Midnight Slice Madness
Coding Standards
Version 1.0

Contents

Changelog

Date	Section(s) Changed	Changed By	Notes
03/14/2024	*	Andrew Plum	Created coding standards document

About this document

This document holds the Asset Flip Studios Coding Standards for the C# programming language. It provides a guide for developers to format their code to improve consistency and readability.

1. Naming

This section covers naming standards for variable names, class names, function names, and file names.

1.1 General Naming Rules

In all cases, names should describe the purpose of the item being named. Developers should avoid names that include abbreviations and acronyms that are not immediately recognizable to other developers. As stated in the Google C++ Style guide, "…an abbreviation is probably OK if it's listed in Wikipedia." [1].

The following code provides examples on how to select names.

```
/* Examples of names that do not follow the coding standards */

int fooBarBaz;     //Variable name is not descriptive.

void GetFRQ();     //Function name uses an uncommon acronym.

class LvlLdr{}     //Class name uses unnecessary abbreviations.
```

```
/* Examples of names that follow the coding standards */

int playerSpeed;  //Variable name is descriptive.

void GetHTTPResponse(); //Function name uses a well-known acronym.

class LevelLoader{}      //Class name is fully spelled out
```

1.2 Naming Classes

Class names should follow the general naming rules outlined in section 1.1. In addition, class names should be written in PascalCase.

When naming Interfaces the name should begin with an `I`.

The following code provides examples of class names.

```
/* Class Names */

class GameManager{}      //Class name uses PascalCase

class Player{}     //One-word class name is capitalized

interface ISampleInterface{}  //Interface name begins with 'I'
```

1.3 Naming Functions and Function Parameters

Function names and parameters should follow the general naming rules outlined in section 1.1. In addition, function names and parameters should be written in PascalCase.

Virtual and override function names should begin with the `v_` prefix.

The following code provides examples of function names and parameters.

```
/* Function Names and Parameters */

int FindArea(int length, int width);  //Example with PascalCase

void Exit();      //One-word function name has uppercase first letter

virtual int v GetHealth();     //Example of a virtual function

override int v_GetHealth();    //Example of an override function
```

1.4 Naming Variables

Variables names should follow the general naming rules outlined in section 1.1. Additional formatting should be applied to variable names based on the access level and any modifiers (i.e. the 'static' keyword) that are applied to a variable.

1.4.1 Public Variables

Public variables should be named using camelCase. If a public variable is static, the name should begin with the s prefix.

The following code provides examples of public variable names.

```
/* Public Variable Names */

public int publicVariable;     //Public variable example

public GameObject player;      //One-word public variable example

public static int sStaticVariable;  //Public static variable example
```

1.4.2 Private Variables

Private variables should use camelCase for the rest of the name. If a private variable is static, the name should begin with the s prefix.

The following code provides examples of private variable names.

```
/* Private Variable Names */

private int privateVariable; //Private variable example

private float speed;     //One-word private variable example

private static sStaticVariable;      //Static private variable example
```

1.4.3 Protected Variables

Protected variables should begin with the p_ prefix and then use camelCase for the rest of the name. If a protected variable is static, the name should begin with the s prefix.

The following code provides examples of protected variable names.

```
/* Protected Variable Names */

protected int p_protectedVariable;  //Protected variable example

protected int p_damage; //One-word protected variable example

protected static int sp_staticVariable;    //Static protected variable example
```

1.4.4 Constant Variables

Constant variable names should be all UPPERCASE with an underscore between each word in the name.

The following code provides examples of constant variable names.

```
/* Constant Variable Names */

const int CONSTANT_VARIABLE;  //Constant variable example
```

1.5 File Names

File names should follow the general naming rules outlined in section 1.1. In addition, these file names should be all lowercase with underscores separating each word.

The following are examples of properly formatted file names.

```
/* File Names */

file_name.txt

high_score.txt

config.json
```

1.6 Naming exceptions

If a variable name has a common purpose or meaning, then that name can be used even if it doesn't follow these naming standards. An example of this includes using i, j, and k for iterating in a for loop.

2. Layout

This section details how to organize and format code to improve consistency and readability.

2.1 General Layout Rules

In all cases, the indentation should be 4 spaces. Any statements that use a curly brace start/end the statement, like functions, classes, conditional statements, and iterating statements should also follow these rules:

1) The opening brace should be placed on the line below the statement.
2) The closing brace should be placed on the line after the end of the statement's content.
3) All content within a statement should appear between the opening and closing braces, and there should be nothing placed on the same line as an opening/closing brace.

The following code shows a general example of how these statements should be formatted.

```
/* General Layout Example */

Statement
{
    /* Place all statement content here */
}
```

2.2 Class Layout

Classes should be written in a way that minimizes coupling and maximizes cohesion. When writing classes, make sure that the content within a class is related (high cohesion) and that only the necessary data can be passed between objects (low coupling).

Classes should follow the general layout rules outlined in section 2.1. In addition, classes should be laid out in the following order for consistency:

1) Class data members organized in order of most to least visible.
    a. i.e. public->protected->private.
2) The Unity Start() method (if used).
3) The Unity Update() method (if used).
4) Any other Unity methods like Awake() or OnCollisionEnter2D()
5) Other custom-made functions

Each separate element of a class should also be separated by an empty line.

The following code provides an example of proper class layout.

```
/* Class Layout Example */

public class ExampleClass : Examples
{
      public int publicNumber;
      public GameObject player;

      protected float p_protectedNumber;
      protected bool p_isPlayerDead;

      private int _privateNumber
      private float _playerHealth
```

```
	void Start()
	{
		/*Code to run on first frame*/
	}

	void Update()
	{
		/*Code to run every frame*/
	}

	void OnCollisionEnter2D(Collision2D collide)
	{
		/*Code to run on collision*/
	}

	public exampleClassMethod()
	{
		/*Example Class Method Code*/
	}
}
```

2.3 Function Layout

Functions should be written in a way that minimizes coupling and maximizes cohesion. When writing a function, keep them brief and make sure they are being written to accomplish a specific task (high cohesion). When a function requires parameters, make sure only the required data is passed into the function (low coupling).

Functions should follow the general layout rules outlined in section 2.1. Function parameters, if any, should be separated by a comma and a space. Functions should start with variable definitions followed by an empty line.

The following code provides an example of proper function layout.

```
/* Function Layout Example */

public bool IsCoprime(int number1, int number2)
{
	int temporaryNumber;

	while(number2 != 0)
	{
		temporaryNumber = number1;
		number1 = number2;
		number2 = temporaryNumber % number2
	}

	if(number1 != 1)
	{
		return false;
	}
	else
	{
		return true;
	}
}
```

2.4 If-Statement Layout

If-statements should follow the general layout rules outlined in section 2.1. They should be written in a way that is readable and easy to understand. To facilitate this, if-statements should follow these guidelines:

1) There should be a space between all variables and operators.

```
/* If-Statement -- Seperating variables and operators */

if(number1 <= number2)
{
   /*If-statement code*/
}
```

2) Any operations that need to take place within an if-statement should be placed in parentheses.

```
/* If-Statement -- Grouping operations */

if(number1 != (number2 * number3))
{
   /*If-statement code*/
}
```

3) If the if-statement has more than one comparison, each comparison should be placed in parentheses.

```
/* If-Statement -- Grouping comparisons */

if((number1 < number2) && (number2 > 0))
{
   /*If-statement code*/
}
```

4) If an if-statement contains more than 2 comparisons, then each comparison should be placed on a separate line with the logical operator at the end of each line.

```
/* If-Statement -- More than 2 comparisons */

if((number1 > number2) ||
(number1 > number3) &&
(numebr1 != 0))
{
   /*If-statement code*/
}
```

5) When using else if or else statements, make sure to follow all the above rules for each statement. If more than 3 else if statements would be used, consider using a switch statement outlined in section. 2.5 instead.

```
/* Else if/Else Statements */

if(number1 > number2)
{
   /*If-statement code*/
}
else if(number1 >= 0)
{
   /*Else-if statement code*/
}
else
{
   /*Else statement code*/
}
```

2.5 Switch Statement Layout

Switch statements should follow the general layout rules outlined in section 2.1. When writing a switch statement make sure to include a default statement.

The following code provides an example for using switch statements.

```
/* Switch Statement Example */

switch(switchVariable)
{
      case 1:
            /*Case 1 code*/
            break;
      case 2:
            /*Case 2 code*/
            break;
      case 3:
            /*Case 3 code*/
            break;
      case 4:
            /*Case 4 code*/
            break;
      case 5:
            /*Case 5 code*/
            break;
      default:
            /*Default code*/
            break;
}
```

2.6 While and Do-While Loop Layout

While loops should follow the layout rules outlined in section 2.1. When writing the conditions for the while or do-while loop, follow the same layout rules as if-statements in section 2.4. Additionally, while and do-while loops must have a way to escape the loop.

The following code provides an example for both while and do-while loops.

```
/* While Loop Example */

Random randomNumber = new Random();
while(loopFlag == true)
{
	randomNumber.Next();
	if((randomNumber % 2) == 0)
	{
		loopFlag == false;	//Can also use 'break' to exit loop
	}
}

/* Do-While Loop Example */
do
{
	randomNumber.Next();
	if((randomNumber % 2) == 0)
	{
		loopFlag == false;	//Can also use 'break' to exit loop
	}
} while(loopFlag == true);
```

2.7 For Loop Layout

For loops should follow the layout rules outlined in section 2.1. When defining the initialization, the condition, and the post-loop housekeeping use spaces between variables and operators, except for '++' or '--' operators.

The following code provides an example of how to format loops.

```
/* For Loop Example */

for(int i = 0; i < 5; i++)
{
	/*For loop code*/
}
```

2.8 Nesting Statements

In general, nesting these statements is okay, but anything beyond about 4-levels deep should be avoided when possible.

3. Code Comments

Comments are important to communicate ideas to other developers. When writing comments, make sure to focus more on *Why* a section of code exists and less of *What* the code does. Comments should be included whenever there is complex language that needs an extra explanation to understand quickly, like a complicated if statement or REGEX statements.

This section will focus on using multi-line comments, inline comments, function headings, and file headings.

3.1 Single-line Comments

If a comment is created on a single new line, use the '`// Comment `' syntax. The '`/* Comment */`' syntax is also accepted for single-line comments.

The following is an example of a multi-line comment.

```
// Single-line Comment Example

// This is an example of a Single-line comment
```

3.2 Multi-line Comments

If a comment is created on multiple new lines, use the '`/* Comment */`' syntax. When creating multi-line comments place each idea within the comment on its own line and place the beginning '`/*`' and ending '`*/`' both on their own lines.

The following is an example of a multi-line comment.

```
// Multi-line Comment Example

/*
This is an example of a Multi-line comment
Make sure to place new ideas on seperate lines
Begin new lines with an asterisk and a space, making sure they line up
*/
```

3.3 Inline Comments

If an inline comment is used, it should be short and to the point. Inline comments will use the '`//`' syntax. If several inline comments are used in a row, make sure that the comments in that block line up.

The following is an example of an inline comment.

```
// Inline Comment Example

private _foo;                  // Foo variable to give example
static private s_fooBarBaz;    // Variable to add more to the example
public testVariableOne;        // Notice that comments in this block lines up
```

3.4 Function Headings

Each function in your code should be preceded by a function heading. This comment should include a high-level view of the function and how it interacts with other functions, variables, and classes. The function heading should use a multi-line comment and follow the formatting rules given in section 3.1.

The following is an example of a function heading.

```
// Function Heading Example

/*
This function is here to identify whether two numbers are coprime
It applies the Euclidean Algorithm to find the GCD
When writing function headings, make sure to line up the asterisks
Make sure to note how this function interacts with other functions
*/
public bool isCoprime(int number1, int number2)
{
```

3.5 File Heading

At the beginning of each script there should be a file heading that explains what the file is and why it is there. This comment should include the developer's name, the developer's role, the project name, and any class relationships (i.e. inheritance) that exist within the file.

The following is an example of a file heading.

```
/* File Heading Example */

/*
Name: Andrew Plum
Role: Team Lead 4 -- Project Manager
Project: Midnight Slice Madness

This file contains the definition for the ExampleClass
This class provides an example for the coding standards
It inherits from Examples and IExamplesInterface
This File Heading would also be a good place to mention design patterns
*/

 public class ExampleClass : Examples, IExampleInterface
{
```