Code Standards

General coding standards for Nuit-Entity, these are to be used in all Nuit Entity projects.

Naming Conventions

Classes

Classes and Structs should start with a capital letter.

```
class Foo;
struct Bar;
```

Methods

Member functions are camel case with the first letter being lower case.

```
someClass.someMethod();
```

Where possible use verbs to help identify what a member method does.

```
void            setPosition(const float x);
float           getPosition() const;
bool            isAlive() const;
```

Functions

C style functions should start with a capital with each additional word being capitalised.

```
void ThisIsAFunction();
```

Variables

Variables should follow the following layout.

```
m_memberVar;    // Class member var.
localVar;       // Class/function local var.
g_globalVar;    // Global var.
anotherVar;     // Unnamed namespace var.
```

Where appropriate you can use prefix's and suffix's to indicate extra information about the type.

```
Vec3 m_positionVec;
Foo *m_compFoo;
bool m_isReady;
```

Member variables should be descriptive. Local variables should be shorter, and as a general rule and the more insignificant the variable the more abbreviated it should be, local vars should be defined as close as possible to the point of usage.

Filenames

Filenames are capitalized at start and every preceding word. `.cpp` and `.hpp` are the extensions used for C++ files and `.cc` and `.h` used for C files.

```
// C++
FooBar.hpp
FooBar.cpp
// C
BarFoo.h
BarFoo.c
```

Const Correctness

Const as much as possible.

Const Variables

This makes it easier to quickly identify variables that change throughout the program.

```
bool someMethod(const Type & input)
{
        const float x = 10;
        float y = x;

        y *= 2;

        return true;
}
```

Const Methods

If a method makes no member changes you must indicate it with the const keyword.

```
class Bar {
public:
        void fooMethod() const;
}; // class
```

Mutable

The mutable keyword is fine as long as it isn't invalidating a logical const.

Class and Function Structure

Class Structure

A class should follow the outline, and be formatted with return types and method names aligned.

```
class Foo
{
public:
        explicit        Foo();
                        ~Foo();

        void            setter(const int x);
        int             getter() const;
private:
        m_memberVar;
}; // class
```

Constructors

Should always be explicit unless a valid reason for it to be implicit.

```
// Valid implicit
class Color {
public:
        Color(const uint32_t hex);
}; // class

void someFuncThatTakesAColor(const Color color);
someObj.someFuncThatTakesAColor(0x332244); // Hex is common for color an implicit is fine.

// Not valid implicit
class Color {
public:
        Color(const float r, const float g, const float b);
}; // class

void someFuncThatTakesAColor(const Color color);
someObj.someFuncThatTakesAColor({1, 2, 3}); // Cannot identify easily what object this is, could be a vector
```

Noncopyable

If there is no valid reason for an object to be copied make it non-copyable. If you are inheriting from a noncopyable class use private inheritance.

```
class Foo : private Noncopyable
{
        //.../
}; // class
```

Virtual Methods

If using C++11 use the `override` keyword, else pre C++11 use a `v` at the start of the method.

```
void vMethod();
void method() override;
```

This helps the programmer quickly identify virtual methods without having to check inherited classes.

Static Methods

Static methods are not to be over used. Consider using a function in place for a static method, classes with lots of static functions will be considered bad design.

Initializer Lists

Initialize all variables in the initialize list, in the *correct* order. Initializer lists are to be formatted with the colon or comma preceding the variable.

```
// Kosher
class Foo {
public:
        explicit Foo();
private:
        int x, y;
        float z;
        bool =z
}; // class

// Source
Foo::Foo()
: x(1)
, y(2)
, z(0.1f)
, v(false);
{}
```

Pointers & References

Reference are preferred over pointers, unless it makes sense to return a nullptr consider using a reference.

Pointers are to be asserted regularly, if there is a known reason an assert might fail place a comment next to it. If a pointer is guaranteed to return a valid result, consider returning a reference so that caller doesn't have to assert it.

Raw Pointers

Raw pointers are to be considered for access only to an object and allocations should be done through smart pointers. The following example is fine, as we are not talking about ownership just access.

```
Foo *foo = someObj.getPtr();
```

Raw pointers should be initialized with a `0` pre C++11 and `nullptr` with C++11. **Do not use NULL**.

```
Foo *ptr{};             // Pre C++11
Foo *ptr{nullptr};      // C++11
```

Smart Pointers

With smart pointers consider who owns the object. For example A component is owned by an entity, and an entity is owned by a scene. In this case the ownership is clear and unique pointers are best suited.

When using C++11 use move semantics to transfer ownership.

```
std::unique_ptr<Entity> entity(new Entity);
std::unique_ptr<Component> component(new Component);

entity->addComponent(std::move(component));
```

When ownership is less clear use shared pointers. Shared pointers are not a replacement for raw pointers they address ownership not access.

Use weak pointers only when possible cyclic situations arise with shared pointers, weak pointers are not a replacement for raw pointers they address ownership not access.

References

References are preferred when possible, even when internally you store a pointer return a reference unless there is good reason to return a pointer, or that it might be null.

```
Foo & getFoo() const { return *m_foo; }
```

Scope

Namespace

Named namespaces should be formatted like so with `{` inline. Unnamed namespace is at the discretion of the coder.

```
namespace Outter {
namespace Inner {

class Foo
{
        //.../
}; // class

} // namespace
} // namespace
```

Unnamed Namespace

Unnamed namespaces are encouraged where appropriate, they are considered superior to static/global variables.

```
namespace
{
        const int foo();
        int bar();
}

int Ortho() { return foo; }
```

Globals

Globals variables are discouraged in all forms, including singletons.

Local Variables

Define local variables as close to use as possible.

```
// Bad
int a = 5;
for(;//.../)
{
        a *= i;
}

// Good
for(;//.../)
{
        int a = i;
}
```

Forward Declarations

In general put all forward declarations in one file, or split them up per module, refrain from doing them inline unless they are for internal structs etc.

```
// ExportClassesFwd.hpp
namespace Outter {
namespace Inner {

class Foo;
class Bar;
class Baz;
class Boo;

} // namespace
} // namespace
```

Operator Overloading

Be wary of operator overloading, while it can be clear to you it might not be clear to somebody else. Overloading is acceptable for math related functions where a clear definition is present.

Other Stuff

RAII

When possible avoid init style functions these can cause confusion about how to properly create an object.

Rule of Three/Five

Rule of three states that if you have to define a `Destructor`, `Copy Constructor` or `Copy Assignment Operator` you should define all three. This is usually because we are dealing with memory.

As of C++11 these could be considered the rule of five as ew also have the `Move Constructor` and `Move Assignment Operator`.

Books (Recommended Reading)

- Effective C++
- Effective STL
- The Design of Everyday Things
- Game Coding Complete