[C++](https://terminalroot.com/tags" \l "cpp) and [C](https://terminalroot.com/tags#clanguage) are the only high-level programming languages that give you complete freedom and permission to work the way you see fit. It is important to point out, that this “insecurity” is a fault of the programmer and not of the programming language. The truth is that almost everything we use is written in C and C++ and by programmers who know how to use these tools. Therefore, you can see that writing good code in C++ will depend exclusively on the programmer. And writing well in C++ is not such a difficult task, just dedicate yourself.

## 01. Avoid using namespace

Several libraries tend to repeat names; the use of namespace is precisely to distinguish where you are using it. When you make this global, it generates conflicts when there are same names. Not to mention that your code is more readable when someone else reads it or even you, it’s good to explicitly know the origin of that function or class.

Avoid using using namespace std; or any other namespace.

using namespace std;

int main()

{

cout << "Hello, World!" << '\n';

}

Make it a point to use directly with scope resolution!

int main()

{

std::cout << "Hello World!" << '\n';

}

If you need to use it for specific words, such as cout, string and others, specify only them, for example:

using std::cout;

using std::string;

int main()

{

string hello = "Hello, World!";

cout << hello << '\n';

}

## 02. Stop using std::endl

std::endl was created to make line breaks portable between different operating systems. For example, on UNIX-like systems (BSD, macOS, GNU/Linux) to make a line break just use '\n', but on DOS-like systems (Windows) it is used like this: "\r\n".

So std::endl directed the flow according to the operating system. But, nowadays the current versions of the compilers already do this. Then there is no need anymore! Using std::endl adds one more useless processing: std::flush . Only use '\n', enclosed in single quotes, or "Hello\n", when used in conjunction with strings.

Instead:

std::cout << "Hello World!" << std::endl;

std::cout << std::endl;

std::cout << var << '\n';

Use like this:

std::cout << "Hello World!\n";

std::cout << '\n';

std::cout << var << '\n';

## 03. Always use curly braces to make blocks explicit

In the past, C/C++ programmers used to use conditions and loops without the use of braces { } this caused semantic problems in a lot of code. To avoid these problems and make your code more readable, use curly braces, for example:

**Instead:**

int main(){

for( int i{}; i < 4; ++i)

std::cout << i << '\n';

if ( 1 == 1 )

std::cout << "One is himself =)\n";

else

std::cout << "He is not himself :(\n";

}

**Use like this:**

int main()

{

for( int i{}; i < 4; ++i){

std::cout << i << '\n';

}

if ( 1 == 1 )

{

std::cout << "One is himself =)\n";

}else{

std::cout << "He is not himself :(\n";

}

}

## 04. Initialize member variables: ()

In addition to your code gaining performance, your code is also leaner and easier to understand.

Instead:

class Vector2 {

int x, y;

public:

Vector2(int x, int y){

this->x = x;

this->y = y;

}

};

Use like this:

class Vector2 {

public:

int x, y;

Vector2(int x, int y): x(x), y(y){ }

};

## 06. Use std::size\_t

Several times we need to use some member or variable to make comparison. When this is done with elements of a vector, the compiler will issue several warnings because their comparison is invalid. This can even result in data loss. So, get used to using the size\_t of std, for example:

Instead:

int y {42};

for(int i{}; i < v.size(); ++i){}

Use like this:

std::size\_t and {42};

for(std::size\_t i{}; i < v.size(); ++i){}

## 07. Use .hpp and .cpp for your file extensions

The use of .cc and .h deviates from most standards, especially .h which is used for C language.

Instead:

main.cc entity.h entity.C common.h lib.h

Use like this:

main.cpp entity.hpp entity.cpp common.hpp lib.hpp

## 08. Whenever possible and in accordance with semantics, make excessive use of references (&) and pointers (\*)

And especially, when also possible, the use of const

Instead:

void print(std::string str){}

std::vector<int> mylist(std::vector<int> list){

return list;

}

std::string use\_str(std::string mystr){

return mystr;

}

Use like this:

void print(const std::string &str){}

std::vector<int> mylist(const std::vector<int> & list){

return list;

}

std::string use\_str(const std::string &mystr){

return mystr;

}

## 09. Avoid accessing memory directly, use smart pointers

This avoids memory leaks and unexpected errors, there are others, however the most modern is shared\_ptr to create the pointer and make\_shared to allocate memory space or share characteristics of a pointer.

Instead:

MyClass \* mc = new MyClass();

Other \* ot = new Other;

delete mc;

delete ot;

Use like this:

std::shared\_ptr<MyClass> mc = std::make\_shared<MyClass>();

auto ot = std::make\_shared<Other>();

## 10. Don’t use arrays, create lists with std::array or std::vector

Both ensure contiguous memory layout of objects and can (and should) completely replace the use of C-style arrays for many of the reasons listed for not using simple pointers.

Instead:

int arr[10] = {1,2,3,4,5,6,7,8,9};

std::string fruits[4] = {"Apple", "Orange", "Banana", "Cherry"};

Use like this:

std::array<int, 10> arr {1,2,3,4,5,6,7,8,9};

std::vector<const char\*> fruits {"Apple", "Orange", "Banana", "Cherry"};

## 11. Use C++-style casting instead of C-style casting: static\_cast<>, dynamic\_cast<> …

C++ style conversion allows more compiler checks and is considerably safer. Also, the C++ conversion style is more visible and has the ability to search.

Instead:

int i = (int) z;

a = (char)x;

y = (int)b;

Use like this:

int i = static\_cast<int>(z);

a = static\_cast<char>(x);

y = static\_cast<int>(b);

## 12. Don’t use macros, use constants with const

Macros are a practice widely used by old programmers of the C language, but in C++ it is a bad idea, even for C itself, among several evils, they include:

* Cannot debug MACROS , even using -g, -Wall, -Werror, …
* MACROS have no “namespace”, so the possibility of duplicating the name is considerably high.
* And among others.

Instead:

#define PI 3.14

#define printf std::cout

#define OUTPUT(x) myfunction(x)

Use like this:

const double PI{3.14};

const typedef std::vector<int> VEC;

## 13. Do not use FILE \* file or DIR from C, use std::filesystem

It was implemented starting with C++17 and is also critical for portability. Not to mention that it has several non-member functions, which facilitate the manipulation of strings separately, especially when you work with files and directories on different disks.

Instead:

bool exists( const char \* home ){

DIR \* path = opendir( home );

if( path ){

closedir(path);

return true;

}else{

return false;

}

}

int main(){

char \* home = getenv("HOME");

const char \* dir = "/Downloads";

strcat(home,dir);

exists(home) ? printf("Yes") : printf("Not");

return 0;

}

Use like this:

bool exists( const std::string &path){

return std::filesystem::exists(path);

}

int main(){

std::string home = getenv("HOME");

home = home + "/";

std::cout << ( exists(home + "Downloads")

? "Yes" : "Not") << '\n';

return 0;

}

## 14. Use template instead of creating multiple classes that do the same thing, make use of the using keyword for *alias* of names.

Instead:

class Vector2i {

public:

int x, y;

Vector2i(int x, int y): x(x), y(y){ }

};

class Vector2f {

public:

float x, y;

Vector2f(float x, float y): x(x), y(y){ }

};

class Vector2u {

public:

unsigned x, y;

Vector2u(unsigned x, unsigned y): x(x), y(y){ }

};

Use like this:

template<class T>

class Vector2 {

public:

Tx,y;

Vector2(T x, T y): x(x), y(y){ }

};

using Vector2i = Vector2<int>;

using Vector2f = Vector2<float>;

using Vector2u = Vector2<unsigned>;

## 15. Prefer ++i than i++

It increments before printing, this seems to be negligible, but I’ve had unexpected results .

Instead:

std::cout << "Points: " << i++ << '\n';

Use like this:

std::cout << "Points: " << ++i << '\n';

## 16. Avoid using \n inside double quotes when using just it, remember it is a char and not a string.

Using double quotes makes the compiler interpret it as a const char \*, with single quotes it is really what a char is.

Instead:

std::cout << hello << "\n";

Use like this:

std::cout << hello << '\n';

**17. Use lambda**

Lambdas solve a problem of readability, expressiveness and practicality. In addition to making your code cleaner and LIKE A BOSS!

Instead:

int sum(int x, int y){

return x + y;

}

int main( int argc , char \*\*argv ){

int n = sum(5, 4);

std::cout << n << '\n';

return 0;

}

Use like this:

int n = [](int x, int y){

return x + y;

}(5, 4);

std::cout << n << '\n';

## 18. Prefer foreach style for loops over traditional

This is a modern practice and don’t forget to always assign reference to the index!

Instead:

std::unordered\_map<const char\*, uint32\_t> levels = {

{"Level Map", 39},

{"Hard Boss", 112},

{"Very Easy", 42},

{"End", 458}

};

for( auto i = levels.begin(); i != levels.end(); ++i ){

std::cout << i->first << " → " << i->second << '\n';

}

Use like this:

std::unordered\_map<const char\*, uint32\_t> levels = {

{"Level Map", 39},

{"Hard Boss", 112},

{"Very Easy", 42},

{"End", 458}

};

for( auto&[i, j] : levels){

std::cout << i << " → " << j << '\n';

}

## 19. C++ doesn’t have methods and properties, but MEMBERS AND MEMBER FUNCTIONS

Programmers who came from other programming languages also tend to inherit terms like: Methods and Properties in Object Oriented.

In C++ there are no methods and properties, but MEMBERS and MEMBER FUNCTIONS .

Using these terms in your software documentation confuses C++ programmers who are not used to it.

## 20. Use additional tools

Whenever possible use additional tools to debug, optimize and test your code, such as:

* GNU GDB <https://www.sourceware.org/gdb/>
* Tracy Profiler <https://github.com/wolfpld/tracy>
* CMake.
* AutoTools.
* Xmake.

If you want to see a video based on this article, follow the same below.

<https://www.youtube.com/watch?v=PftI7P3Kehs&t=3s>

# References

* <https://www.stroustrup.com/>
* <https://en.cppreference.com/>
* <https://github.com/cpp-best-practices>
* <https://terminalroot.com/>