ForkENGINE Coding Conventions

Lukas Hermanns

March 2014

- **Namespaces**, **classes**, **structures**, **enumerations**, **enumeration entries**, **functions**, **function objects** (also lambdas), **function interfaces** and **template typename or class arguments** begin in *upper case*:

```
namespace Math
{
class Vector2 { /* ... */ };
}

enum class ColorFlags
{
  Red, Green, Blue
};

auto AbsDotProduct = [](const Math::Vector2f& a, const Math::Vector2f& b)
{
  return
    std::abs(a.x)*std::abs(b.x) +
    std::abs(a.y)*std::abs(b.y);
};

typedef std::function<void (int x, int y)> DrawCallback;

template <typename Type, class Class, int num, ColorFlags flag> /* ... */
```

- **Variables**, **constants** and **static-constants** begin in *lower case*:

```
std::string firstName;
std::string filename;
std::vector<Vertex> vertices;
Video::VertexBufferPtr vertexBuffer;
static const float pi = 3.141592654f;
```

- **Macros** are always entirely in *upper case*. Names are separated by *underscores*:

```
#define MAX_NUM_LIGHTS  8
#define LIGHT_MIN       0.2
#define LIGHT_RANGE     (1.0 - LIGHT_MIN)
```

- Curly brackets are written like it's typical for C++:

```
// Examples:
void Function()
{
  /* Some statements */
}

if (/* Any expression */)
{
  /* Some condition statements */
}

{
  /* Simple block */
}
```

  Don't write Java like 'Egyptian Brackets':

```
// False examples:
void Function() {
  /* ... */
}

if (/* ... */) {
  /* ... */
}
```

An exception for this is when you write very small lambdas:

```
auto IsWhiteSpace = [](char chr) { return chr == '␣' || chr == '\t'; };
```

- All **private** member variables end with underscore ('_'), e.g. "int memberVar_;", "Math::Vector3f position_;" etc.
- Mixed public and private (or protected) class members are allowed, when they are meaningful.

```
class Person
{
  public:
    Person(int age);
    int GetAge() const;
    std::string name;
  private:
    int age_;
};
```

- Every simple getter function should start with "Get", and every simple setter function should start with "Set".

  Example for getter/ setter:

```
float GetHeight() const
{
  return height_;
}
void SetHeight(float height)
{
  height_ = height;
}
```

Example for non-getter/ setter: (because the return value is not a simple member variable, e.g. for a linked list this could take several computations to determine the list size)

```
size_t Size() const
{
  return container.size();
}
void Resize(size_t size)
{
  container.resize(size);
}
```

An exception for getters is when static members are returned:

```
class Mouse
{
  public:
    static Mouse* Instance()
    {
      return instance_;
    }
  private:
    static Mouse* instance_;
};
```

```
class RenderContext
{
  public:
    static RenderContext* Active()
    {
      return active_;
    }
  private:
    static RenderContext* active_;
};
```

- **Hex-literals** are always in *lower case*, e.g. 0xff and not 0xFF.
- Templates use "typename" types if its meant to be used for data types (like int, float etc.).
- Templates use "class" types if its meant to be used for classes.
- **Custom exceptions** are only in the "Fork" main namespace.
- **Setter** or **getter** with a "Description" (e.g. "FontDescription") are named "Get/Set...Desc" and not "Get/Set...Description" to keep the name short but clear.
- For MS/Windows specific code, don't write "hWnd", "hFont". Instead write "windowHandle", "fontHandle" etc., i.e. use clear names and not weird Microsoft notations.
- Make use of the C++11 scoped enumerations:

```
enum class MyEnum
{
  Entry1,
  Entry2,
  /* ... */
};
```

- Flags enumerations should be written like this:

```
struct MyFlags
{
  typedef unsigned int DataType;
  enum : DataType
  {
    Flag1 = (1 << 0),
    Flag2 = (1 << 1),
    /* ... */
  };
};

MyFlags::DataType flags = MyFlags::Flag1 | MyFlags::Flag2;
```

  This is due to the strong typing with the C++11's enum classes:

```
// False example:
enum class MyFlags
{
  Flag1 = (1 << 0),
  Flag2 = (1 << 1),
  /* ... */
};

MyFlags flags = MyFlags::Flag1 | MyFlags::Flag2; // <-- Error
```

- Error messages, telling that something failed, should be written as "[Do-]ing ... failed" and not "Could not [Do]":

```
// Usage example:
IO::Log::Error("Loading␣texture␣\"" + filename + "\"␣failed");

// False example:
IO::Log::Error("Could␣not␣load␣texture␣\"" + filename + "\"");
```

- Event handler callbacks should never be implemented via function objects. Use class inheritance instead.

```
// Example:
class Widget
{
  public:
    // This is the event handler for all "Widget" events.
    class EventHandler
    {
      public:
        virtual ~EventHandler()
        {
        }
        virtual void OnResize()
        {
        }
      protected:
        EventHandler() = default;
    };

    typedef std::shared_ptr<EventHandler> EventHandlerPtr;

    void AddEventHandler(const EventHandlerPtr& eventHandler)
    {
      if (eventHandler)
        eventHandlers_.push_back(eventHandler);
    }

  private:
    std::vector<EventHandlerPtr> eventHandlers_;
};
```

Only when a single small function is required, use a function object:

```
// Example:
class SceneNodePrinter
{
  public:
    // Callback interface
    typedef std::function<void (const std::string& text)> SimpleCallback;

    void PrintSceenNode(
      Scene::SceneNode* sceneNode, const SimpleCallback& callback
    );

  private:
    SimpleCallback callback_;
};

SceneNodePrinter printer;

printer.PrintSceneNode(
  sceneNode,
  [](const std::string& text)
  {
    IO::Log::Message(text)
  }
);
```

- Never use "NULL"! Instead use "nullptr", except for handles (WinAPI); use "0" for handles, because handles are not always typedefs to pointers.

- Avoid too much implicit programming, e.g. the core classes like "Vector3" only have explicit constructors, when only one input parameter is used:

```
explicit Vector2(float size);
explicit Vector3(float size);
explicit Vector4(float size);
```

- Boolean class members should start with "is", "has", "was" etc. telling what kind of boolean this is.

```
bool isEnabled;
bool hasChildren;
bool wasMovedLastFrame;
bool acceptModifications;
bool preventForPowerSafe;
```

- In a class hierarchy, when up-casting is required, the base class should have the following function interface:

```
class BaseClass
{
  public:
    enum class Types
    {
      Type0,
      Type1,
      //...
    };
    virtual Types Type() const = 0;
};
```

  Each derivated class must then implement the function "Type", which returns the respective class type.

  Here is an example class:

```
// Base class
class Texture
{
  public:
    enum class Types
    {
      Tex1D,
      Tex2D,
      Tex3D,
      TexCube,
    };
    virtual Types Type() const = 0;
    //...
};

// Sub class
class Texture1D : public Texture
{
  public:
    Types Type() const
    {
      return Types::Tex1D;
    }
    //...
};
```

- Special enumeration entries are written with two clasping underscores:

```
enum ComponentTypes
{
  // Special entry for functions which always return an enumeration item.
  __Unknown__,

  Asset,
  Entity,

  // Special entry which is used to determine the number of items.
  __Num__ = 2,
};
```

- 'Enumeration entry to string' conversion functions are always *static functions* and called **"<EnumName>ToString"** where <*EnumName*> specifies the singular enumeration name, e.g.:

```
class Entity
{
  enum Types
  {
    Model,
    Camera,
    Light,
  };
  static std::string TypeToString(const Types type);

  enum Modes
  {
    Editing,
    Moving,
  };
  static std::string ModeToString(const Modes mode);
};
```

- **Code documentation** is written in *doxygen* syntax. Keep the same order of documentation entries (i.e. parameters, return type, remarks, etc.):

```
/**
This is any function.
\param[in] x Specifies the X coordinate.
\param[in,out] y Specifies the Y coordinate.
\return Raw pointer to an integer array.
\remarks This is some additional and more detailed information.
\note Some warning information, e.g: make sure to delete the integer pointer!
\todo Some todo information.
\throws InvalidArgumentException If 'x' is negative.
\see SomeOtherFunction
*/
int* AnyFunction(int x, int& y);

//! Single line documentation.
void SmallFunction();
```

- **Singleton** classes are implemented as follows:

```
class SingletonExample
{

  public:

    SingletonExample(const SingletonExample&) = delete;
    SingletonExample& operator = (const SingletonExample&) = delete;

    static SingletonExample* Instance()
    {
      static SingletonExample instance;
      return &instance;
    }

  private:

    SingletonExample() = default;

};
```

- **Ranges** in documentation are written as follows:

```
/**
\param[in] index Range is [0 .. number-of-indices).
\param[in] interpolator Range is [0.0 .. 1.0].
\param[in] anyFloat Range is (-inf .. +inf).
*/
void ExampleFunction(size_t index, float interpolator, float anyFloat);
```

- **Identification numbers** are written in short as "ID" and not "Id" or "id".