C++ Coding Standards for 1st yr Digital Art & Development ITCarlow 2023

Table of Contents

Names

Names are the heart of programming. In the past people believed knowing someone's true name gave them magical power over that person. If you can think up the true name for something, you give yourself and the people coming after power over the code.

A name is the result of a long deep thought process about the ecology it lives in. Only a programmer who understands the system as a whole can create a name that "fits" with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected.

Language

All names should be written in English.

```
sf::RenderWindow m_window; // not m_fuinneog
```

Include Units in Names where appropriate

If a variable represents time, weight, or some other unit then include the unit in the name so developers can more easily spot problems. For example:

```
        int m_travelTimeHrs;
        int m_rangeKms;
        int m_speedKmh;
```

Class Names (PascalCase)

Use upper case letters as word separators, lower case for the rest of a word

First character in a name is uppercase

No underscores ('_')

```
class Game
class EnemyBullet
class CargoShip
```

Member Methods/Functions (camelCase)

Use upper case letters as word separators, lower case for the rest of a word

First character in a name is lowercase

No underscores ('_')

```
       void   processEvents();
       void   update(sf::Time);
       void   render();
```

Member attributes / variables (camelCase)

Prefix with m_

Use upper case letters as word separators, lower case for the rest of a word

First character in a name is lowercase

No underscores ('_') between words

```
       sf::Vector2f m_heading;
       float m_turnRate;
       float m_frictionRate;
       int m_size;
```

Static attributes/variables (camelCase)

Prefix with s_

Use upper case letters as word separators, lower case for the rest of a word

First character in a name is lowercase

No underscores ('_') between words

```
       static float s_screenHeight;
       static GameState s_currentState;
       static bool s_testMode;
```

Constants (SCREAMING_SNAKE_CASE)

Constants should be All Caps with '_' between words.

```
const float  PI_F = 3.14159265358979f;
const int  MAX_ENEMIES = 100;
const bool  IS_PETE_COOL = true;
```

Enum Names (PascalCase)

Use upper case letters as word separators, lower case for the rest of a word

First character in a name is uppercase

No underscores ('_')

Enum Labels (PascalCase)

Use upper case letters as word separators, lower case for the rest of a word

First character in a name is uppercase

No underscores ('_')

```
enum class    Direction
{
       North,
       South,
       East,
       West,
       NorthEast,
       NorthWest,
       SouthEast,
       SouthWest,

};
```

File Names (PascalCase)

Use upper case letters as word separators, lower case for the rest of a word

First character in a name is uppercase

No underscores ('_')

File name should match the class

A new file per class unless it's a supporting struct enum or class

MyClass.cpp & MyClass.h

C++ File Extensions

Use the following extensions

- **.h** extension for header files
- **.cpp** for source files.
- **.inl** for inline files.

Collections(camelCase[s])

```
Use camelCase for variables holding multiple values/objects.

Use the plural of the name unless defines a collection.

       Sweet bag[];
       Bullet sideKickBullets[];
```

Temporary variables for debugging

It is easy to forget to remove a variable added while debugging and end up checking it into the product. For this reason, try to use "temp" in any variable name or strings added for debugging. A search for temp during the release process will be used to highlight and erroneously checked temporary code. Similarly try not to use "temp" in non-debugging code.

Pointer declaration

Pointer declarations should read correctly from right to left. These rules are applicable to both variable declarations and function arguments.

```
// Correct
// a is a pointer to an int
       int *a;
// b is a constant pointer to an int
       int *const b;
// c is a constant pointer to a int which is constant
const int *const c;
```

```
// Incorrect
int* a;
int const* b;
const int const* c;
```

Pointer Initialisation

Use nullptr instead of NULL or 0;

```
// correct
int *p1 = nullptr;

// incorrect
int *p2 = NULL;
int *p3 = 0;
```

Reference declaration

As far as possible the rules for pointer declaration also apply to reference declaration.

```
// Correct
// a is a reference to x
int &a = x;

// a is a constant reference to x
const int &c = x;

// Incorrect
int& a = x;
const int& b = x;
```

Formatting

Line Length

Line of code should not exceed 80 -100 characters when breaking a line do logically and keep the indentation logical.

```
// Correct long expression
if (   (longNamedVariable > maxLongNamedVariable) &&
       (shortNamedVariable > maxShortNamedVariable))

// Incorrect long expression
if ((longNamedVariable > maxLongNamedVariable) && (shortNamedVariable >
maxShortNamedVariable))
```

Declaring variables

Always declare each variable on a separate line.

```
       int m_height;
       int m_width;
```

Argument lists

When specifying the arguments to a function or method each argument should be on its own line. This creates short lines and makes it easier to scan the list.

```
       // Correct
       void myFunction(char t_initial,
                       int t_id);

       // Incorrect
       void myFunction((char t_initial, int t_id);
```

Argument Names (camelCase)

Prefix with t_

Use upper case letters as word separators, lower case for the rest of a word

First character in a name is lowercase

No underscores ('_') between words

Use full argument name in declaration not just data types.

```
// wrong
int myFunction(int);

// correct
int myFunction(int t_input);
```

You can use the type name for generic functions applying to that type/class

```
void GamePlay::outOfBounds(Asteroid t_asteroid);
```

Spacing and tabs

For code block indentation always use tabs and not spaces. It is preferable to set tabs to be equal to 4 spaces in your file editor.

Use spaces between binary operators and their operands.

```
	// Correct
	int i = 0;

	// Incorrect
	int i=0;
```

Don’t use spaces between unary, postfix and prefix operators and their operands.

```
	// Correct
	i++;
	--i;
	-i * j;

	// Incorrect
	i ++;
	-- i;
	- i * j;
```

Don’t use spaces before semicolons.

```
	// Correct
	i = 0;

	// Incorrect
	i = 0 ;
```

All the above also apply to for, if, else, while, do, switch, case, try, catch etc statements.

```
	// Correct
	for (x = 0; x < xMax; x++)
	// Incorrect
	for (x=0; x<xMax; x++)
```

Indentation and braces

Braces should be placed on separate lines and code indented between braces.

```
	// Correct
	void myFunction()
	{
		//… code …
	}

	// Incorrect
	void myFunction() {
		//… code …
	}
```

Braces should always be used to scope code blocks, even single lines. This applies equally to for, if, else, while, do, switch, case, try, catch etc. statements as it helps reduce errors when refactoring code.

```
	// Correct
	for (x = 0; x < xMax; x++)
```

```
        {
                doOneThing();
        }


        // Incorrect
        for (x = 0; x < xMax; x++)
                doOneThing();

        // correct nested loop
        for (x = 0; x < xMax; x++)
        {
                for (y = 0; y < yMax; y++)
                {
                        //… code …
                }
        }


        // Incorrect nested loop
        for (x = 0; x < xMax; x++)
                for (y = 0; y < yMax; y++)
                {
                        //… code …
                }

        // Correct if … else … statement
        if (test)
        {
                doOneThing();
        }
        else
        {
                doAnotherThing();
        }

        // Incorrect if … else … statement
        if (test)
                doOneThing();
        else
                doAnotherThing();
```

Parentheses

Use parentheses to increase the clarity of expressions, even when operator precedence makes them redundant.

```
        // Correct
        r = ((a * b) + (c / d));

        // Incorrect
        r = a * b + c / d;
```

There is no need to parenthesise return values.

```
        // Correct
        return result;

        // Incorrect
        return (result);
```

Comments

Introduction

Standard Header (every file)

Every file must include the standard header with all the relevant fields filled in.

```
/// <summary>
/// @author Peter Lowe
/// @version 1.0
///
/// </summary>
```

Any file without an @author tag will not be considered student work and ignored in all grading.

Main project file Header (one per project)

Every project must have a main source file (game.cpp) and it must include a project type header including the following tag and details.

```
/// <summary>
/// @mainpage Asteroids - Joint Project one a side scroller.
/// @Author Peter Lowe
/// @Version 1.0
/// @brief Joint Project one a side scroller.
///
/// A game where you control a space ship like R-type
/// moving up and down while enemies fly from right to left
/// You can shoot with different weapons.
///
/// date & time of each session and time taken
/// Total time taken on this project e.g.
///
/// 4/5/16 14:22     40min
/// 4/5/16 16:30     30min
/// 5/5/16 9:00      140min (2hr 20min)
/// 6/5/16 14:00     10min
///
/// Total Time Taken 3hr 40 min
/// </summary>
```

C Style Comments

Don't' use C-Style comments /* c style */.

Instead use c++ style // always follow with a space.

```
/* Bad old c style */
// Good new c++ style
//Bad no leading space
/// Doxygen aware comment
```

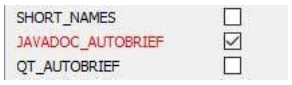
Doxygen commenting

Doxygen enables the automatic generation of documentation from source code comments, through use of certain flags.

To make a comment part of the doxygen library use three forward slashes.

```
/// this line will be examined by Doxygen
// whereas this line will not.
```

Ideally each class/function/variable/etc. should have a brief description and a detailed description. The brief description should not extend past one line in length, and is terminated by a full stop. The detailed description is not always required but it should be considered for all but the most trivial of objects. (if not setting JAVADOC_AUTOBRIEF to true you will to leave a blank line between the brief description and the full description)


SHORT_NAMES ☐
JAVADOC_AUTOBRIEF ☑
QT_AUTOBRIEF ☐

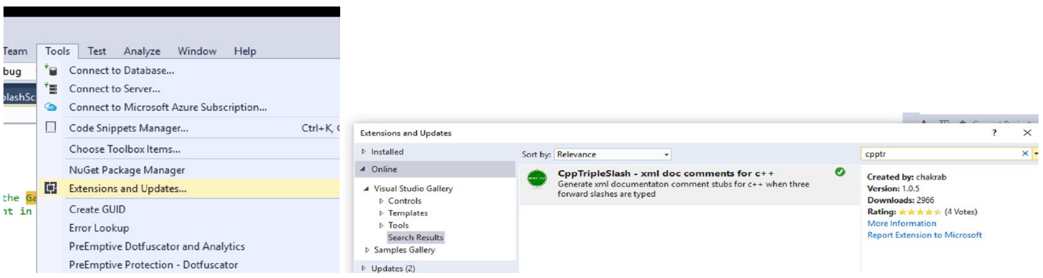
```
/// This is a brief description.
///
/// This description is a much more verbose and
/// potentially more detailed description of the
/// following item. There is no point however in
/// simply saying the same thing as in the brief
/// description.
class Game
{
```

CppTripleSlash

To make it easier to document your functions you can install the free extension for Visual Studio CppTripleSlash this will fill most of the structure for commenting a method automatically including parameter variables.

To install CppTripleSlash

- Select “Extensions and updates” from the Tools menu.
- Select online on the left.
- Search for “cpptr”
- Select “Download” and restart vs when asked.



CppTripleSlash - xml doc comments for c++
Generate xml documentaton comment stubs for c++ when three forward slashes are typed
Created by: chakrab
Version: 1.0.5
Downloads: 2966
Rating: ★★★★☆ (4 Votes)

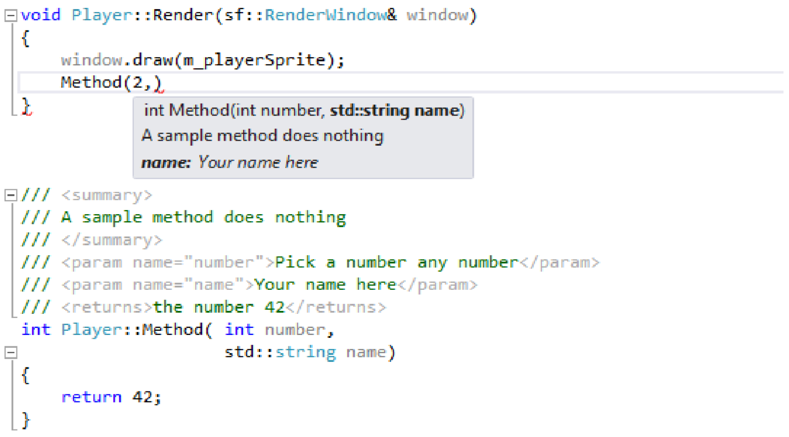
An example of the default comment created by typing 3 slashes before the method

```
/// <summary>
///
/// </summary>
/// <param name="number"></param>
/// <param name="name"></param>
/// <returns></returns>
```

```
int Player::Method(int t_number,
                   std::string t_name)
{

}
```

After you have summarised your method implementation, intellisense will prompt you with the details of the method while you type, displaying each parameter description in turn as you enter them.

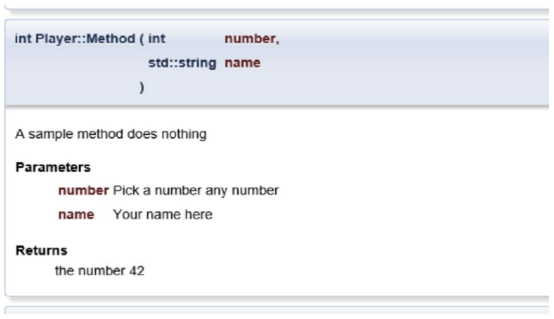


```
void Player::Render(sf::RenderWindow& window)
{
    window.draw(m_playerSprite);
    Method(2,)
}
```

int Method(int number, **std::string name**)
A sample method does nothing
name: *Your name here*

```
/// <summary>
/// A sample method does nothing
/// </summary>
/// <param name="number">Pick a number any number</param>
/// <param name="name">Your name here</param>
/// <returns>the number 42</returns>
int Player::Method( int number,
                    std::string name)
{
    return 42;
}
```

Doxygen will generate a block for the method like below.



```
int Player::Method ( int          number,
                     std::string  name
                   )
```

A sample method does nothing

Parameters

number	Pick a number any number
name	Your name here

Returns

the number 42

No Doxygen Warnings for missing documentation

When running Doxygen you should get no warnings, so every variable will need a comment above it's declaration and every method and class a summary;

```
// correct amount of commenting

protected:
        /// texture of landscape used as background
        sf::Texture m_backgroundTexture;
        /// sprite used to render background texture
        sf::Sprite m_backgroundSprite;
        /// instance of player
        Player m_player;
        /// boolean used to pass jump key from processevents to update loop
        bool m_jumpKeyPressed;

…

/// <summary>
/// @brief check for event prior to update.
```

```
///
/// check for up arrow key press before main loop calls update
/// </summary>
/// <param name="event">sf event from os</param>
void GamePlay::processInput(sf::Event t_event)
{
	if (t_event.type == sf::Event::KeyPressed)
	{
		if (t_event.key.code == sf::Keyboard::Up)
		{
			m_jumpKeyPressed = true;
		}
	}
}
```

Commenting out blocks of code

Use a #ifdef with a meaningful never defined name to comment out blocks of code and remove obsolete code when obsolete.

```
#ifdef NOGRAVITY
/// <summary>
/// @brief apply gravity.
///
/// move the player position down one so long as he is above road
/// </summary>
void Player::Fall()
{
	if (m_playerSprite.getPosition().y < m_roadHeight)
	{
		m_playerSprite.setPosition(m_playerSprite.getPosition().x,
m_playerSprite.getPosition().y + 1);
	}
}
#endif // NOGRAVITY
```

Logical coding conventions

Variables

Avoid reusing variables for unrelated tasks, and give them meaningful names wherever possible.

Declare variables when and where they are used in a method not all at the start, their scope should be the smallest required.

```
// Correct
int subTotal = 0;
for (x = 0; x < xMax; x++)
{
	subTotal += myArray[x];
	doSomething(subTotal);
}

int sum;
sum = xMax + yMax;
doSomethingElse(sum);

// Incorrect
int sum = 0;
```

```
for (x = 0; x < xMax; x++)
{
	sum += myArray[x];
	doSomething(sum);
}

sum = xMax + yMax;
doSomethingElse(sum);
```

Use const variables to break up complex expressions by storing intermediary values.

```
// Correct
const int result = a + b - c;
return result;

// Incorrect
return a + b - c;
```

if then else vs switch

When testing for one of several outcomes if deciding on the basis of a single variable then use a switch statement which can sometimes result in quicker code and always better code.

```
// incorrect
if (number == 6)
{
	message = "Tom Mix";
}
else if (number == 7)
{
	message = "Lucky";
}
else if (number == 8
{
	message = "Garden Gate";
}


// correct
switch (number)
{
case 6:
	message = "Tom Mix";
	break;
case 7:
	message = "Lucky";
	break;
case 8:
	message = "Garden Gate";
	break;
default:
	break;
}
```

Casting

Cast operations should be avoided if possible, but don't use C-style deprecated casts.

```
// Incorrect

float floatPosition = 1.50f;

int intPosition = (int) floatPosition;
```

```
// Correct

int intPosition = static_cast<int>(floatPosition);
```

Public, Private and Protected

There are few circumstances when members should be declared private. As a general rule non-public data and functions should be declared protected.

Public should appear first in the class definition then protected and finally private.

```
// Incorrect

Class MyClass
{
Private:
   int m_someValue;
Public:
   MyClass(int t_value);
};


// Correct

Class MyClass
{
Public:
   MyClass(int t_alue);
Protected:
   int m_someValue;
};
```

Member initialisers

Use member initialisers to set initial object state.

```
MyClass
{
public:
       MyClass(int t_value);
private:
       int m_someValue;
};

// Incorrect
MyClass::MyClass(int t_value)
{
       m_someValue = t_value;
}

// Correct
MyClass::MyClass(int t_value) : m_someValue(t_value)
{
}
```

Yoda Notation

When testing variables against constants within an if statement place the constant on the right hand side , if you forget the second equals sign the force will cause a compiler error.

```
// incorrect
if ( myVariable == CONST_VALUE )
{


// correct
if ( CONST_VALUE == myVariable )
{
```

Don’t answer your own question

Don’t check if a boolean is equal to true or alternately not equal to false;

```
// incorrect
if (truth == true)
{
        // so true
}
if (falsehood != true || falsehood == false)
{
        // so not true
}

// correct
if (truth)
{
        // so true
}
if (!falsehood || !falsehood)
{
        // so not true
}
```

Don’t use “and” or “or” in code

Don’t use the words and, or as logical operators always use || , && for logical AND and OR.

```
// incorrect
        if (true and false)
        {
                // never going to happen
        }
        if (true or false)
        {
                // bound to happen
        }


        // correct
        if (true && false)
        {
                // never going to happen
        }
        if (true || false)
        {
                // bound to happen
        }
```

#includes

All includes should use #include <filename> for existing libraries and #include "filename" for header files from this project.

Never use *#include* if it can be avoided. Header files should only *#include* those files that are strictly necessary:

- headers of inherited classes
- headers for class members

Header File / Include Guards

Don't use #pragma once as it's not standard on all compilers.

Include files should protect against multiple inclusion through the use of macros that "guard" the files.

```
#ifndef FILENAME_H
#define FILENAME_H
// class definition goes here

#endif // !FILENAME_H
```

The macro FILENAME_H should match the filename of the header class all in uppercase. Eg. PLAYER_H for (Class Player saved in Player.h)

Using statements

Don't use using statements rather fully qualify your members.

```
        //correct
        std::string message = "Pete is cool";
        std::cout << message << std::endl;

        //incorrect
        using namespace std;
        string message = "Pete is cool";
        cout << message << endl;
```

Debug mode

In the main header file "game.cpp" use the inbuilt (vs2015) _DEBUG to setup your game up for testing mode and use a release target not to have those blocks included.

Use this for

- FPS counters

- Initial gold/lives
- Starting level
- GOD mode

```
#ifdef _DEBUG
#define TEST_FPS
// #define GOD_MODE
#define STARTRICH

#endif // _DEBUG
```

God Mode is not set at present in the tests but the FPS counter and initial gold/health is.

```
// in debug mode
       m_gold = 0;
#ifdef STARTRICH
       m_gold = 1000;
#endif // STARTRICH


// in release mode
       m_gold = 0;
#ifdef STARTRICH
       m_gold = 1000;
#endif // STARTRICH
```

const usage

Use const wherever possible, generous const usage will result in faster code.

- all *get* methods should be const
- variables that don't change should be const
- references to variables that don't change should be const
- the variables to which pointers point should be const if they don't change
- unless to are doing pointer arithmetic the pointer itself can usually be const

Type matched constants

When setting values in code match the number to the datatype of the variable.

```
// incorrect
int intValue = 0;
float floatValue = 0;

double doubleValue = 0;

// correct
int intValue = 0;
float floatValue = 0.0f;

double doubleValue = 0.0;
```