Full Sail Game Development Engine Coding Standard

Table of Contents

Naming Conventions

1.1) All names, unless otherwise specified, should be in "camel case".

a) **Type names must be capitalized, and Names representing template types should be a single uppercase letter.**

b) **Variable names must be lower case, except constants, which must be all uppercase using underscore to separate words. (This includes enumerated values.)**

c) **Names representing methods or functions must be written in camel case starting with uppercase.**

d) **Abbreviations and acronyms should not be capitalized (html, dvd, etc.)**

```
enum Color { COLOR_RED, COLOR_GREEN, COLOR_BLUE };
Color rgbBackgroundColor;
const int PI = 3.14159;

template <typename T>
T GetMax(T lhs, T rhs)
{
    return (lhs > rhs ? lhs : rhs);
}
```

1.2) Use descriptive variable names.

a) **Avoid single letter variable names.**

b) **Generic variables should have the same name as their type.**

c) **The prefix *num* should be used for variables representing a number of objects.**

d) **The suffix *No* should be used for variables representing an entity number.**

e) **Plural form should be used on names representing collections of objects.**

f) **The prefix *is* or *are* should be used for boolean variables and never as a negation.**

g) **Static variables must start with "the" (theWindowHandle) and global variables with "global" (globalState).**

h) **Attributes (member variables) should not be prefixed. Parameters may be prefixed with an underscore where there may otherwise be confusion.**

```
bool PointSet::IsInSet(Point _point) // NOT: CheckforElement(Point value)
{
    bool isElement = false;          // NOT: isNotElement or elementTrue
    for (int pointNo = 0; pointNo < numPoints; pointNo++)
    {
        if (points[pointNo] == point)
            is element = true;
    }
    return isElement;
}
```

1.3) Use descriptive, but not overly verbose, method and function names.

a) **All method and function names should start with a verb.**

b) **The name of the object is implicit and should be avoided in a method name.**

c) **Complement names must be used for complement operations (get / set, add / remove, etc.)**

d) **The terms *get* and *set* should be used where an attribute (member variable) is accessed directly.**

```
int Line::GetLength() // NOT: GetLineLength() or ReturnLength()
{
    return length;
}
```

Source File Conventions

2.1) All definitions should reside in source files, except inline functions.

```
class MyClass
{
public:
    inline int GetValue() { return value; } // OK

    int ComputeValue()                      // NO!
    {
        if (value < 0)
            return 0;

        return value;
    }
...
}
```

2.2) Source code lines should be readable in typical situations.

a) **File content must be kept within 100 columns.**

b) **The incompleteness of split lines must be made obvious.**

c) **Tabs should be used for indentation and spaces for alignment.**

```
cout << "I am " << myName << ". My friend is " << friendName <<
        ". We went to school together for many, many years.\n";
```

2.3) Use "#pragma once" to prevent duplication of header files.

```
#pragma once // NOT: #ifndef COM_COMPANY_MODULE_CLASSNAME_H ...
```

Statements

3.1) Variables should be used efficiently and should be used in the clearest (most obvious) manner possible.

a) **Class members must be ordered *public*, *protected* and *private*, with all sections identified explicitly.**

b) **Variables must never have dual meaning or be reused for different purposes.**

c) **Variables should not be implicitly tested for zero.**

d) **Use of global variables should be minimized.**

e) **Variables should be declared in the smallest scope possible.**

```
// Yes!
class Singleton
{
public:
    Singleton& GetReference();
    void ResetSingleton();
    int GetResetCount();
private:
    Singleton* singleton;
    static int theResetCount;
}
...
Singleton& Singleton::GetReference()
{
    if (nullptr == singleton)
        ResetSingleton();

    return *singleton;
}
```

```
// NO!
int globalResetCount;

class Singleton
{
    Singleton* singleton;
public:
    Singleton& GetReference();
    void ResetSingleton();
    int GetResetCount();
}
...
Singleton& Singleton::GetReference()
{
    if (!singleton)
        ResetSingleton();

    return *singleton;
}
```

3.2) Loops should be constructed to maximize readability.

a) **for() loop statements should be limited to control statements and assignments.**

b) **do while() loops should be used only when they clearly improve readability of code.**

c) **Loop variables should be initialized as closely as possible to the loop itself.**

d) **The break and continue statements should only be used when necessary for functionality or readability.**

e) **The form `while(true)` should be used for infinite loops.**

```
while (true) // NOT: do {...} while (true), for(;;), or while(1)
...

int index, sum = 0;

for (index = 0; index < length; index++) // NOT: for(index = 0, sum = 0;..
    sum += value[index];
```

3.3) Conditional statements should be constructed in a manner that allows for quick navigation.

a) **For if-else block pairs, the nominal case should be the *if* block and the exception the *else* block.**

b) **The conditional statement should be put on a separate line from the result statement or block.**

c) **Checks for equality against a constant should start with the constant (to prevent errors.)**

d) **Never perform assignment within a conditional statement.**

```
// Yes!
int errorCode = ReadFile(filename);

if (0 == errorCode)
    cout << "File was read successfully!\n";
else
    cerr << "Error #" << errorCode << " when reading file.\n";

// NO!
if (errorCode = ReadFile(filename))
    cerr << "Error #" << errorCode << " when reading file.\n";
else cout << "File was read successfully!\n";
```

3.4) Avoid techniques that obscure meaning or otherwise make code hard to read and understand.

a) **Whenever possible, use named constants in place of magic number literals in code.**

b) **Floating point literals should be written with at least one number before and after the decimal.**

c) **Function return types should be explicitly identified.**

d) ***goto* statements should be avoided unless they clearly improve readability.**

```
int ComputeRoundedCost() // NOT: ComputeRoundedCost()
{
    const float HEURISTIC_WEIGHT = 0.5f;  // NOT: HEURISTIC_WEIGHT = .5f;
    const float HEURISTIC_ADDITION = 2.0f;// NOT: HEURISTIC_ADDITION = 2f;
    ...
    heuristicCost *= COST_WEIGHT;   // NOT: heuristicCost *= 0.5f;
    heuristicCost += COST_ADDITION; // NOT: heuristicCost += 2.0f;
    ...
    return (int) heuristicCost;
}
```

Layout and Comments

4.1) Statement blocks should be laid out in a consistent and readable manner.

a) **With the exception of inline functions, curly braces should always fall on their own lines.**

b) **Nested elements should be indented using a single tab.**

c) **Block braces may be omitted only when the entire block is a single line.**

```
// Yes!
if (isReady)
{
    while (!IsDone())
    {
        DoThisThing();
        DoThatThing();
    }
}
```

```
// NO!
if (isReady) {
    while (!IsDone()) {
        DoThisThing();
        DoThatThing();
    }
}
```

```
//NO!
if (isReady)
    while (!IsDone())
      {
        DoThisThing();
        DoThatThing();
      }
```

4.2) White space should be used to create clear distinction between different variables and statements.

a) **Operators between operands should be surrounded by a space character on either side.**

b) **Control words, commas, and loop semicolons should be followed by a space, except before a semicolon.**

c) **For inheritance and initialization lists, colons should be prefixed by a space, but not in case statements.**

```
while (!IsDone())                              // NOT: while(!IsDone())
{
    switch (costMode)
    {
    case HEURISTIC:                            // NOT: case HEURISTIC :
        distance = sqrt(dx * dx + dy * dy); // NOT: sqrt(dx*dx+dy*dy);
        break;                                 // NOT: break ;
    case ACCUMULATED:
    default:
        distance++;                            // NOT: distance ++;
        break;                                 // NOT: break ;
    }
    CalculateCost();
}
```

4.3) Comments should be descriptive (but not verbose), concise, and well-integrated.

a) **Comments should be minimized by writing self-documenting and clear code that requires less explanation.**

b) **Single line comment style should be used whenever possible (including when writing multi-line comments) in order to allow block comments to be used for debugging purposes.**

c) **Class header and method header comments should be written to the Doxygen comment conventions.**