Coding Standard

Motivation

To provide a set of standards for how code is written within the T163 program. 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.

Note that any of these conventions may be changed for any given class. For example, if you're working in Unreal 4, your instructor may require you to follow their naming conventions. You will be told on the first day of class if this is the case.

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, constexpr variables, and enum values:

```
const char* kpFooBar = "baz";
enum FooBar
{
    kFooBar,
    kFooBarBaz
};

constexpr unsigned int kFoo = 0;
```

`#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();
```

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 and the braces:

```
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; }
};
```

Switch statements have the following form:

```
switch (foo)
{
    case 1:
        Bar();
        break;
```

```
    case 2:
        Baz();
        break;

    default:
        Error("Default reached");
        return false;
}
```

Alternatively, you may add braces to the case statements so that each case contains a local scope. You must either have braces around all of the cases or none of the cases. You may not mix braced and unbraced cases.

```
switch (foo)
{
    case 1:
    {
        Bar();
        break;
    }

    case 2:
    {
        Baz();
        break;
    }

    default:
    {
        Error("Default reached");
        return false;
    }
}
```

Either of the above forms are fine, but keep in mind that not including braces in a case statement can cause subtle errors because they will all share the same scope.

It is also permissible to have single-line case statements as long as they are only a single simple line, plus either a break or return statement:

```
switch (foo)
{
    case 1: Bar(); break;
    case 2: Baz(); break;
    default: Error("Default reached"); return false;  // return OK
```

```
}
```

One final note is to avoid magic numbers or literals as case tests. Prefer constexpr variables instead.

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.

Avoid public data in classes. In other words, you should never have a public member variable. Use accessors and mutators to manipulate variables from outside of the object. The exception is for structs or certain “primitive” mathematical types (Vector3 for instance).

Prefer initializing member variables in the constructor’s initializer list. It should be formatted 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.
    a. This comment should explain the purpose of the function.
    b. It should describe the inputs, outputs, and any assumptions.
2. At the top of a class.
    a. Describe the purpose of the class and how it should be used.
    b. Explain the main public interface.
    c. It's not necessary to explain simple accessors or mutators.
3. For each block of code.
    a. 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.
    a. If you're doing something tricky, write a big comment explaining it.
    b. If you had trouble figuring something out, write out your solution.
    c. 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.
    a. 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 in standard C/C++:

```
// 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 are 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, 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() override 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 / 20

The use of C++ 14 is optional, but encouraged. It is officially apart of the standard and supported by most of the major compilers, including Visual Studio 2017.

The C++ 17 standard has been drafted and is not expected to change. It's not 100% official as of the time this section was last updated (9/4/2017). It is the future of C++ and you are encouraged to look at it, but it's completely optional. Any C++ 17 features you use MUST compile under Visual Studio 2017.

C++ 20 is being talked about, but it has not been accepted or completed. Even if Visual Studio supports C++ 20 features, you are not allowed to use them. Feel free to read up on the upcoming features, though. Chances are, you'll end up using them soon enough.