# Games in C++:
# CODING STYLE V1.11

## ABSTRACT

Throughout the industry, it is common for a series of stylistic coding guidelines to be used. These are not designed to tell you how to program, rather to help maintain a uniform style throughout the codebase. At best, they allow all programmers on a single project the ability to easily follow and work within each other's codebases. At worst, they provide a sensible structure and some reasonable recommendations for a modern maintainable codebase.

## CONTENTS

**ABSTRACT** ..... 1

**NAMING** ..... 3

2. Globals ..... 3

3. Pointers ..... 4

4. Enumerated Types ..... 4

5. Functions ..... 4

6. Classes ..... 5

7. Iteration ..... 5

**LAYOUT** ..... 6

8. C++ Header / Source Files ..... 6

9. Definitions ..... 7

10. Include Statements ..... 7

11. Line Breaks / Lengths ..... 8

12. Split Lines ..... 8

13. Braces / Line Indentations ..... 9

14. Function Spacing ..... 9
15. Block Layout ..... 10
16. Class Layout ..... 10
17. For Statement Layout ..... 11
18. While Statement Layout ..... 11
19. IF / Else Layout ..... 12
20. Switch Statement Layout ..... 13
21. Grouping ..... 14
**COMMENTING** ..... 15
22. Confusing Code ..... 15
23. Inline Comments ..... 15
24. Doxygen ..... 16
**MISC** ..... 17
25. Pointers, Reference Symbols & Null Pointers ..... 17
26. Namespaces ..... 17

## NAMING

### 1. Variables

```
// class variable
int player_health = 0;

// local variable
int old_total = running_total;
int running_total += func();

// scratch variable
int x = 0;
```

- Variables should be named sensibly. There is no need to create excessively long named variables. Local variables or “scratch” variables may have shorter names than those with longer lifespans i.e. class members.
- Variables should start with a lowercase letter and each word should be separated by an underscore whilst maintaining the same case.
- Variables should always be initialised when declared and placed on a new line.

### 2. Globals

```
Player players[2];
int main()
{
	Player players[2];
	for (int i = 0; i < 2; ++i)
	{

	}
}
```

- C++ does not need global variables.
- Consider alternatives to global functions, data or static file scope variables.
- Globals lead to clashes and obscured data often causing undefined behaviour.

## 3. Pointers

```
int* my_game_score_ptr = new int(); // not like this
int* my_game_score = new int();     // but like this
```

- There is no need to state a variable is of type pointer in its name.
- Modern day IDE's will quickly advise you of a variables type i.e. intellisense.

## 4. Enumerated Types

```
enum class PlayerType : int
{
	HUMAN_CONTROLLED = 1,
	AI_CONTROLLED = 2
};
```

- The values defined inside of an enum should be all uppercase with underscores.
- Prefer enum classes as they prevent namespace pollution due to obeying scope.

## 5. Functions

```
bool attackEnemy(int enemy_id) const
{
	// do something
	return false;
}
```

- Functions or methods should always start with a lowercase letter.
- If composed of more than one word, the following words should also start with an uppercase letter.

## 6. Classes

```
class MyTotesAmazeGame
{

};
```

- Class names should always start with an uppercase letter.
- IF composed of more than one word the following words should start with an uppercase letter.

## 7. Iteration

```
const int MAX_COUNT = 25;
int my_array[MAX_COUNT];

for (int i = 0; i < MAX_COUNT; ++i)
{
	for (int j = 0; j < 5; ++j)
	{
		// nested form, j.k.etc
	}
}
```

- Whilst iterating through containers or arrays it is sensible to use the standard notation of i, j, k etc.

NB: On a side note; please do not use Hungarian ‘type’ notation. its intended usage was based not on variable types but the data being worked with.

For a fun read on its history and what went wrong see:
http://www.joelonsoftware.com/articles/Wrong.html

### 6. Constants

```
const int MAX_PLAYER_COUNT = 32;      // normal constant
const int getMaxPlayerCount()         // function that returns a constant
{
      return 32;
}
```

- Constants should be all uppercase with underscores separating words.
- Consider the use of functions that return constant values instead of constant variables. This allows for more flexibility when determining the constant.
- Do **not** use #define to declare constant variables. They are pre-processor directives and are liable to cause confusion and obscure debugging.

## LAYOUT

### 8. C++ Header / Source Files

and


Player.h, Player.cpp

- C++ header files should have the extension .h (preferred) or .hpp
- C++ source files should have the extension .cpp
- Class header and source file prefixes should be named the same.
- Classes are declared in header files and defined in source files.

## 9. Definitions

```
class Player
{
  public:
      Player() = default;
      ~Player() = default;

      // Functions longer than this should be defined in the cpp
      // and not inline in the header file!
      int getHealth() const { return health; }

      // This is the correct way to declare functions.
      int getHealth() const;

  protected:

  private:
      int hp = 100;
      int lives = 5;
};
```

- The header file should declare an interface.
- The source file contains its implementation.

## 10. Include Statements

```
// System libs
#include <iostream>
#include <vector>

// Game Engine usage
#include <Engine/Game.h>
#include <Engine/Sprite.h>

// Our codebase
#include "fsm.h"
#include "player.h"
```

- Include statements should always be located at the top of the file only.
- They should also be sorted and grouped. Sort by their position in the system, with low level and system libs included first.
- Leave an empty line between each group to help readability and understanding of the modules needed.

## 11. Line Breaks / Lengths

```
// this line is too long 100+ width
for (int player_index=0; player_index < NO_OF_PLAYERS_IN_GAME;
++player_index)

// this is within 80 and is much easier to scan when reading
for (int idx = 0; idx < PLAYER_COUNT; ++idx)
```

- Line lengths should be kept (where reasonable) to less than 80 columns width.
- There are a number of reasons for doing so, some historical and some practical i.e. readability, multi-window setups, industry practice etc.
- Visual Studio extension to show line lengths: https://visualstudiogallery.msdn.microsoft.com/da227a0b-0e31-4a11-8f6b-3a149cf2e459/view/Reviews

## 12. Split Lines

```
totalSum = ( a + b + c +
             d + e );

function(param1,
         param2,
         param3);

setText("Long line split"
        "into two parts.");
```

- Split lines are needed when a line exceeds the 80 column width limit.
- It is impossible to give hard and fast rules where line splits should occur but common sense should prevail.
- Whilst the examples above do not always exceed 80, they give a good indication of how you might split lines of code.
- Generally speaking;
  - After a comma
  - After an operator
  - Align the new line with the beginning of the expression on the previous.

## 13. Braces / Line Indentations

```
namespace UberGame
{
	class Game
	{

	public:
		Game() = default;
		~Game() = default;

	};
}

void func123()
{
	Player players[2];
	for (int i = 0; i < 2; ++i)
	{
		for (int j = 0; j < 5; ++j)
		{

		}
	}
}
```

- Indentation of code should be a single tab. There are many arguments for the use of whitespace over tabs and vice-versa, but tabs allow the user the ability to define their spacing to their own liking and helps maintain layout across IDEs.
- An extra level of Indentation should always be used when operating in a new set of {} and the opening and closing braces should be on their own lines.

## 14. Function Spacing

```
void function1()
{
	return;
}

void function2()
{
	return;
}
```

- Use a blank line between function declarations to make functions easier to spot within source files.

## 15. Block Layout

```
while (hp > 0)
{
  int r = rollDie();
  hp = doAttack(r);
}
```

```
while (hp > 0) {
  int r = rollDie();
  hp = doAttack(r);
}
```

```
while (hp > 0)
  {
      int r = rollDie();
      hp = doAttack(r);
  }
```

- Left shows a **correctly** formatted code block.
- Middle is incorrect as they opening brace is on the first line.
- Right is incorrect as the two braces are additionally indented.

## 16. Class Layout

```
class Player
{

public:
      // ...

protected:
      // ...

private:
      // ...

};
```

- Classes should always have public, before protected, before private.

## 17. For Statement Layout

```
for (int idx = 0; idx != PLAYER_COUNT; ++idx)
{
      // Statements
}


for (int idx;
     idx != PLAYER_COUNT;
     ++idx)
{
      // Statements
}
```

- The for loop follows the principle of; *initialisation*; *condition*; *update.*
- When the loop declaration is long, it can be split across several lines.

## 18. While Statement Layout

```
while (health > 0)
{
      int roll = rollDie();
      health = doAttack(roll);
}
```

- Condition should sit on its own line.
- The opening and closing braces are on new lines, but not indented.
- The body of the while loop is indented.

## 19. IF / Else Layout

```
if (health == 0)
{
	// do something
}
else
{
	if (type == GOBLIN)
	{
		// steal something
	}
	else if (type == OGRE)
	{
		// bash something
	}
	else if (type == GIBBON)
	{
		// swing and steal player's hat
	}
	else
	{
		// not supported
	}
}
```

- Quite simply, the if statement condition sits on its own line.
- Opening braces sit on their own lines and are not indented.
- The statements reside within the braces and are indented.
- The closing brace sits on its own line and is not indented.

## 20. Switch Statement Layout

```
switch (type)
{
	case GOBLIN:
	{
		// steal something
		break;
	}

	case OGRE:
	{
		// bash something
		break;
	}

	case GIBBON:
	{
		// swing and steal player's hat
		break;
	}

	default:
	{
		// not supported
		break;
	}

}
```

- The switch statement resides on its own line.
- Opening braces sit on their own lines and are not indented.
- The cases sit within the braces and are indented.
- The body of each of case is contained within a new set of braces.

## 21. Grouping

```
Player human = Player();
Enemy monster = Enemy();
std::cout << "Health: " <<
human.getPlayerHealth() <<
std::endl;
std::cout << "You enter a dark
room.\n
Around you, the stench of decaying
flesh suffocates you. \nSurely this
is the wrong turn!" << std::endl;
_sleep(2000);
std::cout << "Suddenly a gruesome
creature appears in front of you."
<< std::endl;
while(human.isAlive())
{
	monster.attack();
	human.takeDamage();
	_sleep(1000);
}
std::cout << std::endl;
std::cout << "You are dead. \nYou
scored a massive " << hu-
man.getScore() << " Points \nBetter
luck next time!" << std::endl;
```

```
Player human = Player();
Enemy monster = Enemy();


std::cout << "Health: " <<
human.getPlayerHealth() <<
std::endl;

std::cout << "You enter a dark
room.\n Around you, the stench of
decaying flesh suffocates you.
\nSurely this is the wrong turn!" <<
std::endl;

_sleep(2000);
std::cout << "Suddenly a gruesome
creature appears in front of you."
<< std::endl;

while(human.isAlive())
{
	monster.attack();
	human.takeDamage();
	_sleep(1000);
}

std::cout << std::endl;
std::cout << "You are dead. \nYou
scored a massive " << hu-
man.getScore() << " Points \nBetter
luck next time!" << std::endl;
```

- Group code together and use new lines to place them in logical units.
- Use a single line break between the blocks
- This will enhance the readability of the code

## COMMENTING

### 22. Confusing Code

```
/**
*  This function is designed to calculate the length of a rht.
*  c^2 = a^2 + b^2
*/
float pythagoras(float a, float b)
{
	return (a*a) + (b*b);
}
```

- Comments should help to explain the purpose of your code, they should be kept as short and simple as possible. **General guide**; *an excess of comments shows a badly organised and confusing program design.*
- Code should be rewritten or simplified rather than littered with comments.
- Comments themselves are not processed by the compiler nor can you presume they will be read by programmers, so should not be relied on.
- It is more acceptable to comment the functions operation before the body of the function begins and when doing so should utilise Javadoc formatting. See doxygen.

### 23. Inline Comments

```
// bind it
sf::TcpListener listener;
if (listener.listen(SERVER_PORT) != sf::Socket::Done)
{
	// throw error
	return -1; // throw error
}

class Player
{

private:
	int health = 100; /**< a player's health. */

};
```

- Comments should be included relative to their position in the codebase.
- Do not add comments to the right of the code as it harms readability.
- The exception to this are member functions/variables, which can be inline

## 24. Doxygen

```
/**
*  A test class. A more elaborate class description.
*/
class Javadoc_Test
{
public:
        /**
        * An enum.
        * More detailed enum description.
        */
        enum TEnum
        {
                TVal1, /**< enum value TVal1. */
                TVal2, /**< enum value TVal2. */
        }

        /**
        * A constructor.
        * A more elaborate description of the constructor.
        */
        Javadoc_Test();

        /**
        * A destructor.
        * A more elaborate description of the destructor.
        */
        ~Javadoc_Test();

        /**
        * a normal member taking two arguments and returning an integer value.
        * @param a an integer argument.
        * @param s a constant character pointer.
        * @return The test results
        */
        int testMe(int a, const char *s);

        /**
        * a public variable.
        * Details.
        */
        int publicVar;

        /**
        * a function variable.
        * Details.
        */
        int(*handler)(int a, int b);
};
```

- If you wish, you may follow the Doxygen commenting style in headers.
- Doxygen will then be able to create documentation automatically which describes your program's API.
- Use of Doxygen commenting is **not** mandatory.
- https://www.stack.nl/~dimitri/doxygen/manual/starting.html

## MISC

### 25. Pointers, Reference Symbols & Null Pointers

```
std::string  str      = "string";
std::string& str_ref  = str;
std::string* str_ptr  = &str;
void*        void_ptr = nullptr;
```

- Simply, keep them to the left side i.e. next to the type.
- Always initialise pointers to nullptr.
- Do **not** use NULL, null or 0, these are legacy parts of the standard

### 26. Namespaces

```
#include <iostream>
~~using namespace std;~~

void fnc()
{
	int n = 0;
	std::cout << "Enter integer";
	std::cin >> n;


	std::cout << "Player's starting HP: " << n << std::endl;

}
```

- Do **not** use namespaces on a global level.
- They pollute, cause conflicts and obscure interfaces.
- Even worse, if added to a header, all other files that include it automatically gain access to the same namespace, leading to confusion when working on unrelated areas of code.
- Simply, just precede the calls with the relevant namespace.