Google CPP 风格指南简单介绍

Brief Introduction to Google C++ Style Guide

蔡振廷

目录

CONTENTS

01

Intro

- Background
- Goal

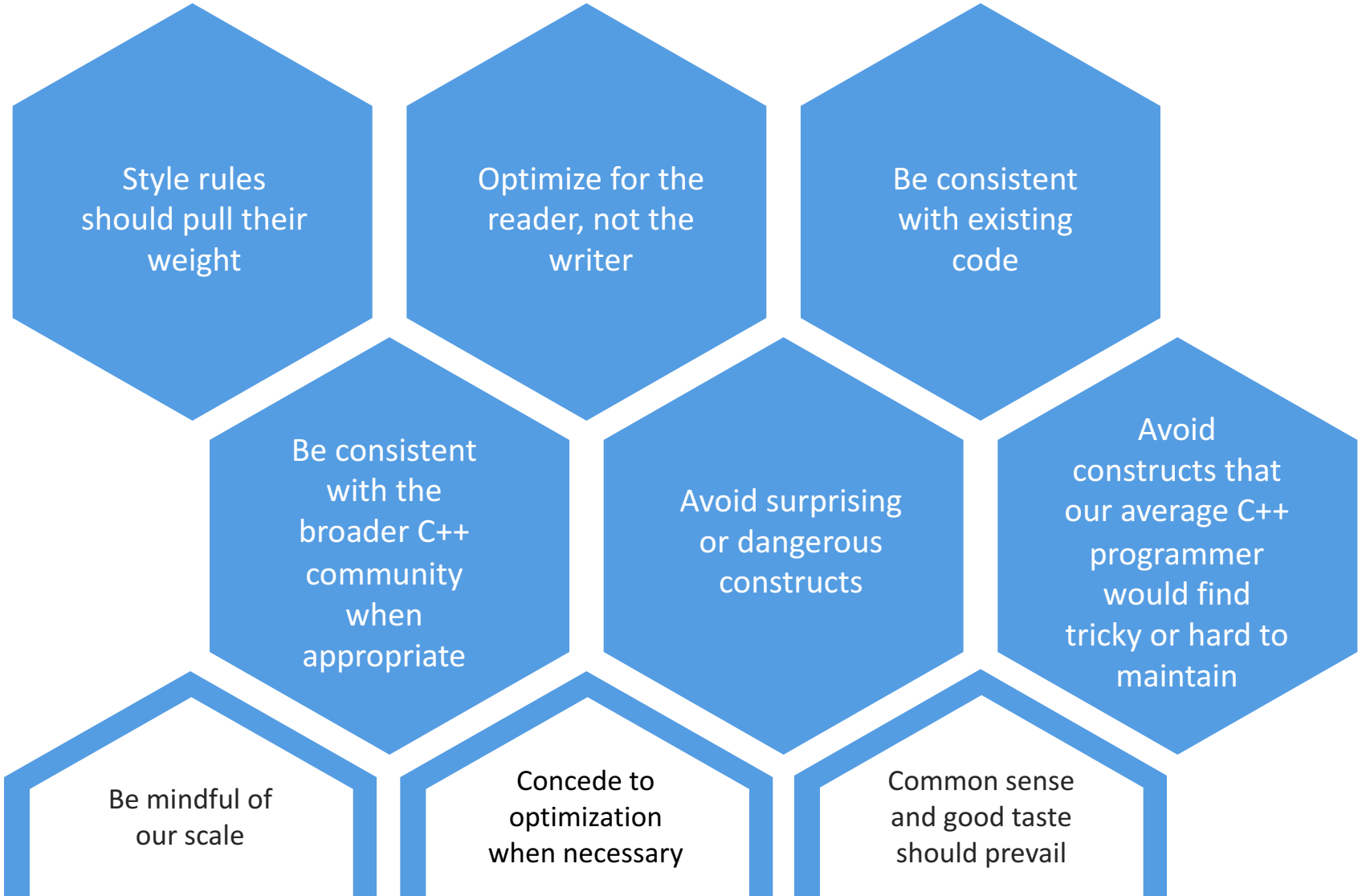
Background

1. Manage complexity by describing in detail the dos and don'ts of writing C++ code.

2. Keep the code base manageable while still allowing coders to use C++ language features productively.

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

Goal


Style rules should pull their weight
Optimize for the reader, not the writer
Be consistent with existing code
Be consistent with the broader C++ community when appropriate
Avoid surprising or dangerous constructs
Avoid constructs that our average C++ programmer would find tricky or hard to maintain
Be mindful of our scale
Concede to optimization when necessary
Common sense and good taste should prevail

Avoid

AMBIGUOUS

02

Guidance

- Header Files
- Functions
- Classes
- Google-Specific Magic

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_.

foo/src/bar/baz.h

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

Inline Functions

- **Define functions inline only when they are small, say, 10 lines or fewer.**
- 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.
- Pros:
  - Generate more efficient object code. (the inlined function is small)
- Cons:
  - Overuse of inlining can actually make programs slower.

Inline Functions

- **Define functions inline only when they are small, say, 10 lines or fewer.**
- 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.
- Decision:
  - Don’t inline a function if it is more than 10 lines long.
  - Beware of destructors. (often longer than they appear)
  - Inline functions with loops or switch statements are typically not worthy.

Names and Order of Includes

- All of a project‘s header files should be listed as descendants of the project’s source directory.
- Without use of UNIX directory shortcuts
  . (the current directory) or
  .. (the parent directory).
- For example：
  - `google-awesome-project/src/base/logging.h`

  should be included as:

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

Names and Order of Includes

- Use standard order for readability and to avoid hidden dependencies:
  - Related header,
  - C library,
  - C++ library,
  - other libraries' .h,
  - your project's .h.

- For example：
  - `google-awesome-project/src/foo/internal/fooserver.cc`

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

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

#include <hash_map>
#include <vector>

#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/server/bar.h"
```

Inheritance

- **Composition is often more appropriate than inheritance.**
  When using inheritance, make it public.
- Implementation inheritance: actual code is inherited by the child.
- Interface inheritance: only method names are inherited.
- Pros:
  - Implementation inheritance reduces code size.
  - Interface inheritance can be used to programmatically enforce that a class expose a particular API.
  - you and the compiler can understand the operation and detect errors.

Inheritance

- **Composition is often more appropriate than inheritance.**
  When using inheritance, make it public.
- Implementation inheritance: actual code is inherited by the child.
- Interface inheritance: only method names are inherited.
- Cons:
  - For implementation inheritance, it can be more difficult to understand an implementation.
  - The sub-class cannot override functions that are not virtual.

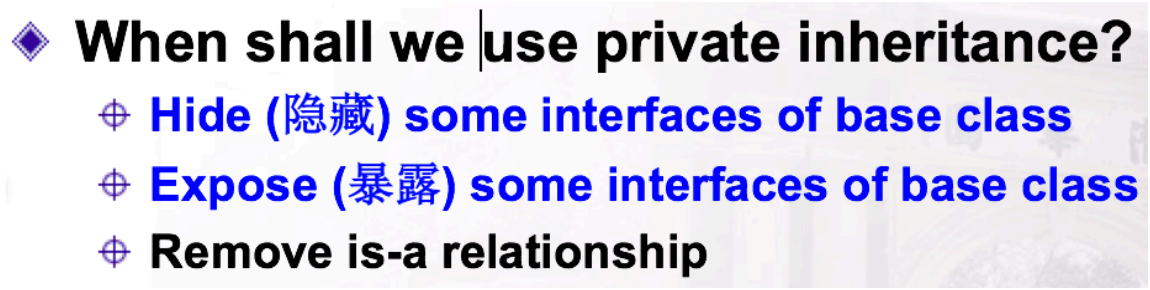
Inheritance

- **Composition is often more appropriate than inheritance.**
  When using inheritance, make it public.
- Implementation inheritance: actual code is inherited by the child.
- Interface inheritance: only method names are inherited.
- 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.

Inheritance

- **Composition is often more appropriate than inheritance.**
  When using inheritance, make it public.
- Implementation inheritance: actual code is inherited by the child.
- Interface inheritance: only method names are inherited.

- 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.



Inheritance

- **Composition is often more appropriate than inheritance.**
  When using inheritance, make it public.
- Implementation inheritance: actual code is inherited by the child.
- Interface inheritance: only method names are inherited.
- Decision:
  - Do not overuse implementation inheritance.
  - Composition is often more appropriate.
  - Try to restrict use of inheritance to the "is-a" case.
  - Limit the use of protected to those member functions that might need to be accessed from subclasses.
    (Data members should be private.)

Parameter Ordering

- **When defining a function, parameter order is: inputs, then outputs.**
- This is not a hard-and-fast rule.
- Consistency with related functions may require you to bend the rule.

Write Short Functions

- **Prefer small and focused functions.**
- 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.
- You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code.

Reference Arguments

- **All parameters passed by reference must be labeled const.**

- Decision:
  - Within function parameter lists all references must be const:

```
void Foo(const string &in, string *out);
```

  - Strong convention in Google code.

- Exception:
  - swap()

Reference Arguments

- **All parameters passed by reference must be labeled const.**

- However, there are some instances where using const T* is preferable to const T& for input parameters. For example:
  - You want to pass in a null pointer.
  - The function saves a pointer or reference to the input.

Google-Specific Magic

cpplint

- **Use cpplint.py to detect style errors.**
- A tool that reads a source file and identifies many style errors.

Google-Specific Magic

cpplint

- **Use cpplint.py to detect style errors.**
- A tool that reads a source file and identifies many style errors.
- False positives can be ignored by putting
  - // NOLINT at the end of the line or
  - // NOLINTNEXTLINE in the previous line.

Reference

- https://google.github.io/styleguide/cppguide.html#Classes
- https://zh-google-styleguide.readthedocs.io/en/latest/