(new MyStruct{3.141, "pi"})`). In code review it's easy to spot the `absl::WrapUnique` calls and evaluate "does that expression look like an ownership transfer?" Usually it's obvious (for example, it's some factory function). When it's not obvious, we need to check the function to be sure that it's actually a raw-pointer ownership transfer.

- If we instead rely mostly on the constructors of `std::unique_ptr`, we see calls like:

```
std::unique_ptr<T> foo(Blah());
std::unique_ptr<T> bar(new T());
```

It takes only a moment's inspection to see that the latter is safe (no leak, no double-delete). The former? It depends: if `Blah()` is returning a `std::unique_ptr`, it's fine, though in that case it would be more obviously safe if written as `std::unique_ptr<T> foo = Blah();` If `Blah()` is returning an ownership-transferred raw pointer, that's also fine. If `Blah()` is returning just some random pointer (no transfer), then there's a problem. Reliance on `absl::make_unique()` and `absl::WrapUnique()` (avoiding the constructors) provides an additional visual clue for the places where we have to worry (calls to `absl::WrapUnique()`, and only those).

O.2 How Should We Choose Which to Use?

- By default, use `absl::make_unique()` (or `std::make_shared()` for the rare cases where shared ownership is appropriate) for dynamic allocation. For example, instead of: `std::unique_ptr<T> bar(new T());` write `auto bar = absl::make_unique<T>();` and instead of `bar.reset(new T());` write `bar = absl::ma`
- In a factory function that uses a non-public constructor, return a `std::unique_ptr<T>` and use `absl::WrapUnique` in the implementation.
- When dynamically allocating an object that requires brace initialization (typically a struct, an array, or a container), use `absl::WrapUnique(new T{...})`.
- When calling a legacy API that accepts ownership via a `T*`, either allocate the object in advance with `absl::make_unique` and call `ptr.release()` in the call, or use `new` directly in the function argument.
- When calling a legacy API that returns ownership via a `T*`, immediately construct a smart pointer with `WrapUnique` (unless you're immediately passing the pointer to another legacy API that accepts ownership via a `T*`).

O.3 Summary

Prefer `absl::make_unique()` over `absl::WrapUnique()`, and prefer `absl::WrapUnique()` over raw `new`.

Appendix P
Tip of the Week #130: Namespace Naming

Titus Winters (titus@google.com)

The precision of naming takes away from the uniqueness of seeing

—Pierre Bonnard

The earliest commit of the Google C++ Style Guide contains the guidance that many people are still using for namespace naming. Roughly, this can be summarized as “namespaces are derived from package paths”. Following on the heels of Java’s package naming requirements, this makes a lot of sense: we want to be able to uniquely identify symbols in C++ and we want there to be uniqueness and consistency in namespace choice.

Except in actuality, we don’t. We just didn’t realize for almost a decade.

P.1 Name Lookup

Let’s start with how name lookup works in C++ and how it’s different from Java.

```
namespace foo {
namespace bar {
void f() {
  Baz b;
}
}
}
```

In C++, lookup on an unqualified name (`Baz`) will search expanding scopes for a symbol of the same name: first in `f()` (the function), then in `bar`, then in `foo`, then in the global namespace.

In Java, there is no such thing as an unqualified symbol: either a symbol is a qualified name:

```
public void f() {
  com.google.foo.bar.Baz b = new com.google.foo.bar.Baz();
}
```

Or it is imported, either as a single package member or via wildcard:

```
import com.google.foo.bar.Baz;
import com.google.foo.bar.*;
```

In no case is `Baz` looked for outside of the package that is explicitly provided: wildcards don’t descend into child packages, nor is search extended into parent packages. As it turns out, this difference in how parent packages/namespaces are handled within Java and C++ is fundamental to why structural namespace naming (making the namespace structure match the package hierarchy) is a mistake within C++.

P.2 The Problem

The fundamental problem for building namespaces out of packages is that we rarely rely on fully-qualified lookup in C++, normally writing `std::unique_ptr` rather than `::std::unique_ptr`. Coupled with lookup in enclosing namespaces, this means that for code in a deeply nested package (`::division::section::team::subteam::`
for example) any symbol that is not fully qualified (`std::unique_ptr`) can in fact reference any of

- `::std::unique_ptr`
- `::std::unique_ptr`
- `::division::std::unique_ptr`
- `::division::section::std::unique_ptr`
- `::division::section::team::std::unique_ptr`
- `::division::section::team::subteam::std::unique_ptr`
- `::division::section::team::subteam::project::std::unique_ptr`

And what's worse: unqualified search starts at the bottom of that list *and stops as soon as there is a namespace match*. This means that your build can be broken if any of your transitive includes add a previously unused namespace that matches the leading namespace of any symbol you use out of an unqualified namespace. Strictly speaking, this doesn't even have to be a build break: if someone adds something with a matching name and a syntactically-compatible API, the implementation of that API may be completely incompatible and cause widespread havoc at runtime. Obviously this isn't too bad with `std` - nobody should ever be adding a nested namespace `std` - but what about more common namespaces? How about things like `testing`?

Names aren't chosen to be unique. Since teams commonly create local utility packages to handle common tasks relating to the infrastructure they rely on, we wind up with local `util` and `pipeline` packages - and sub-namespaces. This is a recipe for unnecessary and unintended collisions.

For comparison, the problem in Java is far reduced: if you wildcard-import from two packages in Java and one adds a new symbol with the same name as the other package, your build can break. This is easily and completely solved by forbidding wildcard imports as is done in many Java styles.

P.3 Two Consistent Options, Three Approaches

There are two features that prevent this build-break-at-a-distance:

- If no leaf namespace (`search::foo::bar`) matches any top-level namespace (`::bar`) or a sub-namespace of any parent of that leaf (`search::bar`), no name collisions will occur.
- If there are no unqualified lookups, there will be no problems.

There are (at least) three ways to achieve this:

- Always fully qualify everything outside of the current namespace. This is very verbose and sort of weird: nothing in C++ (including the standard library) is written with leading `::` on every symbol.
- Build some tooling to identify introduction of new namespaces and ensure that it doesn't overlap with any other namespace in the same hierarchy. That is, do not add `search::bar` if there is a `::bar` or a `search::foo::bar`.
- Don't nest deeply: a single top-level namespace per project gets the same result without long/complicated names, with less exposure to accidents, without causing surprise for new engineers, and without the need to build any tooling.

The current style guide suggests the last option, but allows for the old style (namespaces match package names) if necessary. This is largely because the Google didn't want to cause too much anxiety or trigger anyone re-namespacing things. That said, if we had it to do over again in a fresh codebase we would unambiguously say this: one top-level namespace for public interfaces per project. Ensure uniqueness of namespaces via a common database. Thus we get (only) top-level namespaces like `absl`, and can have no ambiguity in lookup (barring collision between local symbols and those in the global namespace, but modern rules discourage the global namespace anyway).

Because there is so much code that existed before this change, and so much code following the old pattern even after this change, we find ourselves in a sort of half-way space, with some namespaces that often need to be fully qualified (`::util`), and some that are obviously unique and never need to be (`std`).

P.4 But It Keeps Things Organized!

I regularly hear people express that small/nested namespaces "keep things organized". Putting things in their place feels right - why lump together something like `StrCat()` and `make_unique()` other than being in Abseil these have nothing to do with one another! Wouldn't an `absl::strings::utilities` namespace help differentiate from `absl::smart_ptrs`?

In other languages this would probably be good - better organization with no downside. However, because of how lookup works (expanding into successive layers of containing namespace scopes) your fine-grained namespace is impacted by every symbol (and sub-namespace) added in every parent namespace. That is: while you don't exactly "contain" the names from parent namespaces, name/namespace collisions matter nearly as much as if you do. Small/deeply-nested namespaces don't *shield* you from this, they *exacerbate* it.

P.5 Best Practices

Practically speaking, the following is the best we can do given the realities of most codebases:

- Have a database of some form for a codebase to identify the unique namespaces.
- When introducing a new namespace, use that database and introduce it as a top-level.
- If for some reason the above is impossible, never ever introduce a sub-namespace that matches a well-known top-level namespace. No sub-namespaces for `absl`, `testing`, `util`, etc. Try to give sub-namespaces unique names that are unlikely to collide with future top-levels.
- When declaring namespace aliases and using-declarations, use fully qualified names, unless you are referring to a name inside the current namespace, as per TotW #119.
- For code in util or other commonly-abused namespaces, try to avoid full qualification, but qualify if necessary.

The advice in TotW #119 also helps, for `.cc` files: our concern with fully-qualifying is not that it is bad, but that it is *weird* compared to C++ code in the rest of the world. Limited use in using-declarations strikes an acceptable balance. However, even complete adherence to this suggestion doesn't fully mitigate the dangers from unqualified name lookup because we still have header files and don't want to fully qualify every symbol in every header.

Appendix Q
Tip of the Week #140: Constants: Safe Idioms

Matt Armstrong

What are best ways to express constants in C++? You probably have an idea of what the word means in English, but it is easy to express the concept incorrectly in code. Here we'll first define a few key concepts, then get to a list of safe techniques. For the curious, we then go into more detail about what can go wrong, and describe a C++17 language feature that makes expressing constants easier.

There is no formal definition of a "C++ constant" so let's propose an informal one.

- **The value**: A value never changes; five is always five. When we want to express a constant, we need a value, but only one.
- **The object**: At each point in time an object has a value. C++ places a strong emphasis on mutable objects, but mutation of constants is disallowed.
- **The name**: Named constants are more useful than bare literal constants. Both variables and functions can evaluate to constant objects.

Putting that all together, let's call a constant a variable or function that always evaluates to the same value. There are a few more key concepts.

- **Safe Initialization**: Many times constants are expressed as values in static storage, which must be safely initialized. For more on that, see the C++ Style Guide.
- **Linkage**: Linkage has to do with how many instances (or "copies") of a named object there are in a program. It is usually best for a constant with one name to refer to a single object within the program. For global or namespace-scoped variables this requires something called external linkage (you can read more about linkage here).
- **Compile-time evaluation**: Sometimes the compiler can do a better job optimizing code if a constant's value is known at compile time. This benefit can sometimes justify defining the values of constants in header files, despite the additional complexity.

When we say we're "adding a constant" we're actually declaring an API and defining its implementation in such a way that satisfies most or all of the above criteria. The language doesn't dictate how we do this, and some ways are better than others. Often the simplest approach is declaring a `const` or `constexpr` variable, marked as `inline` if it's in a header file. Another approach is returning a value from a function, which is more flexible. We'll cover examples of both approaches.

A note on `const`: it isn't enough. A `const` object is read-only but this does not imply that it is *immutable* nor does it imply that the value is always the same. The language provides ways to mutate values we think of as `const`, such as the `mutable` keyword and `const_cast`. But even straightforward code can demonstrate the point:

```cpp
void f(const std::string& s) {
  const int size = s.size();
  std::cout << size << '\n';
}

```

```
f("");  // Prints 0
f("foo");  // Prints 3
```

In the above code `size` is a `const` variable, yet it holds multiple values as the program runs. It is not a constant.

Q.1 Constants in Header Files

All of the idioms in this section are robust and recommendable.

Q.1.0.1 An inline constexpr Variable

From C++17 variables can be marked as `inline`, ensuring that there is only a single copy of the variable. When used with `constexpr` to ensure safe initialization and destruction this gives another way to define a constant whose value is accessible at compile time.

```
// in foo.h
inline constexpr int kMyNumber = 42;
inline constexpr absl::string_view kMyString = "Hello";
```

Q.1.1 An extern const Variable

```
// Declared in foo.h
ABSL_CONST_INIT extern const int kMyNumber;
ABSL_CONST_INIT extern const char kMyString[];
ABSL_CONST_INIT extern const absl::string_view kMyStringView;
```

The above example **declares** *one* instance of each object. The `extern` keyword ensures external linkage. The `const` keyword helps prevent accidental mutation of the value. This is a fine way to go, though it does mean the compiler can't "see" the constant values. This limits their utility somewhat, but not in ways that matter for typical use cases. It also requires **defining** the variables in the associated `.cc` file.

```
// Defined in foo.cc
const int kMyNumber = 42;
const char kMyString[] = "Hello";
const absl::string_view kMyStringView = "Hello";
```

The `ABSL_CONST_INIT` macro ensures each constant is compile-time initialized, but that is all it does. It *does not* make the variable `const` and it *does not* prevent declarations of variables with non-trivial destructors that violate the style guide rules. See mention of the macro in the style guide.

You might be tempted to define the variables in the `.cc` file with `constexpr`, but this is not a portable approach at the moment.

NOTE: `absl::string_view` is a good way to declare a string constant. The type has a constexpr constructor and a trivial destructor, so it is safe to declare them as global variables. Because a string view knows its length, using them does not require a runtime call to `strlen()`.

Q.1.1.1 A constexpr Function

A `constexpr` function that takes no arguments will always return the same value, so it functions as a constant, and can often be used to initialize other constants at compile time. Because all `constexpr` functions are implicitly `inline`, there are no linkage concerns. The primary disadvantage of this approach is the limitations placed on the code in `constexpr` functions. Secondarily, `constexpr` is a non-trivial aspect of the API contract, which has real consequences .

```
// in foo.h
constexpr int MyNumber() { return 42; }
```

Q.1.2 An Ordinary Function

When a `constexpr` function isn't desirable or possible, an ordinary function may be an option. The function in the following example can't be `constexpr` because it has a static variable:

```
inline absl::string_view MyString() {
  static constexpr char kHello[] = "Hello";
  return kHello;
}
```

NOTE: make sure you use `static constexpr` specifiers when returning array data, such as `char[]` strings, `absl::string_view`, `absl::Span`, etc, to avoid subtle bugs.

Q.1.2.1 A `static` Class Member

Static members of a class are a good option, assuming you are already working with a class. These always have external linkage.

```
// Declared in foo.h
class Foo {
 public:
  static constexpr int kMyNumber = 42;
  static constexpr char kMyHello[] = "Hello";
};
```

Prior to C++17 it was necessary to also provide definitions for these `static` data members in a `.cc` file, but for data members that are both `static` and `constexpr` these are now unnecessary (and deprecated).

```
// Defined in foo.cc, prior to C++17.
constexpr int Foo::kMyNumber;
constexpr char Foo::kMyHello[];
```

It isn't worth introducing a class just to act as a scope for a bunch of constants. Use one of the other techniques instead.

Q.1.2.2 Discouraged Alternatives

```
#define WHATEVER_VALUE 42
```

Using the preprocessor is rarely justified, see the style guide.

```
enum : int { kMyNumber = 42 };
```

The enum technique used above can be justified in some circumstances. It produces a constant `kMyNumber` that cannot cause the problems talked about in this tip. But the alternatives already listed will be more familiar to most people, and so are generally preferred. Use an `enum` when it makes sense in its own right (for examples, see Tip #86 "Enumerating with Class").

Q.1.3 Approaches that Work in Source Files

All of the approaches described above also work within a single `.cc` file, but may be unnecessarily complex. Because constants declared within a source file are visible only inside that file by default (see internal linkage rules), simpler approaches, such as defining `constexpr` variables, often work:

```
// within a .cc file!
constexpr int kBufferSize = 42;
constexpr char kBufferName[] = "example";
constexpr absl::string_view kOtherBufferName = "other example";
```

The above is fine in a `.cc` file but not a header file (see caveat). Read that again and commit it to memory. I'll explain why soon. Long story short: define variables `constexpr` in `.cc` files or declare them `extern const` in header files.

Q.1.3.1 Within a Header File, Beware!

Unless you take care to use idioms explained above, `const` and `constexpr` objects are likely to be *different* objects in each translation unit.

This implies:

- **Bugs**: any code that uses the address of a constant is subject to bugs and even the dreaded "undefined behavior".
- **Bloat**: each translation unit including your header gets its own copy of the thing. Not such a big deal for simple things like the primitive numeric types. Not so great for strings and bigger data structures.

When at namespace scope (i.e. not in a function or in a class), both `const` and `constexpr` objects implicitly have internal linkage (the same linkage used for unnamed-namespace variables and `static` variables not in a function or in a class). The C++ standard guarantees that every translation unit that uses or references the object gets a different "copy" or "instantiation" of the object, each at a *different address*.

Within a class, you must additionally declare these objects as `static`, or they will be unchangeable instance variables, rather than unchangeable class variables shared among every instance of the class.

Likewise, within a function, you must declare these objects as `static`, or they will take up space on the stack and be constructed every time the function is called.

Q.1.3.2 An Example Bug

So, is this a real risk? Consider:

```
// Declared in do_something.h
constexpr char kSpecial[] = "special";

// Does something.  Pass kSpecial and it will do something special.
void DoSomething(const char* value);
```

```
// Defined in do_something.cc
void DoSomething(const char* value) {
  // Treat pointer equality to kSpecial as a sentinel.
  if (value == kSpecial) {
    // do something special
  } else {
    // do something boring
  }
}
```

Notice that this code compares the address of the first char in `kSpecial` to `value` as a kind of magic value for the function. You sometimes see code do this in an effort to short circuit a full string comparison.

This causes a subtle bug. The `kSpecial` array is `constexpr` which implies that it is `static` (with "internal" linkage). Even though we think of `kSpecial` as "a constant" – it really isn't – it's a whole family of constants, one per translation unit! Calls to `DoSomething(kSpecial)` look like they should do the same thing, but the function takes different code paths depending on where the call occurs.

Any code using a constant array defined in a header file, or code that takes the address of a constant defined in a header file, suffices for this kind of bug. This class of bug is usually seen with string constants, because they are the most common reason to define arrays in header files.

Q.1.3.3 An Example of Undefined Behavior

Just tweak the above example, and move `DoSomething` into the header file as an `inline` function. Bingo: now we've got undefined behavior, or UB. The language requires all `inline` functions to be defined exactly the same way in every translation unit (source file) – this is part of the language's "One Definition Rule". This particular `DoSomething` implementation references a static variable, so every translation unit actually defines `DoSomething` *differently*, hence the undefined behavior.

Unrelated changes to program code and compilers can change inlining decisions, which can cause undefined behavior like this to change from benign behavior to bug.

Q.1.3.4 Does this Cause Problems in Practice?

Yes. In one real-life bug we've encountered, the compiler was able to determine that in a particular translation unit (source file), a large static const array defined in a header file was only partially used. Rather than emit the entire array, it optimized away the parts it knew weren't used. One of the ways the array was partially used was through an inline function declared in a header.

The trouble is, the array was used by other translation units in such a way that the static const array was fully used. For those translation units, the compiler generated a version of the inline function that used the full array.

Then the linker came along. The linker assumed that all instances of the inline function were the same, because the One Definition Rule said they had to be. And it discarded all but one copy of the function - and that was the copy with the partially-optimized array.

This kind of bug is possible when code uses a variable in a way that requires its address to be known. The technical term for this is "ODR used". It is difficult to prevent ODR use of variables in modern C++ programs, particularly if those values are passed to template functions (as was the case in the above example).

These bugs do happen and are not easily caught in tests or code review. It pays to stick to safe idioms when defining constants.

Q.2 Other Common Mistakes

Q.2.0.1 Mistake #1: the Non-Constant Constant

Seen most often with pointers:

```
const char* kStr = ...;
const Thing* kFoo = ...;
```

The `kFoo` above is a pointer to const, but the pointer itself is not a constant. You can assign to it, set it null, etc.

```
// Corrected.
const Thing* const kFoo = ...;
```

```
// This works too.
constexpr const Thing* kFoo = ...;
```

Q.2.0.2 Mistake #2: the Non-Constant MyString()

Consider this code:

```
inline absl::string_view MyString() {
  return "Hello";  // may return a different value with every call
}
```

The address of a string literal constant is allowed to change every time it is evaluated, so the above is subtly wrong because it returns a `string_view` that might have a different `.data()` value for each call. While in many cases this won't be a problem, it can lead to the bug described above.

Making the `MyString()` `constexpr` does not fix the issue, because the language standard does not say it does. One way to look at this is that a `constexpr` function is just an `inline` function that is allowed to execute at compile time when initializing constant values. At run time it is not different from an `inline` function.

```
constexpr absl::string_view MyString() {
  return "Hello";  // may return a different value with every call
}
```

To avoid the bug, use a `static constexpr` variable in a function instead:

```
inline absl::string_view MyString() {
  static constexpr char kHello[] = "Hello";
  return kHello;
}
```

Rule of thumb: if your "constant" is an array type, store it in a function local static before returning it. This fixes its address.

Q.2.0.3 Mistake #3: Non-Portable Code

Some modern C++ features are not yet supported by some major compilers.

- In both Clang and GCC, the `static constexpr char kHello[]` array in the `MyString` function above can be a `static constexpr absl::string_view`. But this won't compile in Microsoft Visual Studio. If portability is a concern, avoid `constexpr absl::string_view` until we get the `std::string_view` type from C++17.

```cpp
inline absl::string_view MyString() {
  // Visual Studio refuses to compile this.
  static constexpr absl::string_view kHello = "Hello";
  return kHello;
}
```

- For the `extern const` variables declared in header files the following approach to defining their values is valid according to the standard C++, and would in fact be preferrable to `ABSL_CONST_INIT`, but it is not yet supported by some compilers.

  ```cpp
  // Defined in foo.cc -- valid C++, but not supported by MSVC 19.
  constexpr absl::string_view kOtherBufferName = "other example";
  ```

  As a workaround for a `constexpr` variable in a `.cc` file you can provide its value to other files through functions.

Q.2.0.4 Mistake #4: Improperly Initialized Constants

The style guide has some detailed rules intended to keep us safe from common problems related to run-time initialization of static and global variables. The root issue arises when the initialization of global variable `X` references another global variable `Y`. How can we be sure `Y` itself doesn't somehow depend on the value of `X`? Cyclic initialization dependencies can easily happen with global variables, especially with those we think of as constants.

This is a pretty thorny area of the language in its own right. The style guide is an authoritative reference.

Consider the above links required reading. With a focus on initialization of constants, the phases of initialization can be explained as:

1. **Zero initialization**. This is what initializes otherwise uninitialized static variables to the "zero" value for the type (e.g. `0`, `0.0`, `'\0'`, `null`, etc.).

   ```cpp
   const int kZero;  // this will be zero-initialized to 0
   const int kLotsOfZeroes[5000];  // so will all of these
   ```

   Note that relying on zero initialization is fairly popular in C code but is fairly rare and niche in C++. It is generally clearer to assign variables explicit values, even if the value is zero, which brings us to…

2. **Constant initialization**.

   ```cpp
   const int kZero = 0;  // this will be constant-initialized to 0
   const int kOne = 1;   // this will be constant-initialized to 1
   ```

   Both "constant initialization" and "zero initialization" are called "static initialization" in the C++ language standard. Both are always safe.

3. **Dynamic initialization**.

```cpp
// This will be dynamically initialized at run-time to
// whatever ArbitraryFunction returns.
const int kArbitrary = ArbitraryFunction();
```

Dynamic initialization is where most problems happen. The style guide explains why at Static and Global Variables.

Note that documents like the Google C++ style guide have historically included dynamic initialization in the broad category of "static initialization". The word "static" applies to a few different concepts in C++, which can be confusing. "Static initialization" can mean "initialization of static variables", which can include run-time computation (dynamic initialization). The language standard uses the term "static initialization" in a different, narrower, sense: initialization that is done statically or at compile-time.

Q.3 Initialization Cheat Sheet

Here is a super-quick constant initialization cheat sheet (not in header files):

- `constexpr` guarantees safe constant initialization as well as safe (trivial) destruction. Any `constexpr` variable is entirely fine when defined in a `.cc` file, but is problematic in header files for reasons explained earlier.
- `ABSL_CONST_INIT` guarantees safe constant initialization. Unlike `constexpr`, it does not actually make the variable `const`, nor does it ensure the destructor is trivial, so care must still be taken when declaring static variables with it. See again Static and Global Variables.
- Otherwise, you're most likely best off using a static variable within a function and returning it. See the "ordinary function" example shown earlier.

Q.4 Further Reading and Collected Links

- Static and Global Variables
- constexpr
- inline
- storage duration (linkage rules)
- ub (Undefined Behavior)

Q.5 Conclusion

The `inline` variable from C++17 can't come soon enough. Until then all we can do is use the safe idioms that steer us clear of the rough edges.

- We conclude that string literals are not required to evaluate to the same object from the following language in `[lex.string]` in the C++17 language standard. Equivalent language is also present in C++11 and C++14.
- There is no language in `[lex.string]` describing different behavior in a `constexpr` context.

Appendix R
Tip of the Week #148: Overload Sets

Titus Winters (titus@google.com)

One of the effects of living with electric information is that we live habitually in a state of information overload. There's always more than you can cope with.

—Marshall McLuhan

In my opinion, one of the most powerful and insightful sentences in the C++ style guide is this: "Use overloaded functions (including constructors) only if a reader looking at a call site can get a good idea of what is happening without having to first figure out exactly which overload is being called".

On the surface, this is a pretty straightforward rule: overload only when it won't cause confusion to a reader. However, the ramifications of this are actually fairly significant and touch on some interesting issues in modern API design. First let's define the term "overload set", and then let's look at some examples.

R.1 What is an Overload Set?

Informally, an overload set is a set of functions with the same name that differ in the number, type and/or qualifiers of their parameters. (See Overload Resolution for all the gory details.) You may not overload on the return type from a function - the compiler must be able to tell which member of the overload set to call based on the invocation of the function, regardless of return type.

```
int Add(int a, int b);
int Add(int a, int b, int c);  // Number of parameters may vary

// Return type may vary, as long as the selected overload is uniquely
// identifiable from only its parameters (types and counts).
float Add(float a, float b);

// But if two return types have the same parameter signature, they can't form a
// proper overload set; the following won't compile with the above overloads.
int Add(float a, float b);    // BAD - can't overload on return type
```

R.2 String-ish Parameters

Thinking back on my earliest experiences with C++ at Google, I'm almost positive that the first overloads I encountered were of the form:

```
void Process(const std::string& s) { Process(s.c_str()); }
void Process(const char*);
```

The wonderful thing about overloads of this form is that they meet the letter and the spirit of the rule, in a very obvious fashion. There is no behavioral difference here: in both cases, we're accepting some form of string-ish data, and the inline forwarding function makes it perfectly clear that the behavior of every member of the overload set is identical.

That turns out to be critical, and easy to overlook, since the Google C++ style guide doesn't phrase it explicitly: if the documented behavior of the members of an overload set varies, then a user implicitly has to know which function is actually being called. The only way to ensure that they have a "good idea what is happening" without figuring which overload is being called is if the semantics of each entry in the overload set are identical.

So, the string-ish examples above work because they have identical semantics. Borrowing an example from the C++ Core Guidelines, we would not want to see something like:

```
// remove obstacle from garage exit lane
void open(Gate& g);

// open file
void open(const char* name, const char* mode ="r");
```

Hopefully, namespace differences suffice to disambiguate functions like these from actually forming an overload set. Fundamentally, this would be a bad design specifically because APIs should be understood and documented at the level of the overload set, not the individual functions that make it up.

R.3 StrCat

`StrCat()` is one of the most common Abseil examples for demonstrating that overload sets are often the right granularity for API design. Over the years, `StrCat()` has changed the number of parameters it accepts, and the form that it uses to express that parameter count. Years ago, `StrCat()` was a set of functions with varying arity. Now, it is conceptually a variadic template function … although for optimization reasons small-count arities are still provided as overloads. It has never actually been a single function - we just treat it conceptually as one entity.

This is a good use of overload sets - it would be annoying and redundant to encode the parameter count in the function name, and conceptually it doesn't matter how many things are passed to `StrCat()` - it will always be the "convert to string and concatenate" tool.

R.4 Parameter Sinks

Another technique that the standard uses and that comes up in a lot of generic code is to overload on `const T&` and `T&&` when passing a value that will be stored: a value sink. Consider `std::vector::push_back()`:

```
void push_back(const T&);
void push_back(T&&);
```

It's worth considering the origin of this overload set: when the `push_back()` API first appeared, it contained `push_back(const T&)` which served as a cheap (and safe) parameter to pass. With C++11 the `push_back(T&&)` overload was added as an optimization for cases where the value is a temporary, or the caller has promised not to interfere with the parameter by writing out `std::move()`. Even though the moved-from object may be left in a different state, these still provide the same semantics for the user of the vector, so we consider them a well-designed overload set.

Put another way, the `&` and `&&` qualifiers denote whether that overload is available for lvalue or rvalue expressions; if you have a `var` or `var&` argument, you will get the `&` overload, but if you have a temporary or have performed a `std::move()` on your expression, you will get the `&&` overload. (See Tip #77 for more on move-semantics.)

Interestingly, these overloads are semantically the same as a single method – `push_back(T)` – but in some cases may be slightly more efficient. Such efficiency mostly matters when the body of the function is cheap compared to the cost of invoking the move constructor for `T` – possible for containers and generics, but unlikely in many other contexts. We generally recommend that if you need to sink a value (store in an object, mutate a parameter, etc) you just provide the single function accepting `T` (or `const T&`) for simplicity and maintainability. Only if you are writing very high-performance generic code is the difference likely to be relevant. See Tip #77 and Tip #117.

R.5 Overloaded Accessors

For methods on a class (especially a container or a wrapper), it is sometimes valuable to provide an overload set for accessors. Standard library types provide many great examples here - we'll consider just `vector::operator[]` and `optional::value()`.

In the case of `vector::operator[]`, two overloads exist: one const and one non-const, which accordingly return a const or non-const reference, respectively. This matches our definition nicely; a user doesn't need to know which thing is invoked. The semantics are the same, differing only in constness – if you have a non-const `vector` you get a non-const reference, and if you have a `const` vector you get a const reference. Put another way: the overload set is purely forwarding the const-ness of the `vector`, but the API is consistent – it just gives you the indicated element.

In C++17 we added `std::optional<T>`, a wrapper for at most one value of an underlying type. Just like `vector::operator[]`, when accessing `optional::value()` both const and non-const overloads exist. However, optional goes one step further and provides `value()` overloads based on "value category" (roughly speaking, whether the object is a temporary). So the full pairwise combination of const and value category looks like:

```
T& value() &;
const T & value() const &;
T&& value() &&;
const T&& value() const &&;
```

The trailing `&` and `&&` apply to the implicit `*this` parameter, just in the same way const qualifying a method does, and indicate acceptance of lvalue or rvalue arguments as noted in Parameter Sinks above. Importantly, however, you don't actually need to understand move semantics to understand `optional::value()` in this case. If you ask for the value out of a temporary, you get the value as if it were a temporary itself. If you ask for the value out of a const ref, you get a const-ref of the value. And so on.

R.6 Copy vs. Move

The most important overload sets for types are often their set of constructors, especially the copy and move constructors. Copy and move, done right, form an overload set in all senses of the term: the reader should not need to know which of those overloads is chosen, because the semantics of the newly-constructed object should be the same in either case (assuming both constructors exist). The standard library is becoming more explicit about this: move is assumed to be an optimization of copy, and you should not depend on the particulars of how many moves or copies are made in any given operation.

R.7 Conclusion

Overload sets are a simple idea conceptually, but prone to abuse when not well understood - don’t produce overloads where anyone might need to know which function was chosen. But when used well, overload sets provide a powerful conceptual framework for API design. Understanding the subtleties of the style guide’s description of overload sets is well worth your time when thinking about API design.