**Coding Standards & Practices**

Motivation

To provide a set of standards for how code is written within this mentorship. As a professional software engineer, you will need to conform to the standards set forth by the company you work with. These standards are based on common professional practices.

Naming Conventions

Classes, namespaces, functions, typedef’s and enum names should be named using CamelCase, where the first letter is capitalized:

class FooBar;

namespace FooBar { }

enum FooBar { };

typedef std::vector<int> IntVector;

void DoTheThing();

Filenames should be named in the same way:

FooBar.h

FooBar.cpp

Variables should be named in camel case, but with the first letter lowercase:

int fooBar;

Member variables, global variables, and static variables must have a single letter prefix with an underscore denoting their scope. Local variables have no prefix.

int m\_fooBar; // member variable

int g\_fooBar; // global variable

int s\_fooBar; // static variable (class or non-class)

int fooBar; // local variable

Additionally, pointers should be prefixed with a ‘p’:

int\* m\_pFooBar; // member pointer variable

int\* g\_pFooBar; // global pointer variable

int\* s\_pFooBar; // static pointer variable (class or non-class)

int\* pFooBar; // local pointer variable

Constant variables should be prefixed with a k. This includes const variables, consexpr variables, and enum values:

const char\* kpFooBar = “baz”;

enum FooBar

{

kFooBar,

kFooBarBaz

};

constexpr int kMaxHitPoints = 100;

#define’s should be all caps, with underscores between each word:

#define FOO\_BAR

Bracing & Formatting

Bracing & indentation should adhere to the Allman style:

<https://en.wikipedia.org/wiki/Indent_style#Allman_style>

while (x == y)

{

Foo();

FooBar();

}

FooBarBaz();

Indentation should be set to 4 spaces. Make sure you are inserting spaces rather than inserting the tab character. *In Visual Studio, you can set this by going to Tools -> Options -> Text Editor -> C/C++ -> Tabs. Change the Tab size to 4 and make sure “Insert spaces” is set.*

Binary operators should always be surrounded by a single space on either end, while unary operators should not have any spaces:

int foo = bar + baz;

++foo;

Never use single-line if statements or loops:

if (foo) FooBar(); // NO

while (foo) FooBar(); // NO

Instead, put it on the next line:

if (foo)

FooBar();

while (foo)

FooBar();

Multiple statements on the same line are also not allowed:

x = 10; y = 15; // NO

Single-line functions are allowed only if they are trivial accessors or mutators:

class Foo

{

int m\_bar;

public:

int GetBar() const { return m\_bar; }

void SetBar(int bar) { m\_bar = bar; }

// NO. This function has two statements, so it can’t be

// a single line. It’s also no longer a trivial getter

// since it has additional side effects.

int IncrementAndGetBar() { ++m\_bar; return m\_bar; }

};

It’s okay to omit the braces on an if statement or loop if there can be absolutely no confusion that it’s the only statement within the body, as is the case above. You should avoid single-line bodies that take up multiple lines:

// avoid this -- use braces instead

if (foo)

while (bar)

if (baz)

DoTheThing();

When in doubt, use braces. It is always okay to use braces, even if the body is trivial:

// OK

if (foo)

{

++bar;

}

Classes

Classes should generally be structured as follows:

class FooBar

{

public:

// Nested classes, enums, and typedefs go here. These might

// be private or protected instead of public.

private:

// private data goes here

protected:

// protected data goes here

public:

// public functions goes here

// simple accessors & mutators go here

protected:

// protected member functions go here

private:

// private member functions go here

};

This is more of a guideline than a strict rule. Do whatever makes the class readable.

No public data! In other words, you should never have a public member variable. Use accessors and mutators to manipulate variables from outside of the object.

Prefer initializing member variables in the constructor’s initializer list. It should be formatting like this:

Foo::Foo()

: m\_bar(0)

, m\_baz(0)

{

//

}

This allows you to easily comment out a single line in the initializer list.

In-class initialization is not allowed:

class Foo

{

int m\_bar = 0; // NO

};

The exception is for const static ints that are used to set a member array size:

class Foo

{

static const int s\_kBarArraySize = 100; // OK

int mBar[s\_kBarArraySize];

};

Commenting

You should add comments in the follow circumstances:

1. At the top of a function.
   1. This comment should explain the purpose of the function.
   2. It should describe the inputs, outputs, and any assumptions.
2. At the top of a class.
   1. Describe the purpose of the class and how it should be used.
   2. Explain the main public interface.
   3. It’s not necessary to explain simple accessors or mutators.
3. For each block of code.
   1. There should be a short comment at the top of each block of code describing what that section does.
4. For anything that’s not extremely obvious.
   1. If you’re doing something tricky, write a big comment explaining it.
   2. If you had trouble figuring something out, write out your solution.
   3. If there’s a bit of confusing code, write a comment that explains it.
5. For any bug fixes or optimizations that weren’t obvious.
   1. If you changed something that seems like it should have worked, write a comment explaining what you changed and why.

Headers

For header guards, there is a choice between this:

#pragma once

and this:

#ifndef FOO\_BAR\_H

#define FOO\_BAR\_H

…

#endif // FOO\_BAR\_H

Use the first method. All modern compilers support this method and it is faster to compile in Visual Studio.

Header files should be self-contained. They should have header guards and include all the other headers necessary to compile them. In general, try to avoid adding #include’s inside header files where possible. Use forward declarations when you can..

main()

There are two forms of main() that are valid:

// Form 1: No arguments, returns an int

int main()

{

return 0; // this indicates normal program termination

}

// Form 2: command-line arguments, returns an int

int main(int argc, char\* args[])

{

return 0; // this indicates normal program termination

}

Either of these forms in valid. There is a third form you will see sometimes that has no return value:

// Form 3: No return. This is forbidden.

void main() // NO

{

//

}

This will compile just fine in Visual Studio 2015, but it is NOT considered valid C++. Because of this, it is forbidden. You must use Form 1 or Form 2.

Misc

Avoid the postfix increment and decrement operators. Always use the prefix version unless there’s a real reason not too (though there almost never is):

// yes

++foo;

--bar;

// no

foo++;

bar--;

Use parentheses whenever there’s any confusion over the order of operations. Don’t rely on other programmers memorizing the operator precedence tables. For example:

// NO

if (foo && bar || baz)

DoTheThing();

// YES

if (foo && (bar || baz))

DoTheThing();

**C++ exceptions should not be used for any reason.**

C++ 11

The use of C++ 11 is encouraged and, in many cases, required. All C++ 11 features are compatible with Visual Studio 2015 so you shouldn’t run into any issues.

Always use nullptr instead of NULL.

Range-based for loops are encouraged but never required:

std::vector<GameObject\*> gameObjects; // assume this is filled out

for (GameObject\* pGameObject : gameObjects)

{

pGameObject->Update();

}

The auto keyword may be used anytime that it doesn’t obfuscate the meaning. A common example is when assigning iterators, but other uses are permitted as well:

auto it = gameObjects.begin();

Scoped enums are generally preferred to non-scoped enums:

// scoped enum

enum class Foo

{

kBar,

kBaz,

};

// non-scoped enum

enum Foo

{

kBar,

kBaz,

};

Lambdas are fair game; they may be used in any context that makes sense, as long as they don’t make the code harder to decipher:

auto lambda = []() { cout << "foo" << endl; };

The use of override and final are required where appropriate:

class Foo

{

public:

virtual void DoTheThing() = 0;

virtual void DoSomethingElse() = 0;

};

class Bar : public Foo

{

public:

virtual void DoTheThing() override;

virtual void DoSomethingElse() final;

};

C++’s auto-generated functions should be defined as appropriate. Proper definition of move constructors and move assignment operators should follow the same guidelines as defining copy constructors and copy assignment operators. The delete and default modifiers should be used as appropriate.

For example, a class where copying isn’t appropriate but moving is fine might be defined as follows:

class Foo

{

std::vector<int> m\_bar;

public:

Foo(); // default constructor

// delete copy constructor and copy assignment operator

Foo(const Foo& right) = delete;

Foo& operator=(const Foo& right) = delete;

// define move constructor and move assignment operator

Foo(Foo&& right) : m\_bar(std::move(right.m\_bar)) { }

Foo& operator=(Foo&& right)

{

m\_bar = std::move(right.m\_bar);

return (\*this);

}

};

Prefer C++ 11’s cross-platform libraries over the platform-specific ones. For example, use std::thread instead of Windows’ threading API or pthreads. Use std::chrono instead of GetTickCount().

Prefer constexpr over const whenever you can.

C++ 14 / 17

The use of C++ 14 is optional. The only rule is that it must compile under Visual Studio 2015 with service pack 1. Many features of C++ 14 have not been implemented in Visual Studio 2015 yet, so exercise caution when using any of these features.

C++ 17 has not been officially released as of the writing of this document and as such, it is forbidden use any C++ 17 features in your code.