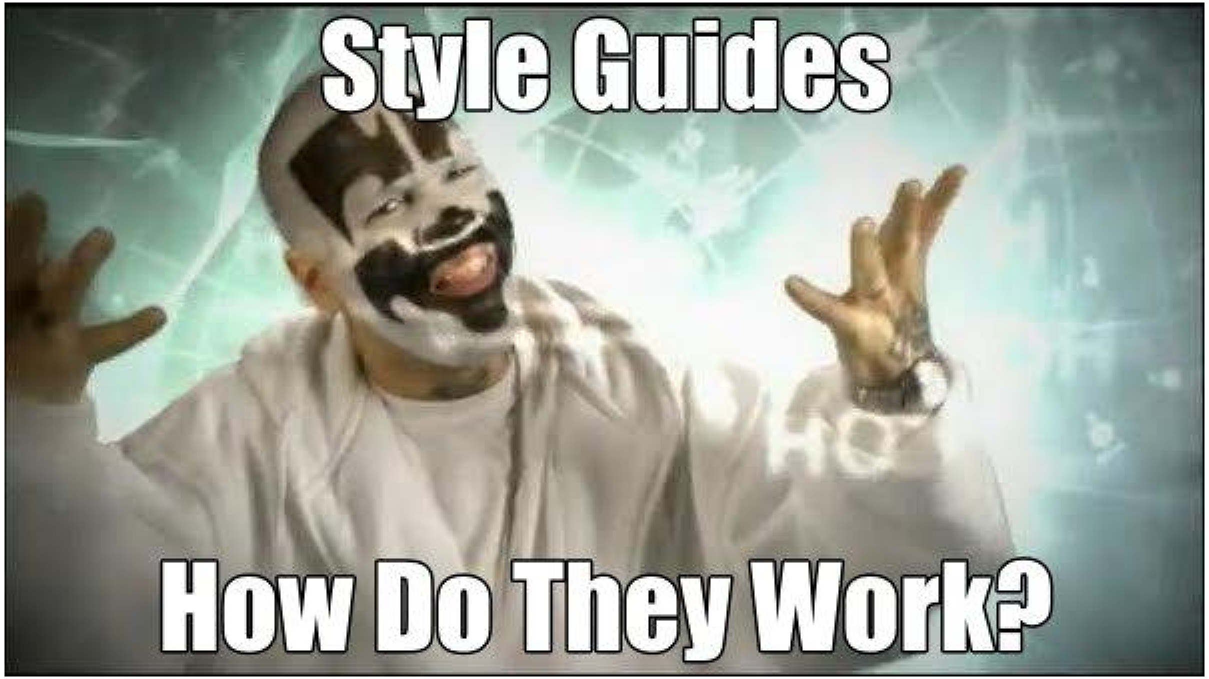
The Philosophy of Google's C++ Style

Titus Winters (titus@google.com)
C++ Codebase Cultivator


Style Guides
How Do They Work?

Style Guides!

Titus Winters (titus@google.com)

How Should We Format Our Code?

How Should We Format Our Code?

BORING QUESTION!

What Goes in a Style Guide?

What Goes in a Style Guide?

WRONG QUESTION!

What is the Purpose of a Style Guide?

What is the Purpose of a Style Guide?

What's the purpose of any rule or set of rules your organization puts out?

What is the Purpose of a Style Guide?

What's the purpose of any rule or set of rules your organization puts out?

- Heavy handed throw-your-weight-around hoop jumping

What is the Purpose of a Style Guide?

What’s the purpose of any rule or set of rules your organization puts out?

- ~~Heavy handed throw-your-weight-around hoop jumping~~
- Make it harder for people to do “bad” things, encourage “good” things
  - Clearly depends on your organization’s goals

Outline

- The underpinnings of Google’s C++ Style Guide
- The contentious rules
- Plenty of time for Q&A

About Us

Context about Google

- 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).

Meaning?

Code is going to live a long time, and be read many times. We choose explicitly to optimize for the reader, not the writer.

Philosophies of the 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

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

#4 Be Consistent

- 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

#4 Be Consistent

- 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

#4 Be Consistent

- 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);
}
```

#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);
}
```

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

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

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

```
// 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();
```

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

```
// 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));
```

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

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)

#6 Avoid tricky and hard-to-maintain constructs

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)

The Contentious Rules

There are two (very) contentious rules:

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

The Contentious Rules: non-const references

Three rules apply:

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

The Contentious Rules: 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.

Recap

Have a style guide. Tailor it to your situation.

Recap

Use your guide to encourage “good” and discourage “bad.”

Recap

Re-evaluate.

And with that . . .

Questions?