C++ Style Guide

Table of Contents

Goals of the Style Guide –

Coding conventions are essential for maintaining code readability, consistency, and collaboration within a development team.

This document discusses recommended practices and style for programmers using the C++ language in general integrated development environment (IDE). Guidelines are based on generally recommended software engineering techniques, industry resources, and local convention. The Guide offers preferred solutions to common C++ programming issues and illustrates through examples of C++ Code.

Naming:

Choosing Names:

Give thigs names that make their purpose or intent understandable to a new reader:

1) Consider the context in which the name will be used.

2) A name should be descriptive even if it used far from the code that makes it available for use.

3) Minimize the use of abbreviation. Do NOT abbreviate by deleting letters with a word.

4) When an abbreviation is used, prefer to capitalize it.

e.g., userID rather than userId.

5) Abbreviation is ok if it universally-knowns, such as: id, i (looping index), T (template parameter) etc…

File Names:

Filenames should be all lowercase and can unclude either underscores ‘_’ or dashes ‘-‘.

Follow the convention that your project uses.

Make your filenames very specific. For instance, use https_server_logs.h rather than logs.h…

Type Names:

Type names start with a capital letter and have a capital letter for each new word, with no underscore (e.g., MyFoo).

The names of all types – classes, structs, namespaces, type definitions and enums. Share this convention.

Function Names:

Functions should start with a capital letter and have a capital letter for each new word (e.g., MyFunction()).

Macros:

#define values and enum fields will be all capitalize with an underscore ‘_’ between each word (e.g., MY_MACRO).

Prefix & Hungarian Notation:

Both refers to adding info to the variable name using prefix.

Prefixes:

Prefix will be at the start of the variable name and be separated by underscore '_' from the rest of the name (e.g., prefix_varname).

Prefix	Meaning	Example
M	private/protected class fields	m_fieldVar
K	constant variables and constexpr	k_constantVar
S	static variables	s_staticVar
G	global variables	g_globalVar
A	Function Argument	a_funcArugment

Hungarian Notation:

Hungarian Notation will add info about the variable type.

It will act as prefix but this prefix formatted differently.

First it will be tougher with the variable name but starts at lower case and then the variable name will be capitalized.

Prefix	Meaning	Example
B	boolean	BMyBoolean
C	char	CMyChar
Str	string	StrMyString
N	int	NMyInt
U	unsigned int	UMyUnsigned
F	float	FMyFloat
D	double	DMyDouble
P	pointer	PMyPointer

For more prefixes.

Comments:

Comments are absolutely vital to keeping our code readable. The following rules describe what you should comment and where. But remember: while comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments.

When writing your comments, write for your audience: the next contributor who will need to understand your code.

Comment Style:

Use only `//` for commenting. Be consistent with it.

While other ways are acceptable, `//` are much more common.

Class & Struct Comments:

Every non-obvious class or struct should have comment that will tell what it is and how it should be used.

Example

```
// Bank account balance; holds money amount and transactions actions

class CBalance

{

        ...

}
```

Function Header Documentation:

Function header documentation purpose is to tell the reader the function's attributes and goal without read the code itself.

Documentation should be as follows:

For the function header:

```
int Add(int a, int b);
```

Documentation will be:

```
// Calculates and return the sum of a and b.
// @param a: The first integer to be added.
// @param b: The second integer to be added.
// @return: The sum of a and b.
```

Final Result:

```
// Calculates and return the sum of a and b.
// @param a: The first integer to be added.
// @param b: The second integer to be added.
// @return: The sum of a and b.
int Add(int a, int b);
```

TODO Comments:

Use TODO comments for code that is temporary and need: fix, add implementation etc..

```
// TODO: Add feature
// TODO: Fix this problem
```

Formatting:

Formatting refers to the rules and conventions that govern how code should look.

The rules must be consistent throughout the project for much easier readability and better flow.

Line Length:

Each line of text in the code base should be at most 80 characters long.

Spaces vs. Tabs:

Use only spaces, indent 4 spaces.

Curly Brackets:

All opening and closing curly brackets must be placed on their own line, aligned vertically with the corresponding code.

Brackets must NOT be placed on the same line as statements or control keywords (e.g., if, for, while, function definitions), except for single-line return statements inside braces.

*** Possible exception for controlled statements that have a one liner return value.**

```
if (condition) { // Bad – Curly brackets do not have their own new separate line.
    ...
} else {
    ...
}
while (condition) { // Bad – Curly brackets do not have their own new separate line.
    ...
}
```

```
if (condition) // Good – Curly brackets have their own new separate line.
{
    ...
}
OR
if (condition) { return; } // Good – One liner return value like this is fine.

while (condition) // Good – Curly brackets have their own new separate line.
{
    ...
}
```

Looping and Conditional Statements:

Those statements are (e.g: if, else, switch, while, do, for etc…).

For these statements:

- **The components of the statements should be separated by single space.**
- **Inside the condition or iteration specifier, put single space between each semicolon and the next token.**
- **Inside the condition or iteration specifier, do NOT put a space after the open or before the close parenthesis.**
- **ALWAYS put any condition or iteration specifier inside blocks (curly braces).**

```
if(condition) // Bad – No space after the if keyword
{
   …
}
while (condition1&&condition2) // Bad – No space between the tokens
{
    …
}
if ( condition ) // Bad – Extra spaces after and before the parenthesis
{
    …
}
if (condition) // Bad – No block (curly brackets) in the if statement body.
   foo();
```

```
if (condition) // Good – Space after the “if” keyword
{
    …
}
while (condition1 && condition2) // Good – Spaces between the tokens
{
    …
}
if (condition) // Good – No extra spaces after and before the parenthesis
{
    …
}
if (condition) // Bad – Have block (curly brackets) in the if statement body.
{
   foo();
}
```

Boolean Expressions:

When a boolean expression is longer than the max line length the expression should be split for each logical condtion.

Be consistent in how you break up the lines, recommended to end each line with the logical operator (AND, OR).

```
if (first_condition > second_condition && third_condition == forth_condition) // Bad – Too long
{
    …
}
```

```
// Good – Break each logic part of the statement
if (first_condition > second_condition &&
    third_condition ==  forth_condition)
{
    …
}
```

Whitespace Usage:

Use whitespace depend on location.

Do Not put trailing (extra) whitespace at the end of a line.

Adding trailing whitespaces can cause extra work for others editing the same file.

Below is example of how you would use whitespace -

Example

```
int var = 0;  // Two white space between logic and comment at the same line
int x = 3;  // Semicolons usually have no space before and after them
int arr[] = { 0 };  // Spaces inside initialization braces are optional
int arr[] = {0};  // Also just as fine
// Spaces around the colon in inheritance
class B : A
{
   …
}
// Spaces around the colon in range-based loops or conditions.
for(const auto& item : list)
{
   …
}
return condition ? x : y;  // Don’t forget spaces around the question mark
```

Magic Numbers:

Do NOT use magic numbers in your code. Instead make a const value (in the narrowest scope possible) to cover the magic value.

Style Checker:

Use cpplint.py to detect styles errors.

It not perfect but still a valuable tool.

Download to project here: cpplint.py

Data Types:

size_t:

Use size_t when:

- Working with STL (sizeof, size/length etc…).

- Indexing through data-structures (such as array, vector…).

Integer Types:

There are few built-in C++ integer types, if different size needed use <stdint.h> for diffrent signed/unsigned int sizes:

Bits	Signed	Unsigned	Type
8	int8_t	uint8_t	signed/unsigned char
16	int16_t	uint16_t	signed/unsigned short
32	int32_t	uint32_t	signed/unsigned int
64	int64_t	uint64_t	signed/unsigned long long

Important points:

- Do NOT use unsigned integers types unless there is a valid reason such as representing a bit pattern.

- Do NOT use unsigned integer to say a number will never be negative.

- Integer conversions can cause undefined behavior, leading to security bugs and other problems. Be careful.

Floating-Point Types:

There are few built-in C++ floating-point types, the only one that should be use are float and double.

Do NOT use types like long double, as it gives non-portable results (it’s not consistent across platforms).

constexpr vs. define:

Where possible MUST use constexpr instead of define.

constexpr provides type-safety and obey scope rules in contrast to define that is global only.

In addition, constexpr provide much more readability and make debugging easier.

Type Deduction:

Type deduction means letting the compiler automatically figures out the type.

Use type deduction only if it makes the code clearer to readers or if it makes the code safer.

Do NOT use it merely to avoid the inconvenience of writing an explicit type.

Function template argument deduction:

A function template can be invoked without explicit template arguments.

The compiler deduces those arguments from the types of the function arguments.

Example

```
template <typename T>
void foo(T obj);


foo(3); // Invokes foo<int>(3);
```

auto variable declaration:

A variable declaration can use the auto keyword instead of the type.

The compiler deduces the type from the variable's initializer.

Example

```
auto a = 42; // auto will make a an int
auto& b = a; // auto will make b an int&
auto c = b; // auto will make c an int
```

Function return type deduction:

auto can also be used in place of a function return type.

The compiler deduces the return type from the return statements in the function body.

Example

```
auto foo()
{
   return 3; // This will return type of int
}
```

Lambda return and arguments type deduction:

auto can be seen in lambda function too as a return type and as argument type.

Casting:

Do NOT use cast formats like (int)x unless the cast is to void. Instead, use casts like static_cast<float>(double_var).

You may use cast formats like T(x) only when T is a class type.

sizeof:

Prefer sizeof(varname) to sizeof(type).

Use sizeof(varname) when you take the size of a particular variable.

You may use sizeof(type) for code unrelated to any particular variable.

Pointers:

nullptr vs. NULL vs. '\0':

When working with chars use '\0' (not 0 literal) as the null character.

For pointers, use nullptr and NOT NULL, as this provides type-safety.

Function Pointers:

Avoid using function pointers. Use function pointers only if it makes the code safer or easier to read.

Ownership & Smart Pointers:

AVOID using raw allocation (new and delete).

Prefer to have single, fixed owners for dynamically allocated objects. Prefer to transfer ownership with smart pointers.

Definition:

Ownership – Technique for managing dynamically allocated memory (and other resources).

The owner of a dynamically allocated object is an object or function, that is responsible for ensuring that the object deleted when no longer needed. Ownership can sometimes be shared; in which case the last owner is typically responsible for deleting it.

Smart Pointers – Classes that act like pointers. e.g., by overloading the (., *, ->) operators.

Some smart pointers types can be used to automate ownership, to ensure these responsibilities are met.

std::unique_ptr – Type of smart pointer. It cannot be copied, but can be moved to represent ownership transfer.

When goes out of scope the object will be deleted automatically.

std::shared_ptr – Type of smart pointer. Express shared ownership of a dynamically allocated object.

It can be copied; ownership of the object is shared amoung all copies, and the object is deleted when the

last std::shared_ptr is destroyed.

The next example shows one way of owner and consumer of object:

Example

```
class FooMaker
{
 public:
        std::unique_ptr<Foo> FooFactory(); // Creates object ownership
}
Class FooTaker
{
 public:
        void FooConsumer(std::unique_ptr<Foo> ptr); // Receives  Ownership (using move)
}
```

Scoping:

Local Variables:

Place local variables in the narrowest scope possible and initialize variables in the declaration.

Another recommendation is to declare the variable as close to the first use as possible.

These will make the code easier to read to find the declaration and see what type and value the variable has.

Following code will show examples of how to use and how not to use local variables:

```
int var;
var = foo(); // Bad – Initialization not with declaration
```

```
int var = foo(); // Good – Initialization comes with declaration
```

```
int jobs = NumJobs();
// More code...
foo(jobs); // Bad – Declaration separate from use
```

```
int jobs = NumJobs();
// Good – Declaration immediately (or closely) followed by user.
foo(jobs);
```

```
std::vector<int> v;
// Bad - Prefer initialization using braces, at least empty braces v{};
v.push_back(1);
```

```
std::vector<int> v = { 1, 2 }; // Good – vector starts initialized
```

Global & Static Variables:

Avoid using global and static variables in your program.

Use global and static variables only when: no custom destructor, virtual destructors and all members/bases share this rule too.

Prefer static variables that could be written as constexpr.

Namespaces:

It recommended to place code in a namespace. Namespaces should have unique names based on the project name, and possibly it path.

Do not use using-directives (e.g., using namespace std).

Nonmember, Static Member, and Global Functions:

Prefer placing nonmember functions in namespace; use completely global functions rarely.

Do not use a class simply to group static members.

Static methods can be in class in case that the method closely related to instances of the class of the class's static data.

Following examples of wrong and correct usage of static and global functions:

```
class MathHelper // Bad – Don't use classes to group static methods
{
public:
        static int Add(int a, int b);
        static double SquareRoot(double a);
}
```

```
namespace MathHelper // Good – Use namespace to group related functions
{
        int Add(int a, int b);
        double SquareRoot(double a);
}
```

```
class Account
{
 public:
        static double GetInterestRate(); // Good – Related to the class concept
}
```

```
int Add(int a, int b); // Bad – Unnecessarily use of global scope for function
```

Functions:

Input & Output:

Where possible prefer the input arguments and output to be const references.

Functions Length:

Prefer small and focused function that has a single goal.

If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Not necessarily hard limit of 40 lines, but for most of the times 40 lines should be enough.

Function Overloading:

Use overloaded functions (including constructors) only if a reader looking at a call site can get a good idea of what is happening

without having to first figure out exactly which overload is being called.

Default Arguments:

Default Arguments are allowed on NON-virtual functions when the default is guaranteed to always have the same value.

Memory Consuming Functions:

When declaring a function that accept a pointer to memory area and a counter or size for the area, we should place them in a fixed order: address first, followed by the counter. Additionally, size_t must be used as the type for the counter.

Example

```
void readData(const char* buffer, size_t bufferSize);
```

noexcept:

The noexcept specifies whether a function will throw exception or not. If an exception thrown from function marked as noexcept,

the program will crash.

noexcept will optimize performance when used correctly.

Return Values:

The return value of non-void functions must be checking by each calling function.

Validity of parameters must be checked inside each function.

In case of irrelevant return value (such as printf/cout) a void cast should be for the return value of the function.

Classes:

Access Control:

Make classes' data members private, unless they are constant.

Declaration Order:

A class definition should start with a public section, followed by protected, then private. Omit sections that would be empty.

Inheritance:

In most cases class should only inherit an interface (abstract class) and not from others concrete classes or use composition instead.

preventing conflicts (such as the "diamond problem") and maintaining readability and optimization.

Composition is often more appropriate than inheritance.

Inheritance Type:

When using inheritance, make it public.

Final Class:

Classes that should not be inherited from should be declared as final.

Help readability and understanding the logic + enable compiler optimizations.

Exceptions:

Exceptions are for unexpected problems in the code (failed open system files, sockets etc…) and NOT for

control flow, logic branching or validation. For that case return values should be use only!

Exception Handling:

Error should be thrown by value and be catch by const reference to avoid unnecessary copying.

```
// Bad - Should throw by value not by reference

throw new std::runtime_error("Something went wrong");

…

try

{

   …

}

// Bad - Should catch by const reference not by value

catch(std::runtime_error e)

{

}
```

```
// Good – Error thrown by value
throw std::runtime_error("Something went wrong");
...
try
{
    ...
}
// Good - Catch by const reference
catch(const std::runtime_error& e)
```

Exception Specification:

Exception should be thrown and catch as specific as possible, only at the end catch general errors (such as: std::exception).

```
int Divide(int a, int b)
{
        if(b == 0)
        {
            // Bad – General exception for specific error
            throw std::excpetion();
        }
      return a / b;
}
```

```
int Divide(int a, int b)
{
        if(b == 0)
        {
            // Good – Specific error thrown
            throw std::invalid_arument("Can't divide by 0!");
        }
      return a / b;
}
```

```
try
{
        result = Divide(3, 0);  // Will perform 3 / 0
}
catch(const std::exception& e) // Bad – Catch general error
{
        ...
}
```

```
try
{
        Divide(3, 0);  // Will perform 3 / 0
}

catch(const std::invalid_argument& e) // Good – Catch specific error
{
        …
}
catch(const std::exception& e) // Now try catch general error
{
        …
}
catch(…) // Wrap it all up with this catch for unexpected error
{
        …
}
```

Header Files:

In general, every .cpp file should have an associated .h file. There are some few exceptions, such as unit tests and main() files.

Order of Includes:

Include headers in the following order:

1) **Related headers.**
2) **C system headers.**
3) **C++ standard library headers.**
4) **Other external libraries headers.**
5) **Your project's headers.**

Each section of headers should be separated by a blank line.

Example for file server.cpp:

Example

```
#include "server.h"


#include <sys/types.h>
#include <unistd.h>


#include <string>
#include <vector>


#include "flags.h"
```