RISING TIDE's 2018 Style Guide :

Comments

File Header Comments Style

```
/******************************************************************************/
/*!
  @file   file_name.hpp
  @author Author Name
  @par    project: GAM200 - Tide Engine 2018
  @brief
      A small description or something. I don't think it's necessary though.

      Copyright (C) 2018 DigiPen (USA) Corporation.
*/
/******************************************************************************/
```

General Comment Style

- C++ or C style comments are both accepted
  - You should prefix notes with "NOTE(YourName):"
    - EX:

      -
        ```
        // NOTE(Shareef): This is a C++ style comment.
        /* NOTE(Shareef): This is a C style comment. */
        ```
  - And TODOs should contain the name of the person who's job it is to get done.
    - EX:

      -
        ```
        // TODO(Amir): Get this shit fixed.
        ```

Include Comments

- You should comment your includes for what you are using the include for, this helps document

Indentation

Tabs vs Spaces

- We use **SPACES** not tabs.
- Preferably 2 spaces since that's how Mead likes it but honestly doesn't matter to me. Just be consistent and be reasonable, 4 should be the max.

Namespaces

- Code within namespaces should be indented:
  - Right:

```
  // my_class.hpp
namespace engine
{
  namespace detail
  {
    class InternalClass
    {
      protected:
        void internalImpl();
      ...

    };
  }

  class MyClass
  {
    public:
      MyClass() = default;
      ...

  };
}

  // my_class.cpp
namespace engine
{
  namespace detail
  {
    void InternalClass::internalImpl()
    {
      ...
    }
  }
}
```

  - Wrong:

```
  // my_class.hpp
namespace engine
{
namespace detail
{
class InternalClass
{
  protected:
    void internalImpl();
  ...

};
}
class MyClass
{
  public:
    MyClass() = default;
    ...

};
}

  // my_class.cpp
namespace engine
{
namespace detail
{
void InternalClass::internalImpl()
{
  ...
}
}
}
```

Switch Statements

- Switch cases should be indented.
  - I prefer to also put indented braces since I find it easier to read that way.
  - EX:

```
  // The braces are optional unless you declare a variable inside of a case.
  // but just to be universal I just add them regardless.
switch (condition)
{
  case CONDITION_ONE:
    {
      ++i;
      break;
    }
  case CONDITION_TWO:
    {
      --i;
      break;
    }
  default:
    {
      i += 5;
      break;
    }

}
```

Spacing

- **Do not** place spaces around unary operators.
- **Do** place spaces around binary and ternary operators.
- Place spaces around the colon in a range-based for loop.
- Do not place spaces before comma and semicolon.
- Place spaces between control statements and their parentheses.
- Do not place spaces between a function and its parentheses, or between a parenthesis and its content.
- When initializing an object, place a space before the leading brace as well as between the braces and their content.

Line Breaks

- Each statement should get its own line.
- An else statement should go on the same line as a preceding close brace if one is present, else it should line up with the if statement.
- An else if statement should be written as an if statement when the prior if concludes with a return statement.

Floating Point Literals

- Unless required in order to force floating point math, do not append .0, .f and .0f to floating point literals.

Braces

- Control clauses without a body should use empty braces:

Null, True, False, 0

- In C++, the null pointer value should be written as nullptr. In C, it should be written as NULL.
- Tests for true/false, null/non-null, and zero/non-zero should all be done without equality comparisons.

File Naming Conventions

- File names should be all lowercase with words separated by underscores:
  - Ex: "batch_renderer.hpp"
- File Extensions:
  - If you are writing C++ code use ".hpp" and ".cpp" file extensions
  - If you are writing C code use ".h" and ".c" file extensions
  - This is because for some Compilers you can make them interpret the different files extensions in different ways.
- • If you can put the code files in the actual directory in the SVN then add it to the project. This is so the project folder is not cluttered up with random ass files everywhere.

Free Function Naming Conventions

Class / Struct Naming Conventions

- Classes and Structs should use the CamelCase naming convention.
  - C++ Example

```
  // Example C++ Class
class CamelCaseClass
{
  ... fields and methods ...

};
```

  - C Example

```
  // Example C Struct
  // The 'CStyleCamelCaseStruct_t' can be left off but will be
  // there in my code,
  // it's for easy C and C++ Interop.
typedef struct CStyleCamelCaseStruct_t
{
  ... fields and shit ...


} CStyleCamelCaseStruct;
```

Member Variable Naming Conventions (C++ Only)

Method Naming Conventions (C++ ONLY)

Overiding Methods

Functions that Operate on Struct(s) (C ONLY)

- When using C you should try to emulate C++ conventions to make things easier for other programmers using C++ to interop with your code.
- You can do this by defining these types of functions for you type:

```
  // To Initialize 'Stack' Allocated Objects
  // Ctor stands for "Constructor"
void             <struct_name>Ctor(<struct_name>* self, argument...);

  // To Destroy 'Stack' Allocated Objects
  // Dtor stands for "Destructor"
void             <struct_name>Dtor(<struct_name>* self);

  // To Initialize 'Heap' Allocated Structs
  // Using ‘new’ emulating C++
  // This function should just malloc / calloc the self pointer and then call
  // the 'Constructor' method.
<struct_name>*  <struct_name>_new(arguments...);

  // To Destroy 'Heap' Allocated Structs
  // Using ‘delete’ emulating C++
  // This function should just call the 'Destructor' method and then free
  // the self pointer.
void             <struct_name>_delete(<struct_name>* self);
```

- The first parameter should almost always be a pointer to the object you are manipulating so by convention I use the word “self” (stolen from languages such as Lua, Swift, Python etc. . .) although it can be anything. Just make it make sense.

- Functions that operate on structs formatting:

  ```
  <return_type> <struct_name>_<verb>(<struct_name>* self, arguments...);
  ```

  – For Example:

    ```
    void CamelCase_update(CamelCase* const self, double deltaTime);
    ```

- By using my rules of only using verbs for function names we can get rid of the use of “get/Get” for getters.

  – Function that just get a piece of state should just return that object.

  - Also generally your getters should return a **CONSTANT** pointer to that object.

```
BAD:  Animation*        GameObject_getAnimation(CamelCase* const self);
GOOD: const Animation*  GameObject_animation(const CamelCase* const self);
```

- Functions that return state as a boolean should use being verbs suchs as “is” and “are”.

```
bool CamelCase_isDead(const CamelCase* const self);
```

Enums Naming Conventions

- Enums should follow the same naming conventions as structs, but enum values should be in all capitals with a prefix.
- The enum values should end in a "_MAX" format to denote the maximum value the enum has. This may be left off but is often useful for error checking and using it as a loop condition.
- Enums in C++ should be “enum class” since they are more type safe and allow you to define the underlying datatype.
  - C++ Example

```
  // NOTE(Shareef):
  //   C++ Example Enum Class:
  //     The Prefix in this case is uncessary since
  //     you must always use the scope resolution operator ("::")
  //     to access the value.
enum class TestEnumCpp
{
  TE_FIRST_VALUE,
  TE_ANOTHER_ONE,
  TE_MAX
};

  // NOTE(Shareef):
  //   Even better is to define the underlying type if you know
  //   the maximum value of the enum.
  //   This will help save on space if done diligently.
enum class BetterTestEnumCpp : uint8_t
{
  FIRST_VALUE,

    // NOTE(Shareef) : MAX may be a bad name since MAX is
    //                 a macro defined by some headers.
  MAX_VALUE
};
```

– C Example

```
// NOTE(Shareef):
//   C Example Enum:
//   The 'TestEnumC_t' can be left off but will be there in my code.
//   The Prefix 'TE' just stands for TestEnum but you can prefix it anyway
//   you want. Just be consistent with all values.
typedef enum TestEnumC_t
{
  TE_FIRST_VALUE,
  TE_ANOTHER_ONE,
  TE_MAX

} TestEnumC;
```

Typedef (and Using)

- In C++ there is a new 'using' keyword.
  - For proper C++ style you should prefer to use *using* rather than *typedef* although there is no technical difference between the two.
  - (Although) *using* does provide some extra features if you are a template god.

```
  // NOTE(Shareef): This is prefered.
using MyFunctionPtr = void (*)(int someArg);
  // NOTE(Shareef): You can see how this is a bit messier with no advantage.
typedef void (*MyFunctionPtr)(int someArg);
```

Macros

- Macros should be declared at the top of the file and in all capitals:

```
  // NOTE(Shareef): This macros gives me the max of two values.
#define MAX(a, b) ((a) > (b)) ? (a) : (b)
```

- If you can '#undef' ing the macro when you're done with it is a good idea.

```
  // NOTE(Shareef): This is so this macro doesn't clutter the global
  //   'namespace'
#undef MAX(a, b)
```

Other Code Style Notes

- Avoid using global variables as much as possible, if you absolutely needs a global variable make sure to have is scopes as small as possible. Such as using the ‘static’ keyword in a ‘*.c’ file.
- I prefer having curly bois (‘{‘ and ‘}’) on a new line but won’t enforce it.