Modern C++

A short, opiniated and not very well thought through introduction to modern C++

Simon Egli

Prerequisites

C++14 (17?)

Good, standard compatible compilers

Non-minimal system

Red = BAD GREEN = RECOMMENDED

KISS

Small functions

Don't repeat yourself

Prefer simple over clever

Convention over Configuration

Self explanatory – the code is the documentation

SOLID

Wikipedia:

Single responsibility principle[6]

A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

Open–closed principle[7]

"Software entities ... should be open for extension, but closed for modification."

Liskov substitution principle[8]

"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.

Interface segregation principle[9]

"Many client-specific interfaces are better than one general-purpose interface."[4]

Dependency inversion principle[10]

One should "depend upon abstractions, [not] concretions."[4]

No long functions

Every function that is longer than e.g. 30 lines should be broken down

Maintainability, code reuse, decluttering

Forces you to break down complexity

Return early

Bad:

```
 if (someCondition)
    {
        // Do Something
    }
```

Good:

```
if (!someCondition)
        return;
```

Use {} blocks – scope things

Is safer and prevents errors

```
int i{5};
{
int i{6};
}
Std::cout << i;
```

Use containers

Vector

Array

List

(Unordered) map

Queue

Stack

Use std::array/vector over array[]

Array is unsafe

std::array/vector are safer and containers

```
std::array<int, 3> a1{ {1, 2, 3} };
```

Use algorithm library

for_each

count/count_if

find/find_if

random_shuffle,rotate,unique,reverse,stable_partition, binary_search,min,max

Transform

```
std::string s("hello");

std::transform(s.begin(), s.end(),
      std::back_inserter(ordinals),
      [](unsigned char c) -> std::size_t { return c; });
```

→ 72 69 76 76 79

Use range based for loops

Simpler and less room for errors

May be faster

```
for (const auto& x : v)    // OK
    cout << x << '\n';
```

Use modern variant of std includes

Use cstdlib vs stdlib.h

stdlib.h is for c not for c++, it is deprecated in C++

Cstdlib is namespaced

Use {}-enclosed initializer lists

Simpler, more general, less ambiguous and safer

No narrowing

```
int x {7.9};   // error: narrowing
int y = 7.9;   // OK: y becomes 7. Hope for a compiler warning
int z{}; // default initialization
```

Don't use macros

```
constexpr float x = 42.0;
constexpr int mynumber(int n)
{
   Return 100 + n * 5;
}

template<typename T> T square(T a, T b) { return a * b; }

static_assert(sizeof(int)==4, "Int is not size 4");
```

No raw loops

Is a code smell – have you missed a container?

```
for(size_t i=0; i<=l; i++)
 std::cout << array[0] << std::endl;
```

```
for(const auto& i: array)
  std::cout << i << std::endl;
```

Use nullptr

Nullptr is what is used in C++ to signal an empty pointer

Enhances type deduction and thus safety

Enhances readability

Use unnamed (anonymous) namespaces

Only in source file

Put «helpers» in there

Encapsulates logic and prevents leaks

Don't return via function parameters

Makes things harder to read and more error prone

```
void func(int param1, int param2, int* out);
```

Use by value, std::tuple, …

```
return std::make_tuple(1, -1);
```

Use std::chrono for time stuff

This is the standard

Handles duration, calculations, conversions, ...

```
std::chrono::seconds sec(59);
std::chrono::duration_cast<std::chrono::minutes>(sec).count(); -> 0
```

Use lambdas

Encapsulates code

Makes functions more readable

```
std::vector<int> v;
...
auto it = std::find_if(v.cbegin(), v.cend(),
[](int i) -> bool { return i > 0 && i < 10; });
```

Let C++ handle dynamic memory

Using shared_ptr

Vectors

Maps

malloc/free new/delete is a code smell

Learn the basics about templates

Many people don't and miss easy opportunities to generalize code

```
template <typename T>
class MyClass {
    public:
        MyClass(T var):var_(var) {}
    private:
        T var_;
};
```

Use #pragma once (maybe)

Eases headaches and decreases complexity

Goes against CppCoreGuidelines,

Because it is non-standard

Mutex and lock_guards

```
std::mutex m;
{
std::lock_guard<std::mutex> L(m);
...
}
```

Bitsets

```
#include <bitset>
#include <iostream>

int main()
{
    std::bitset<4> b1("0110");
    std::bitset<4> b2("0011");
    std::cout << "b1 & b2: " << (b1 & b2) << '\n';
    std::cout << "b1 | b2: " << (b1 | b2) << '\n';
    std::cout << "b1 ^ b2: " << (b1 ^ b2) << '\n';
}
```

Output iterator

```
// ostream_iterator example
#include <iostream>     // std::cout
#include <iterator>     // std::ostream_iterator
#include <vector>       // std::vector
#include <algorithm>    // std::copy

int main () {
  std::vector<int> myvector;
  for (int i=1; i<10; ++i) myvector.push_back(i*10);

  std::ostream_iterator<int> out_it (std::cout,", ");
  std::copy ( myvector.begin(), myvector.end(), out_it );
  return 0;
} → 10, 20, 30, 40, 50, 60, 70, 80, 90,
```

Thread

```
#include <iostream>
#include <thread>
#include <chrono>

void foo()
{
   // simulate expensive operation
   std::this_thread::sleep_for(std::chrono::seconds(1));
}

   std::cout << "starting first helper...\n";
   std::thread helper1(foo);
```

Futures

```
std::future<double> f =
std::async(std::launch::async, []{ return
bestValue(10, 20); });
...
while (f.wait_for(std::chrono::seconds(0)) !=
std::future_status::ready) {
...
}
double val = f.get(); // grab result
```

Build using cmake

Much easier to handle than big makefiles

Can generate (and search/use) libraries

Wide support of libraries, e.g. Gtest/Gmock

```
cmake_minimum_required(VERSION 2.8.9)
project (hello)
add_executable(hello helloworld.cpp)
```

Try clang compilers

Easier error messages

Better analytics

Static code analysis

Code formatting

Use clang-format

Define style in .clang-format file

Use it to automatically enforce coding style

Use git

Best workflow

Used by most projects nowadays

There are nice GUIs – does not need be hard

Can be hard (inconsistend) if you want to deep dive in to git

Use Continous build servers

Build every commit

Eliminates «works on my machine» problems

Gives people early feedback

Ccache to speed up compilation

Nice for large projects

Speeds up e.g. make clean && make all

https://ccache.dev/manual/3.7.1.html

Ressources

https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

https://github.com/lefticus/cppbestpractices

https://github.com/rigtorp/awesome-modern-cpp

https://bbv.ch/images/bbv/pdf/downloads/booklets/C++17_DS.pdf

Scott Meyers books about the topic

Modern c++

A short, opiniated and not very well thought through introduction to modern C++

Simon Egli