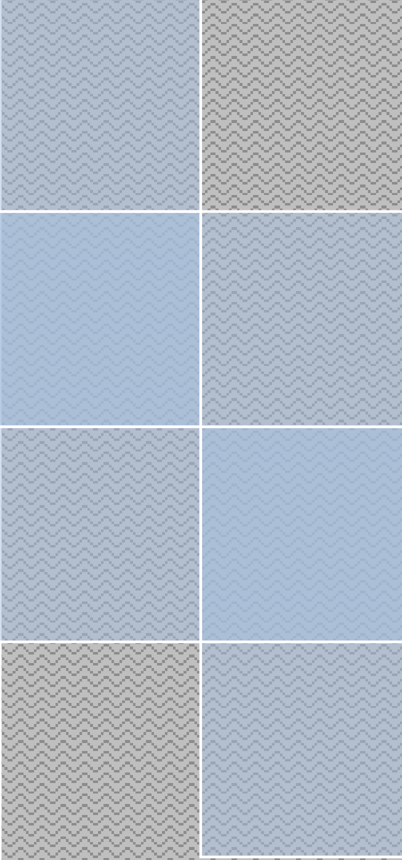

# C++ Coding Standard

Karsten Vermeulen

# *Table of Contents*

Introduction ........ 3
Good Coding Practise ........ 4
Variables ........ 9
Functions ........ 12
Classes ........ 15
Header & Source Files ........ 20
Resource Files ........ 22
References ........ 24

# Introduction

This coding standard has been written to ensure all game code written in C++ adheres to the same basic guidelines. It is imperative that all code remains well organised and structured, so that it can be easily re-used in the future. By enforcing this basic standard, all future projects can be developed with ease as the code will be a lot more clearer to read, easier to debug and possible to extend.

There are many *do's* and *don'ts* in the programming field and it is important to be aware of these issues. While most of the practises in this document won't determine a program's fate, they will help you improve your programming style, and force you to remain consistent at all times. Better code leads to better software, so take the following ideas into consideration.

# Good Coding Practise

- ***Indentation***: Although *Visual Studio* offers automatic indentation, some compilers may not, so make it a habit to indent all your code, as this will make your program more readable and clearer to others. When writing blocks of code and functions, keep the open and closing brace lined up. This is a good example :

```
while(i < 100)
{

    //do something

    if(i == 20)
    {
        //do something else
    }

}
```

- ***Braces*** : Use an open and closing brace, even if only one line of code is used, because it looks neater and allows for easily adding more code to the block later without re-typing too much.

- ***Comments*** : Commenting your code is important, as it makes things clear to other programmers what your intentions are or how a piece of code functions. However, don't comment the obvious and try to enforce self-documenting code in your programs. This allows for minimal comments and maximum clarity.

```
bool SetupScreen(int width, int height);
```

- ***Re-use tools*** : Always use the correct tool for the job. Remember, C++ comes with an extremely convenient *STL* library, so first look in there for something that might do the job, before attempting to write the code yourself. Never *re-invent the wheel*.
  Use the correct algorithm for the right calculation, or use the correct container for storing your data. Some containers prove expensive when used incorrectly and this can lead to poor game performance. When using loops, use the correct one for the task at hand.

- ***Avoid C-style code*** : C++ stems from the old C language and still has many old style features incorporated. This is because C++ is backward compatible. However it is good practise to avoid any old C-style coding (if and when possible) and stick to the modern form. This includes not using arrays and rather using vectors. Instead of structs, use classes. Use string objects instead of char arrays or char pointers.
  Having said that, it also really depends on what you are coding. Again, use your own initiative and decide what will work best for your current situation, and what will help improve performance in your program.

- ***Avoid 'goto'*** : There are some things to avoid in C++ and these involve the use of the `goto` statement. This can be handy to move to any place in your code, but it can become very confusing, break your program flow, and

generally lead to errors. Therefore it is best to keep clear of this keyword to the point of not using it at all.

- ***Avoid global variables*** : Also, try to avoid using global variables. Keep your variables as localized as possible, as this will also make things more clear as to what the variable is intended to do. For more on variables, see the *Variables* section.

- ***Heap memory*** : With great power comes great responsibility. Therefore it is vital that memory is dealt with in the correct manner. The first thing to bear in mind is that for every `new` keyword there needs to be a corresponding `delete` keyword. If you keep this small rule in mind, you will almost certainly avoid most memory leaks.
  Count all of your `new` keywords and place the same amount of `delete` keywords into your code to be safe. Another thing to remember is that when de-allocating memory on the heap, you are left with "dangling" pointers, so after every `delete` expression, make sure you re-assign the pointer to 0.

- ***Compile & build often*** : While coding away, always be sure to compile and build your game or program as frequently as possible. This helps find errors early on in the development stage, rather that compiling a very large project that will reveal a long list of endless bugs.

- ***Don’t ignore warnings*** : Also, even though warnings may not always be serious, make sure you don’t have any in your code. If you do, investigate and try to solve the issue rather than ignore it, as it could lead to bigger bugs later on.

- ***Logic errors*** : Beware of logic errors, these errors will compile perfectly, but at runtime, the game logic will behave in an unusual way. For instance consider the following block of code :

  ```
  int x = 10;

  if(x = 20)
  {
      //do something
  }
  ```

  This will compile OK, but the program result will behave unexpectedly. This is a common programming error where programmers forget the second **‘=’** sign. A better way to avoid these types of errors is to get into the habit of putting the constant value *before* the variable :

  ```
  if(20 == x) //do something
  ```

  That way, if the equal sign is forgotten, you will get a compiler error straight away.

- ***Use standard English*** : When naming your variables, functions, and classes, be sure to use standard English names, as this is the preferred language and it is

international. This will ensure that a vast majority of programmers will be able to read and understand your code.

- ***Code line limit*** : When coding in your IDE of choice, you may become accustomed to your widescreen monitor and write long lines of code across the screen, without realising that other programmers who will read and edit your code might not have a widescreen monitor at all. Therefore it is imperative to stick to a line limit, preferably no more than 100 lines across.

# Variables

It is essential that variables are correctly labelled to ensure less confusion when using them. Badly named variables may not necessarily lead to bugs, but can lead to longer development time, as programmers will end up searching through code in frustration, not knowing exactly what each variable does.

- ***Use correct labelling*** : A good practise is to clearly label each variable with a descriptive name that is not too long, and is in lowercase.

  ```
  int score;
  string name;
  float time;
  ```

  There is an exception to this rule, and that is when naming variables that will be used for a very short time. In these cases, letters like x or i are sufficient.

  ```
  for(int i = 0; i < 10; i++)
  ```

- ***Camel casing*** : Sometimes names might be longer or contain multiple words in them, so in that case, it is advised to use *Camel Case* in naming your variables.

  ```
  int totalScore;
  string playerName;
  float totalTimePassed;
  ```

- ***Maximum 20 characters*** : Variable names shouldn't be too long either, as this could also lead to confusion, so keep the names to a maximum of 20 characters.

- ***Capitol letters*** : Variables may also be declared as constant sometimes, so a good way to do that would be to label the variable name entirely in capital letters :

  ```
  const float GRAVITY = 12.5f;
  ```

  Another use for capital letters is when dealing with enumerated types :

  ```
  enum { ONE, TWO, THREE };
  ```

- ***Immediate initialisation*** : All variables should be initialized as soon as they are declared. This ensures that they don't contain cryptic values that could lead to unknown runtime errors later on.
  This also makes it clear to other programmers what the variable is doing or going to do.

  ```
  string enemyName = "The Big Boss";
  int zombieHealth = 100;
  int totalAttack = 0;
  ```

- ***Localisation*** : Keep variables as local as possible. For instance, if a variable is being used in a class, consider whether to make it a data member variable or a local variable of one of that class' member functions.

Avoid using global variables as much as you can. Usually, if you have enough data to initialize the variable, then declare it at that point.

- ***Use correct data types*** : When deciding what type of variable you want to declare, consider what it will be used for. Try use only what is needed, so if you are storing a player health value, use `unsigned` variables, if you are working with time and gravity, use `float` or `double` . To preserve memory consider using `short` rather than `long` .
  For the sake of portability, when using an API like *OpenGL* or *DirectX*, consider using the data types that come with those libraries. For instance in the case of *OpenGL*, instead of `float` rather use GLfloat. This ensures that compilers on different platforms will compile the code correctly, and the need for recoding is lessened.

- ***Use 'typedef'*** : If a variable type name is too long, it might be a good idea to substitute your own name for it. This leads to more readable code.

```
typedef unsigned long long Ulong;
```

# Functions

It is important to break your code up into as many different functions as possible. Your *main()* function should never be longer than a few lines of code. Each individual task in the program or game should be handled by a function, and even some functions may want to call others to perform certain actions. If you find you have a lot of repetitive code in your program, consider using functions instead.

- ***Use cohesion*** : Never give functions multiple tasks or too much of a task to handle. Make sure they instead perform one specific task. This is known as *cohesion*, and ensures that each function has a certain responsibility.

- ***Declare all functions*** : It is always a good idea to prototype your function. Even if you have functions that are defined before they are called in the program, sometimes this might not be the case, so enforce the habit of declaring all your functions, ensuring they are labelled the same way as they appear in the function definition.

```
//function prototype
void SetPlayerScore(int score);

void SetPlayerScore(int score)
{
    //do something
}
```

Make sure to include the parameter name in the function prototype. Even though you don't have to, it makes things a little clearer.
*Note : The function parameter names and any variables inside the function should follow the same guidelines as stated in the Variables section.*

- ***Function naming*** : Use descriptive names for the function prototypes and headers, as this can make it clear as to what the function is designed to do. If the name consists of multiple words, ensure that you capitalize the first letter of each word.

```
void TalkToPlayer(string message)
```

- ***Use correct parameter types*** : When passing variables to functions, if they are large data types like string objects or pre-defined classes, rather pass them by reference or as pointers to the objects than by value. This prevents unnecessary copying of large objects that could become expensive on the program.

- ***Use 'const'*** : Generally, make your reference parameters `const`, due to the fact that references can change the objects inside the function. This ensures that the function is read-only.

- ***Avoid nesting*** : nesting functions too deeply can lead to confusion and possible errors.

- ***Predicate functions*** **:** For predicate functions, name them all using **'is'** or **'has'** keywords, like *IsFullscreen().* This is essentially like a getter function, but it's clearer as to what it will return.

- ***Complimentary functions*** **:** Create functions that complement each other like *Initialize()* and *Shutdown()* or *Load()* and *Unload().* What you do in the one you oppose in the other. This presents a good structure in the code and avoids confusion.

# Classes

Creating your own data types to break your game down into smaller bite-sized chunks is vital, and the key to OOP. A few guidelines for using classes are listed in this section.

- ***Naming conventions*** : All class names should begin with a capitol letter. All data members in the class should begin with an **'m_'** and if the data member is a static variable, it should begin with an **'s_'**. Member functions should follow the same rules as stated in the *Functions* section of this document.

  ```
  class Player
  {

  public :
      Player();
      int GetScore();

  private :
      int m_score;
      static int s_totalScore;

  };
  ```

- ***Use 'public' & 'private'*** : Make sure your class has access modifiers at all times, to keep the code clear. Good practise is to make all your data member variables `private` and use `public` accessor member functions to gain indirect access to them from the main code. This

allows you to create *getter* or *setter* member functions that can make the class read- or write-only.
Break all class data members and member functions into *public* and *private* categories to create a neat arrangement.
To make your code even more readable, consider breaking your class members into multiple public or private sections. For instance, your getters, setters and normal member functions could be structured like below :

```
public :

    int GetSomething();
    int GetAnother();

public :

    void SetSomething(int data);

public :

    void Update();
    void Draw();
```

- ***Encapsulation*** : Keep *encapsulation* in mind. Data hiding is important, so be sure to conceal as much of your class as possible. Don't write accessor member functions if they are not needed.

- ***Inlining*** : You can inline your member functions to help improve performance; however don't inline very large

functions as this will bloat your final *EXE* file and program memory.

- ***Accessing heap memory*** : If your class is going to access memory on the heap, it is important to write your own destructor. It is equally important to then also write your own copy constructor and overload the assignment operator to make sure *deep copies* will be made.

- ***Constructor guidelines*** : Make sure your constructor initializes all data members accordingly. When giving your data members values, make sure you initialize them in the same order as they appear in the class declaration. The constructor is also the place to assign heap memory to any data member pointers.
  If you are creating your own constructors and you create ones that take parameters, make sure you create default ones too, so the compiler doesn't complain if you ever create the object the default way.

- ***Destructor guidelines*** : Use your destructor to do any necessary cleanup like de-allocating heap memory.

- ***Virtual functions*** : If you have virtual functions in your class, make sure you put the `virtual` keyword in both the base class and derived class member function declaration to make the code clearer. Also remember to then create a `virtual` destructor too. Note : *Virtual functions are an expensive feature of C++, so use them wisely.*

- ***Singleton classes*** : If you are intending to make your class a Singleton, make sure your constructor, copy constructor and assignment operator is declared under the `private` keyword in the class declaration.

- ***Use 'const'*** : Member functions that only return values and do not alter any data members should be declared as constant like so :

  ```
  bool GetAnswer() const;
  ```

- ***'h' & 'cpp' files*** : All classes should have their own header and cpp file. The class declaration is to be placed inside the header file and all member function definitions inside the corresponding cpp file.
  *See the section Header, Source & Resource Files for more information on this topic.*

- ***Getters & Setters*** : These functions should be named appropriately. A getter function name should start with the keyword *Get*, and similarly, a setter function should have the keyword *Set*.
  Some data members need to be accessed and assigned regularly, so instead of using separate getter and setter functions, make one member function only, which returns a reference to the data member. That way, calling only one function will allow the client code to assign and access data to/from the variable at the same time.

# Header & Source files

Part of being organized in your code is to break everything up accordingly. Using your own built classes and functions is one way, but another important method is to keep your actual code organized in a good manner.

- ***'h' & 'cpp' files*** : Use header and source files for your code. All headers should contain the declarations of everything that is to be used in the accompanying source file. Therefore all class declarations, function prototypes, preprocessor commands, enumerated types, etc should be listed in the header file. The accompanying source file should contain all the class member and normal function definitions.

- ***Template headers*** : When using template classes, there is an exception to the above mentioned rule however. With templates, both the class declaration and member function definitions should be in the same file.

- ***Naming conventions*** : To keep things clear, the header and source files of classes should have the same name as the class itself.

- ***Include guards*** : The header file should contain include guards. This should be the same name as the header file, but all in capital letters, with an **'_H'** attached at the end, like so :

```
#ifndef ENEMY_H
#define ENEMY_H
```

- ***Self-containment*** : Ensure that each header is self-contained. This means that each .h and corresponding .cpp file should have included everything they need to compile on their own. When including other header files inside your own header, keep it localized. Don't include files in your header that you don't need for that particular class. The same goes for the source file. Only include header files for things you need and use in that source file alone. It is a waste of performance to include files that are not needed.

- ***Include order*** : When including other header files, use a specific order to keep things organized. First include all standard C++ headers and libraries, then all 3rd party libraries like API headers, then include your own predefined headers, like so :

```
#include <iostream>
#include <SDL.h>
#include "Player.h"
```

Further, include the files in alphabetic order for brevity.

# Resource files

Resource files are what the game engine feeds on to run properly. All resources like images, sounds, fonts, strings, etc are stored in these files and so it is important to create and use them properly.

- ***Naming conventions*** : Name the resource files properly to avoid confusion. If you are using raw resource files like *PNG, WAV* or *TTF* files, then the filename itself should be descriptive enough to understand what it contains. A simple *Image.bmp* will not suffice, so something like *Background_Room_1.bmp* is more credible.
  Alternatively if you are using resource files that contain multiple resources, then it is imperative that both the filename and extension are marked clearly. Files like *Resource.res* say nothing, but *Backgrounds.img* might be a little more descriptive as to what is inside.
  Be consistent in naming your files to ensure an organized library of files.

- ***Tidy file system*** : A good directory structure is also essential to keep your game folder clean and tidy. Place all image, audio, font etc files in separate folders. When the product ships, this should also be the structure of the game folder system.
  In addition, name your C++ projects clearly too and organize them appropriately in well marked folders. Fellow team members and future programmers joining

the team at a later stage will appreciate a well organized directory structure, and this will prevent important files from accidentally being deleted or getting lost.

- ***Backup*** : Additionally, always remember to backup your projects.