# Introduction to C++ Coding Style

Xie Saining

Shanghai Jiao Tong University

Oct 18, 2011

# Contents I

# Contents II

# The #define Guard

Tip: All header files should have #define guards
to prevent multiple inclusion.
Format: <PROJECT>_<PATH>_<FILE>_H_

Example: foo/src/bar/baz.h

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
...
#endif FOO_BAR_BAZ_H_
```

**Listing 1: sample B**

```
//a.h
#include "b.h"
class A
{
....
private:
    B b;
};
//b.h
#include "a.h"
class B
{
....
private:
    A a;
};
```

**Listing 2: sample B**

```
//a.h
//#include "b.h"
class B;
class A
{
 ....
private:
 B *b;
};
//b.h
#include "a.h"
class B
{
 ....
private:
 A a;
};
```

## Listing 3: sample B

```
//a.h
#include "b.h"
class A
{
....
private:
    B b;
};
//b.h
#include "a.h"
class B
{
....
private:
    A a;
};
```

## Listing 4: sample B

```
//a.h
//#include "b.h"
class B;
class A
{
  ....
private:
 B *b;
};
//b.h
#include "a.h"
class B
{
  ....
private:
 A a;
};
```

# Header File Dependencies

How can we use a class Foo in a header file without access to its definition?

- We can declare data members of type Foo$\star$ or Foo&
- We can declare (but not define) functions with arguments, and/or return values, of type Foo.
- We can declare static data members of type Foo. this is because static data members are defined outside the class definition.

On the other hand,you must include the header file for Foo if your class subclasses Foo or has a data member of type Foo.

# Header File Dependencies

How can we use a class Foo in a header file without access to its definition?

- We can declare data members of type Foo$\star$ or Foo&
- We can declare (but not define) functions with arguments, and/or return values, of type Foo.
- We can declare static data members of type Foo. this is because static data members are defined outside the class definition.

On the other hand,you must include the header file for Foo if your class subclasses Foo or has a data member of type Foo.

# Header File Dependencies

How can we use a class Foo in a header file without access to its definition?

- We can declare data members of type Foo$\star$ or Foo&
- We can declare (but not define) functions with arguments, and/or return values, of type Foo.
- We can declare static data members of type Foo. this is because static data members are defined outside the class definition.

On the other hand,you must include the header file for Foo if your class subclasses Foo or has a data member of type Foo.

# Header File Dependencies

How can we use a class Foo in a header file without access to its definition?

- We can declare data members of type Foo$\star$ or Foo&
- We can declare (but not define) functions with arguments, and/or return values, of type Foo.
- We can declare static data members of type Foo. this is because static data members are defined outside the class definition.

On the other hand,you must include the header file for Foo if your class subclasses Foo or has a data member of type Foo.

# Inline Function

- **Tip**: Define functions inline only when they are small say 10 lines or less.
- **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.
- You will gain a deeper understanding working on the Tiger Compiler Project.

# Function Parameter Ordering

- Tip: When defining a function, parameter order is: inputs, then outputs.
- This is not a hard-and-fast rule. Parameters that are both input and output (often classes/structs) muddy the waters, and, as always, consistency with related functions may require you to bend the rule

# Function Parameter Ordering

- Tip: When defining a function, parameter order is: inputs, then outputs.
- This is not a hard-and-fast rule. Parameters that are both input and output (often classes/structs) muddy the waters, and, as always, consistency with related functions may require you to bend the rule

# Names and Order of Includes

- Tip: Use standard order for readability and to avoid hidden dependencies: C library, C++ library, other libraries' .h, your project's .h.
- Within each section it is nice to order the includes alphabetically.

# Names and Order of Includes

**Listing 5: An Example**

```
#include "foo/public/fooserver.h"  //preferred position
#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>
#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/public/bar.h"
```

---

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

# Local Variables

- Tip: 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,

**Listing 6: An Example**

```
int i;
i = f();      //Bad:initialization separate from declaration.
int j = g();  //Good:declaration has initialization.
```

# Local Variables

- Warning:
  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.

**Listing 7: sample A**

```
for (int i = 0; i < 1000000;
     ++i) {
Foo f;
f.DoSomething(i);
}
```

**Listing 8: sample B**

```
Foo f;
for (int i = 0; i < 1000000;
     ++i) {
f.DoSomething(i);
}
```

# Local Variables

- Warning:
  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.

**Listing 9: sample A**

```
for (int i = 0; i < 1000000;
     ++i) {
Foo f;
f.DoSomething(i);
}
```

**Listing 10: sample B**

```
Foo f;
for (int i = 0; i < 1000000;
     ++i) {
f.DoSomething(i);
}
```

# Static and Global Variables

- **Tip**: Static or global variables of class type are forbidden: they cause hard-to-find bugs due to indeterminate order of construction and destruction.
- Objects with static storage duration, including global variables, static variables, static class member variables, and function static variables, must be Plain Old Data (POD): only int, char, float, and void, and arrays of/structs of/pointers to POD. Static variables must not be initialized with the result of a function; and non-const static variables must not be used in threaded code.

# Doing Work in Constructors

- **Tip**: Do only trivial initialization in a constructor. If at all possible, use an Init() method for non-trivial initialization.
- If your object requires non-trivial initialization, consider having an explicit Init() method and/or adding a member flag that indicates whether the object was successfully initialized.

# Default Constructors

- **Tip**: You must define a default constructor if your class defines member variables and has no other constructors.Otherwise the compiler will do it for you, badly.
- if you have no other constructors and do not define a default constructor, the compiler will generate one for you. This compiler generated constructor may not initialize your object sensibly.
- If your class inherits from an existing class but you add no new member variables, you are not required to have a default constructor.

# Access Control

- **Tip**: Make all data members private, and provide access to them through accessor functions as needed. Typically a variable would be called foo_ and the accessor function foo(). You may also want a mutator function set_foo().
- The definitions of accessors are usually inlined in the header file.

# Declaration Order

- **Tip**: Use the specified order of declarations within a class: **public:** before **private:**, **methods** before **data members (variables)**, etc.
- public: section, then protected: section, then private: section. If any of these sections are empty, omit them.
- Within each section,

Typedefs and Enums
Constants
Constructors
Destructor
Methods, including static methods
Data Members, including static data members

# Write Short Functions

- **Tip**: 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.
- Long function results in bugs that are hard to find

# Default Arguments

- **Tip**: We do not allow default function parameters.
- We require all arguments to be explicitly specified, to force programmers to consider the API and the values they are passing for each argument rather than silently accepting defaults they may not be aware of.

# Variable-Length Arrays and alloca()

- **Tip**: We do not allow variable-length arrays or alloca().
- NOT part of Standard C++.
- Data-dependent, lack of portability, hard to transplant, modify, and debug.

# Preincrement and Predecrement

- **Tip**: Use prefix form (++i) of the increment and decrement operators with iterators and other template objects.
- post-increment (or decrement) requires a copy of i to be made, which is the value of the expression. If i is an iterator or other non-scalar type, copying i could be expensive.

# Line Length

- **Tip**: Each line of text in your code should be at most 80 characters long.
- Exception: if a comment line contains an example command or a literal URL longer than 80 characters, that line may be longer than 80 characters for ease of cut and past
- Exception: an #include statement with a long path may exceed 80 columns. Try to avoid situations where this becomes necessary.
- Exception: you needn't be concerned about header guards that exceed the maximum length.

# Horizontal Whitespace

- **Tip**: Use of horizontal whitespace depends on location. Never put trailing whitespace at the end of a line.
- See some examples.

# Vertical Whitespace

- **Tip**: Minimize use of vertical whitespace.
- See some examples.

# Further Reading I

*Google Style Guide*.
http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

Cpplint
A tiny Software(Actually a python script file)
http://google-styleguide.googlecode.com/svn/trunk/cpplint/cpplint.py
2000.