Best Practices for Modern C++

Rainer Grimm

Training, Coaching, and Technology Consulting

C++ Core Guidelines

- Why do we need best practices?
  - C++ is a complex language in a complex domain.
  - An new C++ standard is published all three years.
  - C++ is used in safety-critical systems.

C++ Core Guidelines has community-driven guidelines for writing good software.

C++ Core Guidelines

The C++ Core Guidelines consist of 350 rules and 600 pages.

- Sections
  - Philosophy
  - Interfaces
  - Functions
  - Classes and Class Hierarchies
  - Enumerations
  - Resource Management
  - Expressions and Statements
  - Performance
  - Concurrency
  - Error Handling
  - Constants and Immutability
  - Templates and Generic Programming
  - C-Style Programming
  - The Standard Library
- Supporting Sections
  - Guidlines Support Library

Philosophy

The philosophical rules provide rationales for the following concrete rules. Ideally, the concrete rules can be derived from the philosophical rules.

- Express intent and ideas directly in code.
- Write in ISO Standard C++ and use support libraries and supporting tools.
- A program should be statically type-safe and should, therefore, check at compile-time. When this is not possible, catch run-time errors early.
- Don't waste resources such as space or time.
- Encapsulate messy constructs behind a stable interface.

Clean Code in C++

- **Interfaces**
- Functions
- Classes and Class Hierarchies
- Resource Management
- Expressions and Statements
- Performance
- Concurrency
- Error Handling
- Constants and Immutability
- Templates and Generic Programming

Interfaces

An interface is a contract between a service provider and a service user.

- Avoid globals and singletons. They break
  - testability
  - refactoring
  - optimization
  - concurrency

```
int glob{2011};

int mutiply(int fac) {
    glob *= glob;
    return glob * fac;
}
```

Interfaces

Interfaces should

- be explicit
- be precisely and strongly type
- have a low number of arguments
- separate similar arguments

```
void showRectangle(double a, double b, double c, double d) {
    a = floor(a);
    b = ceil(b);
...
}

void showRectangle(Point top_left, Point bottom_right);
```

Interfaces

Don’t pass arrays as single pointer.

```
template <typename T>
void copy_n(const T* p, T* q, int n) { ... }

template <typename T>
void copy(std::span<const T> src, std::span<T> des){ ... }

...

int a[100] = {0, };
int b[100] = {0, };
copy_n(a, b, 101);
copy(a, b);
```

Interfaces

- Examples:
  - `singleton.cpp`
  - `singletonMeyer.cpp`
- Exercises:
  - Interfaces have a functional and a non-functional data channel. What is the functional and the non-functional data channel?
  - Functions support by design interfaces. Which other software components should also have interfaces?
  - The singleton pattern is partly seen as a pattern and partly seen as an anti-pattern. What are the pros and cons of the singleton pattern?
- Further information:
  - Interfaces

Clean Code in C++

- Interfaces
- **Functions**
- Classes and Class Hierarchies
- Resource Management
- Expressions and Statements
- Performance
- Concurrency
- Error Handling
- Constants and Immutability
- Templates and Generic Programming

Functions

Software developers master complexity by dividing complex tasks into smaller units. Functions are "the most critical part in most interfaces”.

- Functions should perform one operation. The benefits are
  - good names by design
  - short functions, which can easily be understood
  - testability by design

```
void read_and_print() {
    int x;
    std::cin >> x; // check for errors
    std::cout << x << '\n';
}
```

Functions

Make your functions `constexpr` if possible.

- `constexpr` functions
  - have the potential to run at compile-time.
  - are almost pure.
  - are thread-safe when executed at compile-time.

```
constexpr auto gcd(int a, int b) {
    while (b != 0) {
        auto t= b;
        b = a % b;
        a = t;
    }
    return a;
}
```

Functions

Distinguish between in, in/out, and out parameter

	Cheap or impossible to copy	**Cheap or moderate costs to move and don't know**	**Expensive to move**
In	func(X)	func(const X&)	
In & retain "copy"			
In & move from	func(X&&)		
In/Out	func(X&)		
Out	X func()		func(X&)

Kind of data:

- Cheap of impossible to copy: int or std::unique_ptr
- Cheap to move: std::vector or std::string
- Moderate costs to move: std::vector<BigPOD>
- Don't know: template
- Expensive to move: std::array<BigPod>

Functions

Costs of operations:

- cheap to copy: <= 3 ints
- cheap of moderate to move: <= 1000 bytes without memory allocation
- expensive to move: >= 1000 bytes

RVO (Return Value Optimization)

```
Type f() {
  return Type{}; // no copy
}
Type my = f();   // no copy
```

NRVO (Named Return Value Optimization)

```
Type f() {
  Type myVal;
  return myVal; // one copy
}
Type my = f();  // no copy
```

Functions

Ownership semantic of function parameters

Example	Ownership Semantic
`func(value)`	`func` is an independent owner of the resource
`func(pointer*)`	`func` has borrowed the resource
`func(reference&)`	`func` has borrowed the resource
`func(std::unique_ptr)`	`func` is an independent owner of the resource
`func(shared_ptr)`	`func` is a shared owner of the resource

Functions

- Examples:
  - `constexpr.cpp`
  - `ownershipSemantic.cpp`

- Exercises:
  - The function read_and_print has many issues:

```
void read_and_print() {
    int x;
    std::cin >> x; // check for errors
    std::cout << x << '\n';
}
```

    - Refactor the function and use it.
    - Solution: `readAndPrint.cpp`
  - Analyze the program `constexpr.cpp` with the help of the Compiler Explorer.
    - What are the advantages of `constexpr` functions?

Functions

- Further information:
  - Functions

Clean Code in C++

- Interfaces
- Functions
- **Classes and Class Hierarchies**
- Resource Management
- Expressions and Statements
- Performance
- Concurrency
- Error Handling
- Constants and Immutability
- Templates and Generic Programming

Classes and Class Hierarchies

A class is a user-defined type where the programmer specifies the behavior. Class hierarchies organize related classes into hierarchical structures.

Class versus struct

- Use a class if it has an invariant
- Establish the invariant in a constructor

```
struct Point {
    int x;
    int y;
};
```

```
class Date {
 public:
     Date(int yy, Month mm, char dd);
 private:
     int y;
     Month m;
     char d;
};
```

Concrete Types

A concrete type (value type) is not part of a type hierarchy.
It can be created on the stack.

A concrete type should be regular.

- Default constructor: `X()`
- Copy constructor: `X(const X&)`
- Copy assignment: `operator = (const X&)` ➡ **`Big Six`**
- Move constructor: `X(X&&)`
- Move assignment: `operator = (X&&)`
- Destructor: `~(X)`
- Swap operator: `swap(X&, X&)`
- Equality operator: `operator == (const X&)`

Classes and Class Hierarchies

The Big Six

- The compiler can generate them
  - E.g.: The compiler can autogenerate the move constructor if all members and all bases have a move constructor.
- You can request a special member function via `default`
- You can delete a generated function via `delete`
- Define all of them or none of them (rule of six or rule of zero)
- Define them consistent
- There are dependencies between the big six

Classes and Class Hierarchies

	compiler implicitly declares					
user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

by [Howard Hinnant]

- user-declared: a method which is used (defined, `default`ed, or `delete`d)
- defaulted: a method which the compiler generates or is requested via `default`

Classes and Class Hierarchies

- Examples:
  - `delete.cpp`
  - `swap.cpp`
  - `bigArray.cpp`
- Exercises:
  - In the program `bigArray.cpp`, a `BigArray` with 10 billion entries will be pushed onto a `std::vector`.
    - Compile the program and measure its performance.
    - Extend `BigArray` with move semantic and measure the performance once more. How big is the performance gain?
      - Solution: `bigArray.cpp`

Classes and Class Hierarchies

- Further information:
  - Classes and class hierarchies
  - Rvalue References Explained from Thomas Becker

Constructor

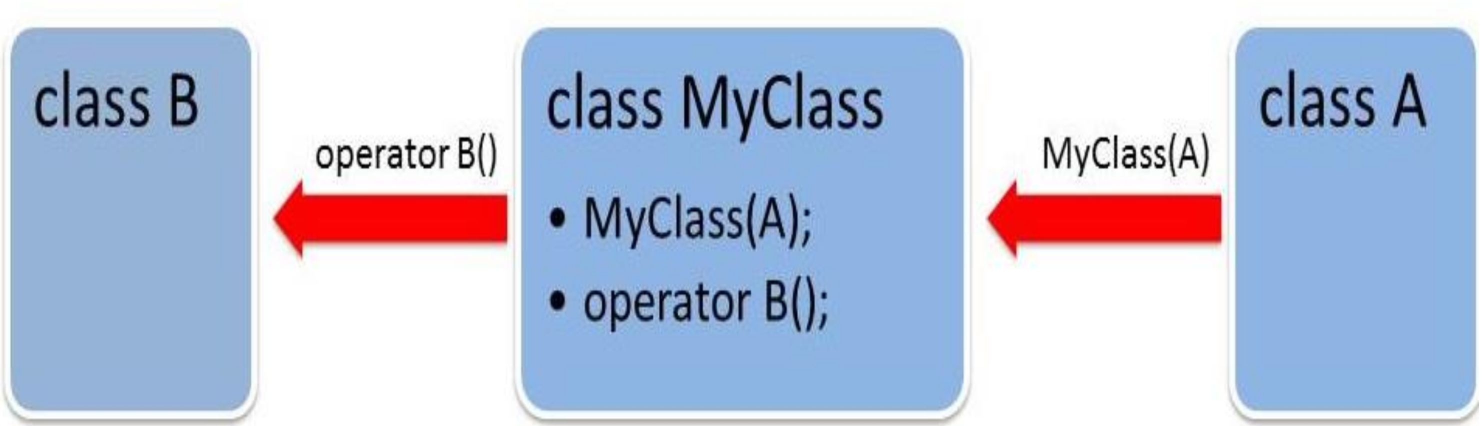
Don’t define a default constructor that only initializes data members; use member initialization instead

```
struct Widget {
    Widget() = default;
    Widget(int w): width(w) {}
 private:
     int width = 640;
};
```

Define the default behavior of each object in the class body. Use explicit constructors for variations of the default behavior.

Conversion Constructor and Operator

Make single-element constructors (conversion constructor) and conversions operators `explicit`.




```
class MyClass{
  public:
    explicit MyClass(A){}    // converting constructor
    explicit operator B(){}  // converting operator
};
```

Delegating Constructor

Use delegating constructor to represent common actions for all constructors of a class.

```
class Degree {
 public:
    explicit Degree(int deg) {
        degree= deg % 360;
        if (degree < 0) degree += 360;
    }
    Degree(): Degree(0) {}
    explicit Degree(double deg):
        Degree(static_cast<int>(std::ceil(deg))) {}
 private:
    int degree;
};
```

Polymorphic Class

A polymorphic class should suppress copying. A polymorphic class is a class that defines or inherits at least one virtual function.

Danger of slicing

Slicing: copying an object during assignment or initialization returns only a part of the object.

Destructors

- Define a destructor if a class needs an explicit action at object destruction
  - E.g.: the class own pointers or references

- A base class constructor should either be public and virtual, or protected and non-virtual
  - `public` and virtual:
    - instances of derived classes can be destroyed through a base class pointer; the same holds for references
  - `protected` and non-virtual:
    - instances of derived classes can not be destroyed through a base class pointer; the same holds for references

- Destructors should not fail; make them `noexcept`

Constructor/Destructor (virtual)

Don’t call virtual functions in constructors and destructors.

- Pure virtual: ⇒ undefined behavior
- Virtual: ⇒ virtual call mechanism is disabled

Operator Overloading

- Use non-member functions for symmetric operators

```
class MyInt {
    int num;
 public:
    explicit MyInt(int n): num(n) {};
    friend bool operator==(const MyInt& lhs, const MyInt& rhs) {
         return lhs.num == rhs.num;
    }
    friend bool operator==(int lhs, const MyInt& rhs) {
        return lhs == rhs.num;
    }
    friend bool operator==(const MyInt& lhs, int rhs) {
        return lhs.num == rhs;
    }
};
```

Classes and Class Hierarchies

- Examples:
  - `classMemberInitializerWidget.cpp`
  - `conversionOperator.cpp`
  - `convertingConstructor.cpp`
  - `slice.cpp`

Classes and Class Hierarchies

- Exercises:
  - The program `convertingConstructor.cpp` supports basic type-safe arithmetic with user-defined literals.
    - Execute the program.
    - I made an error in the program. Single-argument constructors should be `explicit`. Fix it.
    - Extend the program so that floating-point numbers can be added to the user-defined literals.
      - Solution: `convertingConstructor.cpp`
  - Refactor the constructors.
    - The constructors of the class `Widget` in `classMemberInitializerWidget.cpp` can be simplified. Use direct initialization for the class members.
      - Solution: `classMemberInitializerWidget.cpp`

Classes and Class Hierarchies

- Further information:
  - Classes and class hierarchies
  - Rvalue References Explained from Thomas Becker

Class Hierarchies

A class hierarchy represents a set of hierarchically organized concepts. Base classes act typically as interfaces.

- **Interface inheritance** uses public inheritance. It separates users from implementations to allow derived classes to be added and changed without affecting the users of base classes.

- **Implementation inheritance** uses often private inheritance. Typically, the derived class provides its functionality by adapting functionality from base classes.

Class Hierarchies

Use class hierarchies to represent concepts with inherent hierarchical structure.

```
template<typename T>
class Container {
 public:
       // list operations:
    virtual T& get() = 0;
    virtual void put(T&) = 0;
    virtual void insert(Position) = 0;
      // vector operations:
    virtual T& operator[](int) = 0;
    virtual void sort() = 0;
      // tree operations:
    virtual void balance() = 0;
};
```

Abstract Classes

- If a case class is used as an interface, make it an abstract class
- Use abstract classes when complete separation of interface and implementation is needed
- An abstract class typically doesn‘t need a constructor

Virtuality

- A class with a virtual function should have a `public` and virtual or a `protected` destructor

- Virtual functions should exactly specify one of `virtual`, `override`, or `final`

- For making deep copies of polymorphic classes prefer a virtual `clone` instead of copy construction/assignment

  ➡ Beware of slicing

  - **Covariant return type:** allows it for an overriding member function to return a subtype of the return type of the overridden member function

Traps (Shadowing)

- Create an overload set for a derived class and its bases with `using`

```
struct Base {
    void func(double) { std::cout << "f(double) \n"; }
};

struct Derived: public Base {
     void func(int) { std::cout << "f(int) \n"; }
};
...
Derived der;
der.func(2011);     // f.int()
der.func(2020.5);   // f.int()
```

```
struct Derived: public Base {
    void func(int i) { std::cout << "f(int) \n"; }
    using Base::func; // exposes func(double)
};
```

Traps (Virtual Functions and Defaults)

- Do not provide default arguments for a virtual function and an overrider

```
struct Base {
    virtual int multiply(int value, int factor = 2) = 0;
};
struct Derived : public Base {
    int multiply(int value, int factor = 10) override {
        return factor * value;
    }
};
```

Unions

- Use `union`s to save memory
- Avoid “naked” `union`s
- Use tagged `union`s (`std::variant`)

```
std::variant<int, float> v, w;
v = 12;
int i = std::get<int>(v);

w = std::get<int>(v);
w = std::get<0>(v);
w = v;
```

Classes and Class Hierarchies

- Examples:
  - `adapter.cpp`
  - `virtualCall.cpp`
  - `cloneFunction.cpp`
  - `overrider.cpp`
  - `variant.cpp`
- Exercises:
  - C++ support interface inheritance and implementation inheritance.
    - Do you know a use-case for implementation inheritance?
    - Study the programm `adapter.cpp` implementing the [adapter pattern]() using multiple inheritance.
    - Do you know another way to implement the adapter pattern?

Classes and Class Hierarchies

- Never assign a pointer to an array of derived class objects to a pointer to its base
  - What is wrong with this code snippet?

```
struct Base { int x; };
struct Derived : Base { int y; };
Derived d[] = {{1, 2}, {3, 4}, {5, 6}};
Base* pB = d;
pB[1].x = 7;
```

- Further information:
  - [Classes and class hierarchies]()

Clean Code in C++

- Interfaces
- Functions
- Classes and Class Hierarchies
- **Resource Management**
- Expressions and Statements
- Performance
- Concurrency
- Error Handling
- Constants and Immutability
- Templates and Generic Programming

Resource Management

A resource is something that you have to manage. That means you have to acquire and release it, or you have to protect it

- The critical question to resources is: Who is the owner?

Resource Management: Ownership

- **Local objects**: The C++ runtime as the owner automatically manages the resources. This holds for global or class members.
- **References**: I'm not the owner. I only borrowed the resource that cannot be empty. I must not delete the resource.
- **Raw pointers**: I'm not the owner. I only borrowed the resource that can be empty. I must not delete the resource.
- **`std::unique_ptr`**: I'm the exclusive owner of the resource.
- **`std::shared_ptr`**: I share the resource with other shared pointer. I may explicitly release my shared ownership.
- **`std::weak_ptr`**: I'm not the owner of the resource, but I may become the temporary owner of the resource

Resource Management: RAII

RAII stands for **R**esource **A**cquisition **I**s **I**nitialization.

- Key idea:
  - Create a local, proxy object for your resource.
  - The constructor of the proxy acquires the resource and the destructor of the proxy releases the resource.

  ⇒ The C++ runtimes manages the lifetime of the proxy and, therefore, of the resource.

- Implementations
  - Smart pointers
  - Locks
  - Containers of the STL
  - `std::jthread`

Resource Management: NNN

NNN stands for **N**o **N**aked **N**ew and means, that memory allocation should not be done as a standalone operation, but insight a manager object.

Smart Pointer:

- **std::unique_ptr**: exclusive owner
- **std::shared_ptr**: shared owner
- **std::weak_ptr**: temporary owner

std::unique_ptr

- Allocate the resource not outside

```
int* myInt = new int(2011);
std::unique_ptr<int> uniq1 = std::unique_ptr<int>(myInt);
std::unique_ptr<int> uniq2 = std::unique_ptr<int>(myInt);
```

- Prefer `std::make_unique` to `std::unique_ptr`

```
func(std::unique_ptr<int>(new int(2011)),
     std::unique_ptr<int>(new int(2014)));
```

possibly memory leak

```
func(std::make_unique<int>(2011),
     std::make_unique<int>(2014));
```

no memory leak guaranteed (performance improvement)

- Prefer `std::unique_ptr` to `std::shared_ptr`

std::shared_ptr

- Use it only to express shared ownersnip
  - `std::unique_ptr` can be moved

```
void func(std::unique_ptr<int> myUniq);
…
auto myUniq = std::make_unique<int>(2014));
func(std::move(myUniq));
```

- Prefer `std::make_shared` to `std::shared_ptr`
  - is exception-safe
  - needs one allocation instead of two
- The control-block is thread-safe, but not the resource
  - `std::atomic_shared_ptr` with C++20
- Can be used with an own deleter (also `std::unique_ptr`)

```
std::shared_ptr<int>(2011, Deleter());
```

Smart Pointer as Parameter

Function Signature	Semantic
`func(std::unique_ptr<int>)`	`func` takes ownership
`func(std::unique_ptr<int>&)`	`func` might reseat `int`
`func(std::shared_ptr<int>)`	`func` shared ownership
`func(std::shared_ptr<int>&)`	`func` might reseat `int`
`func(const std::shared_ptr<int>&)`	`func` might retain a reference counter

- `func(const std::shared_ptr<int>&)`
  - Adds no value
  - A raw pointer or a reference would also be fine

Resource Management

- Examples:
  - raii.cpp
  - sharedPtrDeleter.cpp
  - lifetimeSemantic.cpp

- Exercises:
  - Write a simple lock such as std::lock_guard which takes care of its mutex.
    - Solution: myGuard.cpp
  - Create 100 million std::shared_ptr<int> with std::shared_ptr and std::make_shared. Measure the performance.
    - Solution: sharedPtrPerformance.cpp and makeSharedPerformance.cpp

Resource Management

- Analyze the program `lifetimeSemantic.cpp`.
  - Compile and run the program.
  - Why is the function `asSmartPointerBad` bad?

- Assume, you have the following function.

```
void shared(std::shared_ptr<Widget>& shaPtr){
    oldLongRunningFunc(*shaPtr);
}
```

  - How can you ensure, that the underlying resource of the `shaPtr` stays valid during the call of the function `oldLongRunningFunc`?

- Further information:
  - [Resource management]

Clean Code in C++

- Interfaces
- Functions
- Classes and Class Hierarchies
- Resource Management
- **Expressions and Statements**
- Performance
- Concurrency
- Error Handling
- Constants and Immutability
- Templates and Generic Programming

Expressions and Statements

Expressions and statements are the lowest and most direct way of expressing actions and computation.

- An **expression** evaluates to a value.
- A **statement** does something and is often composed of expressions or statements.

```
5 * 5;             // expression
std::cout << 25; // print statement
auto a = 10;       // assignment statement
auto b = 5 * 5;   // expression statement
```

Good Names

A **declaration** is a statement which introduces a name into a scope.

- Good names are probably the most important rule for good software.

- Good names should
  - be self-explanatory. The shorter the scope, the shorter the name.
  - don‘t be reused in nested scopes.
  - should avoid similar-looking names.

```
if (readable(i1 + l1 + ol + o1 + o0 + ol + o1 + I0 + l0)) {
    surprise();
}
```

Good Names

- be as local as possible.

```
std::map<int,std::string> myMap;
if (auto result = myMap.insert(value); result.second) {
  useResult(result.first);
}
else {
} // result is automatically destroyed
```

- not have `All_CAPS` names.

```
#define NE !=                          // somewhere in a header
enum Coord { N, NE, NW, S, SE, SW}; // in another header
switch (direction) {                   // in some cpp
case N:
    // ...
case NE:
   // ...
}
```

Good Names

- be declared exclusively per line.

```
char* p, p2;
char a = 'a';
p = &a;
p2 = a;
int a = 7, b = 9, c, d = 10, e = 3;
```

`auto`: Don’t get the wrong type

```
auto a= 5;
auto b= 10;
auto sum =  a * b * 3;
auto res = sum + 10;
std::cout << typeid(res).name();          // i

auto a2 = 5;
auto b2 = 10.5;
auto sum2 = a2 * b2 * 3;
auto res2 = sum2 * 10;
std::cout << typeid(res2).name();         // d

auto a3 = 5;
auto b3 = 10;
auto sum3 = a3 * b3 * 3.1f;
auto res3 = sum3 * 10;
std::cout << typeid(res3).name();         // f
```

`auto`: Always initialize

```
struct T1 {};

class T2{
 public:
    T2() {}
};

auto n = 0;

int main() {
    auto n2 = 0;
    auto s = ""s;
    auto t1 = T1();
    auto t2 = T2();
}
```

{}-Initialization

{}-Initialization is

- always applicable.
- overcomes the most vexing parse.
- prevents narrowing conversion.

- `auto`

Example	**C++11**	**C++17**
`auto i{1};`	`std::initializer_list<int>`	`int`
`auto i = {1};`	`std::initializer_list<int>`	`std::initializer_list<int>`
`auto i{1, 2};`	`std::initializer_list<int>`	`ERROR`
`auto i = {1, 2};`	`std::initializer_list<int>`	`std::initializer_list<int>`

`nullptr`

Use `nullptr` instead of `0` or `NULL` to initialize a pointer.

- **0**: The literal 0 can be the null pointer `(void*)0` or the number `0`.
- **NULL**: NULL is a macro and, therefore, you don't know what's inside.
  - A possible implementation according to cppreference.com:

```
#define NULL 0
//since C++11
#define NULL nullptr
```

➡ The null pointer `0` or `NULL` do not work in generic code.

Casts

- If necessary, use named casts
  - **`static_cast`**: casts similar types such as pointers or numbers
  - **`const_cast`**: adds or removes const or volatile
  - **`reinterpret_cast`**: casts pointers or integrals and pointers
  - **`dynamic_cast`**: casts polymorphic pointers or references in the same class hierarchy
  - **`std::move`**: converts to an rvalue reference
  - **`std::forward`**: casts an lvalue to an lvalue reference and an rvalue to an rvalue reference
- Don’t cast away const from an original const object

```
const int constInt = 10;
const int* pToConstInt = &constInt;
int* pToInt = const_cast<int*>(pToConstInt);
*pToInt = 12;    // undefined behavior
```

Statements

- Prefer algorithms of the STL to loops

```
std::vector<int> vec = {-10, 5, 0, 3, -20, 31};
std::sort(std::execute::par, vec.begin(), vec.end());
std::sort(std::execute::par_unseq, vec.begin(), vec.end())
```

- Prefer range-based for-loops to for-loops; prefer for-loops to while-loops
- Don’t rely on implicit fallthrough in switch statements
- Use `[[fallthrough]]` to indicate that fallthrough is intentional

```
switch (n) {
    case 1:
        g();
        [[fallthrough]];
    case 2:
        h();
}
```

Arithmetic

- Don’t mix signed an unsigned arithmetic.

```
#include <iostream>

int main() {
    int x = -3;
    unsigned int y = 7;
    std::cout << x - y << std::endl; // 4294967286
    std::cout << x + y << std::endl; // 4
    std::cout << x * y << std::endl; // 4294967275
    std::cout << x / y << std::endl; // 613566756
}
```

Expressions and Statements

- Examples:

  - `shadowClass.cpp`
  - `uniformInitialization.cpp`
  - `mostVexingParse.cpp`
  - `narrowingConversion.cpp`
  - `nullPointer.cpp`
  - `unspecified.cpp`
  - `strange1.cpp` and `strange2.cpp`
  - `overUnderflow.cpp`

- Exercises:
  - Why does the compilation of the program `shadowClass.cpp` fail. Fix the bug.
    - Solution: `shadowClass.cpp`

Expressions and Statements

- Exercises:
  - Fix the error in the program `mostVexingParse.cpp`.
    - Solution: `mostVexingParserSolved.cpp`
  - Fix the error in the program `narrowingConversion.cpp`.
    - Solution: `narrowingConversionSolved.cpp`
  - Execute the program `unspecified.cpp` on GCC and clang. Which compiler is right?
  - The programs `strange1.cpp` and `strange2.cpp` produce different results. In the program `strange1.cpp` the summation variable `x` is an `unsigned short` and in `strange2.cpp` `x` is an `short`. What is happening?
  - How long does the program `overUnderflow.cpp` run on your platform? Try it out.
- Further information:
  - [Expressions and statements]()

Clean Code in C++

- Interfaces
- Functions
- Classes and Class Hierarchies
- Resource Management
- Expressions and Statements
- **Performance**
- Concurrency
- Error Handling
- Constants and Immutability
- Templates and Generic Programming

Performance

Wrong optimization

- “premature optimization is the root of all evil” (Donald Knuth)

- Rule for optimization
  - Measure with real-world data
  - Build a base line
  - Versionize your performance test

- Importance of measuring
  - Which part of the program is the bottleneck?
  - How fast is good enough for the user?
  - How fast could the program potentially be?

Performance

Wrong assumptions

- Don’t assume that complicated code is necessarily faster than simple code
- Don’t assume that low-level code is necessarily faster than high-level code
- Don’t make claims about performance without measurements

➡ Get the ultimate truth to the optimized code with the Compiler Explorer

Performance

Enable Optimization

- Use move semantic if possible
  - Use cheap move operations instead of expensive copy operations
  - No memory allocation required $\Rightarrow$ no `std::bad_alloc` exception possible
  - You can use move-only types such as `std::unique_ptr`
- Rely on the static type system
  - Write local code
  - Write simple code
  - Give the compiler additional hints (`noexcept, final`)

Performance

- When your program could possible run at compile-time, make it `constexpr`
  - A `constexpr` function can run at compile-time or run-time

➡ You can execute a `constexpr` function at compile-time and at run-time.

- Respect cache lines
  - When you read an `int` from memory, an cache-line of typically 64 bytes (16 `int`’s) is read and stored in a fast cache

➡ Reading contiguous memory blocks is cache friendly

Performance

- Examples:
  - `singletonAcquireRelease.cpp`
  - `singletonMeyers.cpp`
  - `memoryAccess.cpp`

- Exercises:
  - Measure the performance of the programs `singletonMeyers.cpp` and `singletonAcquireRelease.cpp`.
    - Which program is faster?
    - What is the performance you can possibly get?

Performance

- Measure the performance of the program `memoryAccess.cpp`. Did you expected this numbers?
- The ordered associative containers such as `std::map` do not have a cache line aware layout. Implement a cache line aware fast variant of a `std::map`.
  - Solution: `flatMap.cpp`

- Further information:
  - [Performance]()

Clean Code in C++

- Interfaces
- Functions
- Classes and Class Hierarchies
- Resource Management
- Expressions and Statements
- Performance
- **Concurrency**
- Error Handling
- Constants and Immutability
- Templates and Generic Programming

Concurrency and Parallelism

Locks

- NNM (**N**o **N**aked **M**utex) → put the mutex immediately into a managing object such as a lock
- Use `std::lock` or `std::scoped_lock` to acquire multiplex mutexes atomically.
- Give a lock a name

```
{
    std::lock_guard<std::mutex> {m};
    std::cout << "CRITICAL SECTION" << std::endl;
}
```

Concurreny and Parallelism

Threads

- Prefer `std::jthread` to `std::thread`
- Don't detach a `thread`
- Pass small amounts of data between threads by value
- To share ownership between unrelated threads use `std::shared_ptr`

Condition variables

- Don't wait without a condition; be aware of spurious wakeup and lost wakeup
- Prefer tasks to condition variables, if possible

Concurrency and Parallelism

- Use each tool you can get to validate your concurrent code
  - ThreadSanitizer
    - Dynamic code analyzer
    - Part of clang 3.2 and GCC 4.8
    - Compile your program with –sanitize=thread -g -O2
  - CppMem
    - Static code analyser
    - Validates small code snippets, typically including atomics
    - Gives your deep insight into the C++ memory model

Concurrency and Parallelism

Parallelism

- Prefer the parallel algorithms of the STL to handcrafted solutions with threads

```
std::vector<int> v = {5, -3, 10, -5, -10, 22, 0};

std::sort(v.begin(), v.end());                              // sequential

std::sort(std::execution::seq, v.begin(), v.end());       // sequential
std::sort(std::execution::par, v.begin(), v.end());       // parallel
std::sort(std::execution::par_unseq, v.begin(), v.end()); // vectorized
```

Concurrency and Parallelism

Message passing

- Think in tasks (promises/future pairs) and not in threads
- Use a future to get a value or an exception from a concurrent task
- Prefer tasks to `std::condition_variables` to synchronize threads

Criteria	Condition Variables	Tasks
Multiple synchronizations	Yes	No
Critical section	Yes	No
Spurious wakeup	Yes	No
Lost wakeup	Yes	No

Concurrency and Parallelism

Atomics

- Don‘t program lock-free but only for very simply jobs
- Don‘t trust your intuition
- Carefully study the literature before you program lock-free

- **Herb Sutter**: *Lock-free programming is like playing with knives.*
- **Anthony Williams**: "*Lock-free programming is about how to shoot yourself in the foot.*"
- **Tony Van Eerd**: "*Lock-free coding is the last thing you want to do*."
- **Fedor Pikus**: "*Writing lock-free programs is hard. Writing correct lock-free programs is even harder.*"
- **Harald Böhm**: "*The rules are not obvious.* "

Concurrency and Parallelism

- Examples:
  - threadDetach.cpp
  - threadSharesOwnership.cpp
  - conditionVariable.cpp
  - transformExclusiveScan.cpp
  - promiseFutureException.cpp
  - promiseFutureSynchronize.cpp
  - sequentialConsistency.cpp
  - relaxedSemantic.cpp

Concurrency and Parallelism

- Exercises:
  - Why is the following code snippet very bad?

```
std::mutex m;
m.lock();
sharedVariable = unknownFunction();
m.unlock();
```

  - What are the issues of the program `threadDetach.cpp`? Fix the issue.

  - The threads in the program `threadSharesOwnership.cpp` share the variable `tmpInt`. Use a `std::shared_ptr` to share the resource between the threads.
    - Solution: `threadSharesOwnershipSmartPtr.cpp`

Concurrency and Parallelism

- Write a simple ping pong game.
  - Two threads should alternatively set a `bool` value to `true` or `false`. One thread set the value to `true` and notifies the other thread. The other thread sets the value to `false` and notifies the other thread. That play should end after a fixed amount of iterations.
  - Solution: `conditonVariablePingPong.cpp`

- Further information:
  - Concurrency
  - Anthony Williams: C++ Concurrency in Action. Manning Publications
  - Bartosz Milewski: Bartosz Milewski's Programming Cafe
  - Herb Sutter: Effective Concurrency
  - Jeff Preshing: Preshing on Programming

Clean Code in C++

- Interfaces
- Functions
- Classes and Class Hierarchies
- Resource Management
- Expressions and Statements
- Performance
- Concurrency
- **Error Handling**
- Constants and Immutability
- Templates and Generic Programming

Error Handling

Error handling consists of

- Detecting the error
- Transmitting information about an error to some handler code
- Preserve the state of a program in a valid state
- Avoid resource leaks

Abrahams Guarantees

- No-throw guarantee
- Strong exception safety (rollback semantic)
- Basic exception safety (invariants preserved; no leak)
- No exception safety

Error Handling

- Each software unit has two channels:
  - Regular channel: What the software unit should do.
  - Irregular channel: How the software unit should operate.
- Design your error handling around invariants. If invariants can not be established, throw.
- Use user-defined types for exceptions.
- Catch exceptions form specific to general.
- Use exceptions only for error handling.
- Never throw while you are a direct owner

Error Handling

- Exercises:
  - What is the issue with the following code?

```
int getIndex(std::vector<std::string>& vec,
             const std::string& x) {
    try {
        for (auto i = 0; i < vec.size(); ++i)
            if (vec[i] == x) throw i; // found x
    } catch (int i) {
        return i;
    }
    return -1; // not found
}
```

Error Handling

- What is the issue with the following code?

```
void leak(int x) {
    auto p = new int{7};
    if (x < 0) throw Get_me_out_of_here{};
    // ...
    delete p;
}
```

- Further information:
  - Error handling
  - David Abrahams: Exception-Safety in Generic Components

Clean Code in C++

- Interfaces
- Functions
- Classes and Class Hierarchies
- Resource Management
- Expressions and Statements
- Performance
- Concurrency
- Error Handling
- **Constants and Immutability**
- Templates and Generic Programming

Constants and Immutability

- By default, make objects immutable
  - Cannot be a victim of a data race
  - Guarantee that they are initialized in a thread-safe way
- By default, make functions `const`
  - Distinguish between physical and logical constness of an object
- Casting away `const` from a original `const` object is undefined behavior
- If possible, make your functions `constexpr`
  - Provides additional optimization opportunities
  - `constexpr` functions are almost pure

Constants and Immutability

- Examples:
  - `mutable.cpp`
  - `constCastAway.cpp`
- Exercises:
  - Implement a class `Lock` with two constant method `lock` and `unlock`. The class `Lock` should have a `std::mutex` which does the locking.
    - Solutions: `strategizedLockingCompileTime.cpp`
  - A `constexpr` function can run at run-time, if the results is not required at compile-time. How can you check if the `constexpr` function runs at compile-time or run-time?
- Further information:
  - [Constants and immutability]()

Clean Code in C++

- Interfaces
- Functions
- Classes and Class Hierarchies
- Resource Management
- Expressions and Statements
- Performance
- Concurrency
- Error Handling
- Constants and Immutability
- **Templates and Generic Programming**

Templates and Generic Programming

Use

- Use templates to raise the level of abstraction
- Use templates to express algorithms that apply to many argument types

Interfaces

- Use function objects (lambdas) to pass operations to algorithms.
- Let the compiler deduce the template arguments.
- Template arguments should be at least `SemiRegular` or `Regular`.

Templates and Generic Programming

- **Argument-dependent lookup**: Unqualified functions names are additionally looked up in the namespace of their arguments.

```
#include <iostream>
int main() {
    std::cout << "Argument-dependent lookup";
}
```

➡ Why is the `operator<<` found?

- Fake concepts with `std::enable_if`

Templates and Generic Programming

Definition

- Place non-dependent class template members in a non-templated base class.
- There are various ways to extend a user-defined type `MyType` to a generic functions such as `isSmaller`.

```
template <typename T>
bool isSmaller(T fir, T sec) {
    return fir < sec;
}
```

  - Implement `operator <` for `MyType`
  - Implement a full specialization for `MyType`
  - Extend `isSmaller` with a predicate

Templates and Generic Programming

Hierarchies

- Virtual member functions are instantiated each time in a class template. In contrast non-virtual member functions are only instantiated when needed.
- Member function templates can not be virtual.

Variadic Templates

- Factory functions in modern C++ rely on two powerful features: perfect forwarding and variadic templates
- Thanks to perfect forwarding and variadic templates, a factory function can accept an arbitrary many lvalues or rvalues.

Templates and Generic Programming

Metaprogramming

- Metaprogramming is programming at compile-time.
- Metaprogramming can be done with template metaprogramming, the type-traits library, or `constexpr` functions.
- Prefer `constexpr` functions to the type-traits library; prefer the type-traits library to template metaprogramming

Other rules

- Use a lambda if you need a simple operation in place.
- Give operations with the potential to reuse a name.
- Don’t write unintentionally nongeneric code.

Templates and Generic Programming

- Examples:
  - `templateArgumentDeduction.cpp`
  - `semiRegular.cpp`
  - `argumentDependentLookup.cpp`
  - `enable_if.cpp`
  - `isSmaller.cpp`
  - `genericArray.cpp`
  - `genericArrayInheritance.cpp`
  - `virtualNonVirtualMemberFunctiontTemplates.cpp`
  - `perfectForwarding.cpp`
  - `records.cpp`
  - `notGeneric.cpp`
  - `functionObjects.cpp`

Templates and Generic Programming

- Exercises:
  - How can the following code snippet be improved?

```
template<typename T>
    requires Incrementable<T>
T sum1(vector<T>& v, T s) {
    for (auto x : v) s += x;
    return s;
}

template<typename T>
    requires Addable<T>
T sum2(vector<T>& v, T s) {
    for (auto x : v) s = s + x;
     return s;
 }
```

Templates and Generic Programming

- The equal operator in `argumentDependentLookup.cpp` is highly visible. Fix the issue.
  - Solution: `argumentDependentLookupResolved.cpp`

- Study the instantiation process of `Array` in the programs `genericArray.cpp` and `genericArrayInheritance.cpp`. Use C++ Insights for your study.

- Extend the user-defined type `Account` (`isSmaller.cpp`) so that instances of `Account` can be compared based on the balance. Discuss the pros and cons of the various solutions.
  - Solution: `accountIsSmaller1.cpp`, `accountIsSmaller2.cpp`, `accountIsSmaller3.cpp`

Templates and Generic Programming

- Study the program `perfectForwarding.cpp` in C++ Insights.
- Refactor the program `records.cpp`. Give the predicate for the case-insensitive comparison a name and use it.
  - Solution: `records.cpp`

- Further information:
  - Templats and Generic programming