Preface

Rule of Thumb

1. Readability first (your code should be your documentation most of the time)
2. Crash/Assert early. Don't wait until the worst case happens to make the crash condition.
3. Follow IDE's auto formatted style unless you have really good reasons not to do so. (Ctrl + K + D in VC++)
4. Learn from existing code

References

This coding standards is inspired by these coding standards

- Unreal engine 4 coding standard
- Doom 3 Code Style Conventions
- IDesign C# Coding Standard

1. Naming Conventions and Style

1. Use Pascal casing for class and structs

```
class PlayerManager;
struct AnimationInfo;
```

2. Use camel casing for local variable names and function parameters

```
void SomeMethod(const int someParameter);
{
  int someNumber;
}
```

3. Use verb-object pairs for method names
    a. Use pascal casing for public methods

```
public:
 void DoSomething();
```

    b. Use camel casing for other methods

```
private:
  void doSomething();
```

4. Use ALL_CAPS_SEPARATED_BY_UNDERSCORE for constants and defines
```
constexpr int SOME_CONSTANT = 1;
```

5. Use all lowercase letters for namespaces
```
namespace abc{};
```

6. prefix boolean variables with b
```
bool bFired;     // for local and public member variable
bool mbFired;    // for private class member variable
```

7. prefix interfaces with `I`
```
class ISomeInterface;
```

8. prefix enums with e
```
enum class eDirection
{
  North,
  South
}
```

9. prefix class member variables with `m`.
```
class Employee
{
protected:
  int mDepartmentID;
private:
  int mAge;
}
```

10. Methods with return values must have a name describing the value returned
```
uint32_t GetAge() const;
```

11. Use descriptive variable names. e.g `index` or `employee` instead of `i` or `e` unless it is a trivial index variable used for loops.

12. Capitalize every characters in acronyms if they have only 2 characters.
```
int ID;
```

13. Capitalize only first character in acronyms if they have more than 2 characters
```
int HttpCode;
```

14. Always use setter and getters for class member variables
    Use:
```
class Employee
```

```
{
  public:
    Const string& GetName() const;
    void SetName(const string& name);
  private:
    string mName;
}
```

Instead of:

```
class Employee
{
  public:
    string Name;
}
```

15. Use only public member variables for a struct. No functions are allowed. Use pascal casing for the members of a struct.

```
struct MeshData
{
  int32_t VertexCount;
}
```

16. Use `#include<>` for external header files. Use `#include ""` for in-house header files

17. Put external header files first, followed by in-house header files in alphabetic order if possible.

```
#include <vector>
#include <unordered_map>

#include "AnimationInfo.h"
```

18. There must be a blank line between includes and body.

19. Use `#pragma once` at the beginning of every header file

20. Use Visual Studio default for tabs. If you are not using Visual Studio, use real tabs that are equal to 4 spaces.

21. Declare local variables as close as possible to the first line where it is being used.

22. Always place an opening curly brace (`{`) in a new line

23. Add curly braces even if there's only one line in the scope

```
if (bSomething)
{
```

```
  return;
}
```

24. Use precision specification for floating point values unless there is an explicit need for a double
```
float f = 0.5f;
```

25. Always have a default case for a switch statement.
```
switch (number)
{
  case 0:
    ...
    break;
  default:
    break;
```

26. Always add predefined FALLTHROUGH for switch case fall through. This will be replaced by `[[fallthrough]]` attribute coming in for C++17 later
```
switch (number)
{
  case 0:
    DoSomething();
    FALLTHROUGH
  case 1:
    DoFallthrough();
    break;
  case 2:
  case 3:
    DoNotFallthrough();
  default:
    break;
}
```

27. If default case must not happen in a switch case, always add Assert(false). In our assert implementation, this will add optimization hint for release build.
```
switch (type)
{
  case 1:
    ...
    break;
  Default:
    Assert(false, "unknown type");
    break;
}
```

28. Use consts as much as possible even for local variable and function parameters.

29. Any member functions that doesn't modify the object must be const
```
int GetAge() const;
```

30. Do not return const value type. Const return is only for reference and pointers

31. Names of recursive functions end with "Recursive"
```
void FibonacciRecursive();
```

32. Order of class variables and methods must be as follows:
    a. list of friend classes
    b. public methods
    c. protected methods
    d. private methods
    e. protected variables
    f. private variables

33. Function overloading must be avoided in most cases
    Use:
```
const Anim* GetAnimByIndex(const int index) const;
const Anim* GetAnimByName(const char* name) const;
```

    Instead of:
```
const Anim* GetAnim(const int index) const;
const Anim* GetAnim(const char* name) const;
```

34. Overloading functions to add 'const' accessible function is allowed.
```
Anim* GetAnimByIndex(const int index);
const Anim* GetAnimByIndex(const int index) const;
```

35. Avoid use of `const_cast`. Instead create a function that clearly returns an editable version of the object

36. Each class must be in a separate source file unless it makes sense to group several smaller classes.

37. The filename must be the same as the name of the class including upper and lower cases
```
class Anim;

Anim.cpp
Anim.h
```

38. When a class spans across multiple files, these files have a name that starts with the name of the class, followed by an underscore and a subsection name.
```
class RenderWorld;

RenderWorld_load.cpp
RenderWorld_demo.cpp
RenderWorld_portals.cpp
```

39. Platform specific class for "reverse OOP" pattern uses similar naming convention
```
class Renderer;

Renderer.h            // all renderer interfaces called by games
Renderer.cpp     // Renderer's Implementations which are
                      // to all platforms
Renderer_gl.h         // RendererGL interfaces called by
                      // Renderer
Renderer_gl.cpp       // RendererGL implementations
```

40. Use our own version of `Assert` instead of standard c assert

41. Use assert for any assertion you have. Assert is not recoverable. This can be replaced by compiler optimization hint keyword __assume for the release build.

42. Any memory allocation must be done through our own New and Delete keyword.

43. Memory operations such as memset, memcpy and memmove also must be done through our own MemSet, MemCpy and MemMove.

44. Generally prefer reference(&) over pointers unless you need nullptr for any reason. (exceptions are mentioned right below)

45. Use pointers for out parameters. Also prefix the function parameters with out.

    function:
```
void GetScreenDimension(uint32_t* const outWidth, uint32_t* const
outHeight)
{
}
```

    caller:
```
uint32_t width;
uint32_t height;
GetScreenDimension(&width, &height);
```

46. The above out parameters must not be null. (Use assert, not if statement)

```
void GetScreenDimension(uint32_t* const outWidth, uint32_t* const
outHeight)
{
  Assert(outWidth);
  Assert(outHeight);
}
```

47. Use pointers if the parameter will be saved internally.

```
void AddMesh(Mesh* const mesh)
{
  mMeshCollection.push_back(mesh);
}
```

48. Use pointers if the parameter should be generic void* parameter

```
void Update(void* const something)
{
}
```

49. The name of a bitflag enum must be suffixed by Flags

```
enum class eVisibilityFlags
{
}
```

50. Do not add size specifier for enum unless you need that specific size (e.g, for serialization of data members)

```
enum class eDirection : uint8_t
{
  North,
  South
}
```

51. Prefer overloading over default parameters

52. When default parameters are used, restrict them to natural immutable constants such as nullptr, false or 0.

53. Prefer fixed-size containers whenever possible.

54. reserve() dynamic containers whenever possible

55. Always put parentheses for defined numbers
```
#define NUM_CLASSES (1)
```

56. Prefer constants over defines

57. Always use forward declaration if possible instead of using includes

58. All compiler warnings must be addressed.

59. Put pointer or reference sign right next to the type
```
int& number;
int* number;
```

60. Shadowed variables are not allowed.
```
class SomeClass
{
public:
    int32_t Count;
public:
    void Func(const int32_t Count)
    {
        for (int32_t count = 0; count != 10; ++count)
        {
            // Use Count
        }
    }
}
```

61. Take advantage of NRVO, when you are returning a local object. This means you need to have only one return statement inside your function. This applies only when you return an object by value.

62. <<<__restrict keyword

2. Modern Language Features

1. `override` and `final` keywords are mandatory

2. Use enum class always
```
enum class eDirection
{
  North,
  South
```

```
}
```

3. Use `static_assert` over `Assert`
4. Use `nullptr` over `NULL`
5. Use `unique_ptr` when a object lifetime is solely handled inside a class. (i.e. new in constructor delete in destructor)
6. Range based for are recommended where applicable
7. Do not use auto unless it is for a iterator
8. Do not manually perform return value optimization using std::move. It breaks automatic NRVO optimization.
9. Move constructor and move assignment operator are allowed.
10. Use constexpr instead of const for simple constant variables

    ```
    constexpr int DEFAULT_BUFER_SIZE = 65536
    ```

    Instead of

    ```
    const int DEFAULT_BUFER_SIZE = 65536
    ```

    .
11. <<<TBD: constexpr
12. <<<TBD: Lambda
13. <<<TBD: do not use shared_ptr

3. Project Settings and Project Structure

1. Visual C++: Always use property sheets to change project settings
2. Do not disable compile warnings in project settings. Use #pragma in code instead.