Core C++ 2019
May 14-17, 2019 :: Tel-Aviv, Israel

Tuesday May 14th - Workshop #3

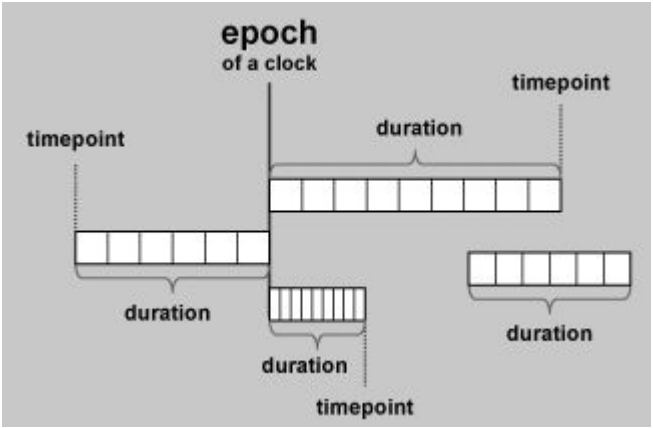
C++ Best Practices Revisited: Better Code, Better Work-Life Balance

Amir Kirsh

THE ACADEMIC of TEL AVIV-YAFFO 25 YEARS

Plan for today

9:00 - 10:30	Learn and practice
10:30 - 10:45	Short break
10:45 - 12:30	Learn and practice
12:30 - 13:30	Lunch (Economics building)
13:30 - 15:20	Learn and practice
15:20 - 15:45	Coffee Break (Economics building)
15:45 - 17:00	Learn and practice



About me

Agenda

What and Why

Classes, Types and Flow

Hierarchies

Documentation, Naming and Style

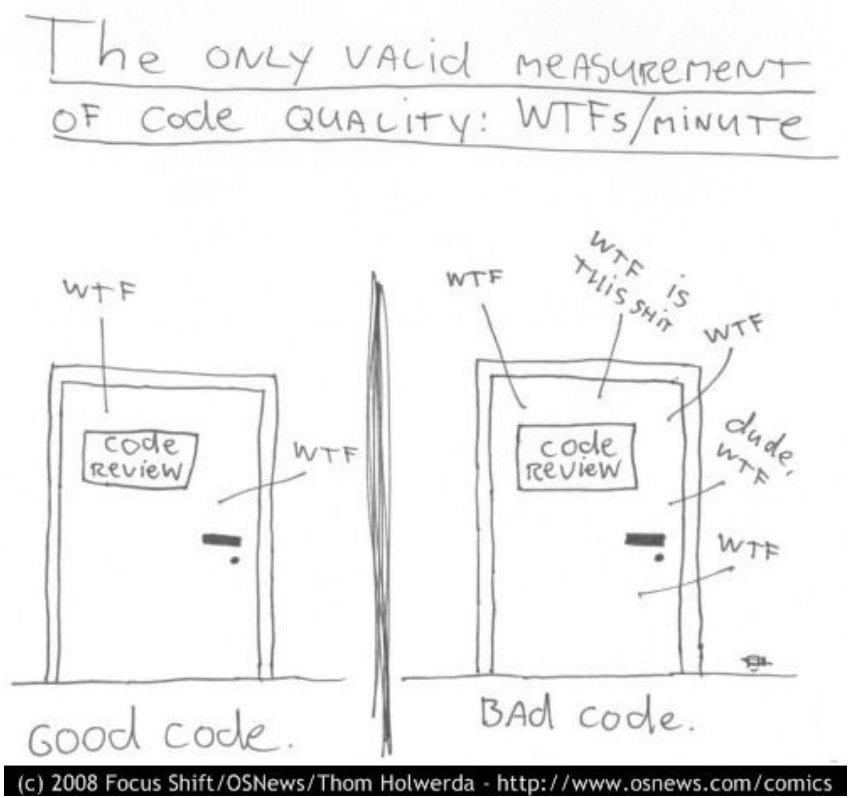
Memory

Concurrency

Templates and Lambda

Complex code and Refactoring

Our Goal


The ONLY VALid MEASUREMENT OF Code QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF is THIS SHIT
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Best Practices

Commercial or professional procedures that are accepted or prescribed as being correct or most effective.

www.dictionary.com

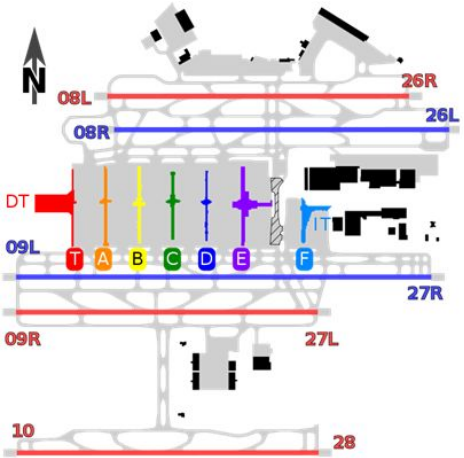
Revisited

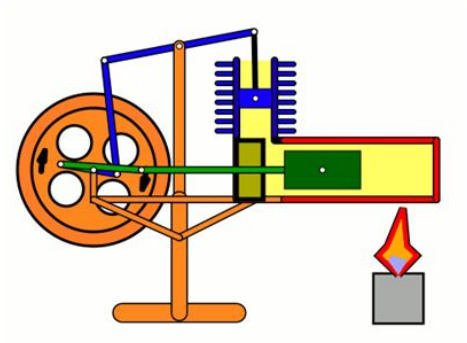
Re-examine (a topic or theme) after an interval, with a view to making a fresh appraisal

www.dictionary.com

Actual Goals and Motivation

It’s Complex!





Complicated things need:

- Clear requirements
- Documented design
- Collaborative work
- Thinking
- Being careful
- Re-evaluating
- Understanding the alternatives and their tradeoffs

Work-life balance

Better code:

- Less bugs, things are less fragile
- Code is more maintainable
- New tasks are more predictable
- No “sole ownership”
- Work can be shared
- When something does go wrong it is easier to locate and fix

This is work, not life

Work-life balance

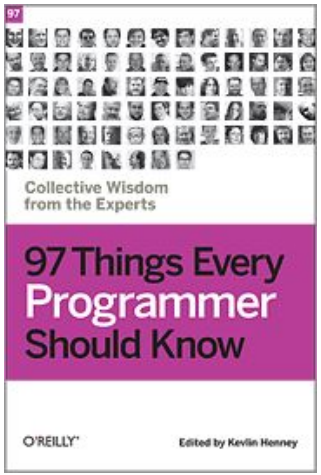
"You need to spend evenings, weekends, and holidays educating yourself, therefore you cannot spend your evenings, weekends, and holidays working overtime on your current project."

— Kevlin Henney

97 Things Every Programmer Should Know

Collective Wisdom from the Experts

http://shop.oreilly.com/product/9780596809492.do



Just before we start (1)

- **Compiler warnings:**
  always solve them, they are stronger than any best practice!
- **Static code analysis tools:**
  use them, they help you conform with best practices
- **Best practices:**
  this presentation is a partial list, keep reading and exploring!

  https://isocpp.github.io/CppCoreGuidelines
  https://isocpp.org/wiki/faq/coding-standards
  https://google.github.io/styleguide/cppguide.html

  and other (sometimes contradicting...) resources

Just before we start (2)

- **Not everything is a decisive clear cut**
  understand the rationale, trade-offs, considerations
- **Do not deviate from the rules just because of pure laziness**
  saving ***a hour*** of work is **laziness**, saving ***2 days*** is a **trade-off**
- **When you do deviate from the rules…**
  discuss it in design review / code review and document your rationale or future plans (e.g. TODO comments)
- **Some best practices or style rules are controversial**
  it's OK to agree not to agree - but at a company level it is important to conform to some uniform decisions (and decide to allow differences in some stuff)

A good read: https://eyakubovich.github.io/2018-11-27-google-cpp-style-guide-is-no-good

Just before we start (3)

We would focus mostly on C++ stuff.

But would also mention some generic programming best practices.

Just before we start (4)

All class exercises can be done in an online compiler such as:

http://coliru.stacked-crooked.com

http://ideone.com

https://wandbox.org

http://cpp.sh … or any other

We may also use: http://quick-bench.com for benchmarks

https://taas.trust-in-soft.com/tsnippet/# to pinpoint undefined behavior

https://cppinsights.io to see our code in the eyes of the compiler

and https://godbolt.org to see the generated assembler from our code

Agenda

What and Why

Classes, Types and Flow

Hierarchies

Documentation, Naming and Style

Memory

Concurrency

Templates and Lambda

Complex code and Refactoring

1. Use classes properly

Sounds funny but people still use too many (far more than appropriate):

- global functions
- structs (i.e. “all public” APIs - where not appropriate)
- classes that behave as structs (👆)

Make sure your class represents ONE THING

This is the ‘S’ in SOLID: https://en.wikipedia.org/wiki/SOLID

Don’t use pair and tuple improperly to manage your data

(See: https://arne-mertz.de/2017/03/smelly-pair-tuple)

Image Source:
https://www.connectsafely.org/one-good-thing/

SOLID

Single responsibility principle

A class should have only a single responsibility

Open–closed principle

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

Liskov substitution principle

Objects in a program should be replaceable with instances of their subtypes

Interface segregation principle

Many specific interfaces are better than one general-purpose interface

Dependency inversion principle

Code should depend upon abstractions, not concrete type

Single Responsibility Principle

👉 We've all seen this class

CppCon 2017: Victor Ciura “10 Things Junior C++ Devs Don't Get”
https://www.youtube.com/watch?v=dSSIXKe6iXE

2. Encapsulation => hide your privates

Data members should be private

if you hold *T[]* => don't expose it, provide ***get(int index)***
or: ***get(SomeEnum requiredValueCode)*** => bid, ask, low, high, …

`std::pair.first, std::pair.second` => is considered a ***language accident…***

Why? Because you cannot properly allow "different behavior"

(See a counter implementation example for pair: http://coliru.stacked-crooked.com/a/d5acb904b5df8e69)

✎ SquarePair Exercise

Create a class SquarePair that behaves like std::pair but initialized with a single number: *first* would be the number and *second* its square, but with lazy evaluation

Solution (don't peek): http://coliru.stacked-crooked.com/a/4c31320c394bcbb5

Note: above is a trick and not a best practice! (at least not replacing methods - if you need a method create a method!)

3. Rule of Zero

It is the best if your class doesn't need any resource management

- no need for dtor, copy ctor, assignment operator
- defaults do the job
- that includes defaults for move operations

To achieve that, use properly managed data members:
std::string, std containers, std::unique_ptr, std::shared_ptr

Image Source:
https://www.fluentcpp.com/2019/04/23/the-rule-of-zero-zero-constructor-zero-calorie/

4. Rule of Three

If you need a destructor, first thing block the copy ctor and assignment operator

- No TODO, no let's check if we need to implement them, BLOCK NOW

```
MyClass(const MyClass&) = delete;
MyClass& operator=(const MyClass&) = delete;
```

- If you need them later => implement

5. Rule of Five

If you implement or block any one of the five, you lose the defaults for the move operations

- Make sure to ask back for the defaults if they are fine

```
MyClass(MyClass&&) = default;
MyClass& operator=(MyClass&&) = default;
```

- Implement them if the defaults are not good enough, preferably as “noexcept”

Understand std::move

When moving, don't forget to use std::move to overcome the 'if it has a name it is an Lvalue'

Bad:

```
Person(Person&& p) : name(p.name) ...
```

Good:

```
Person(Person&& p) : name(std::move(p.name)) ...
```

Best:
Rule of Zero!

Understand std::forward

When forwarding, don't forget to use std::forward to overcome the 'if it has a name it is an Lvalue'

Bad:

```
template<typename T>
void dispatchAndLog(T&& t) { dispatch(t); log("dispatched!"); }
```

Better:

```
template<typename T>
void dispatchAndLog(T&& t) { dispatch(std::forward(t)); log("^"); }
```

Note that T&& above is not Rvalue reference, it is Universal (=Forward) Reference

Move if noexcept

There are cases where move can be used only if it promises not to throw an exception:

```
A(A&& a) noexcept {
    // code
}
```

Scenario:

- we call push_back to add a Godzilla to vector<Godzilla>
- capacity of vector is exhausted, so vector capacity shall be enlarged to allow insertion
- new bigger allocation is made, all old Godzillas shall be moved / copied to the new place
- vector is allowed to use move, to move the elements from the old location to the new one, but only if the move constructor of Godzilla is declared as noexcept, to avoid the bad case of “partial work done - there is no good vector but two broken ones...”

Read: https://en.cppreference.com/w/cpp/utility/move_if_noexcept
https://stackoverflow.com/questions/28627348/noexcept-and-copy-move-constructors

Overload resolution table

		A	B	C	D**
	Who is sent => Candidate Functions	lvalue	const lvalue	rvalue	const rvalue
1	f(X& x)	(1) ✔			
2	f(const X& x)	(2) ✔	✔	(3) ✔	(2) ✔
3	f(X&& x)			(1) ✔	
4*	f(const X&& x)			(2) ✔	(1) ✔

* Row 4 is rare - rationale for handling rvalue reference is to “steal” it.
But you cannot steal if it is a const rvalue reference, so usually you will not implement row 4.
Still, possible use case: http://stackoverflow.com/a/14742636

** Column D is also rare and mostly irrelevant - if it happens, would be usually handled by row 2 and not by row 4 (-- as row 4 is most probably not implemented).

https://stackoverflow.com/questions/47734382/c-how-does-the-compiler-decide-between-overloaded-functions-with-reference-t#47736813

6. Pass by value?

- If you may need to **change** the original - pass by ref (of course)
- If you want to **read from**
  - if copy might be expensive (or not supported) - pass by const ref
  - if copy is cheap - now and in the future - pass by value
- If you want to **copy from**
  - if move is supported - pass by value, then move
  - if move is not supported (or not known to be) - pass by const ref, then copy

https://stackoverflow.com/questions/10231349/are-the-days-of-passing-const-stdstring-as-a-parameter-over
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#fcall-parameter-passing

✎ Pass by value - Exercise

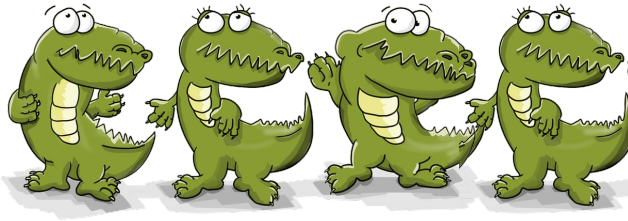
Present the potential performance gain of previous slide rule:

- If you want to **copy from**
  - if move is supported - pass by value, then move
  - if move is not supported (or not known to be) - pass by const ref, then copy

No need to create a type that actually gains from the move operation,
we assume that move ctor *may* be more efficient than copy in some cases

Solution (don't peek): http://coliru.stacked-crooked.com/a/a663b66199f9d25c

7. Copy and Swap - Rule of Four



To use the potential gain of pass by value for copy, implement a **single assignment operator** for both: copy assignment and move assignment. Get the parameter by-value and move from it:

```
MyClass& operator=(MyClass obj) noexcept {
    swap(*this, obj);    // we call here our own swap
                         // we can't rely on std::swap here...
    return *this;
}
```

Example: http://coliru.stacked-crooked.com/a/f339f9ad2dd5605f

8. Be careful with operators overloading

Make sure the operator behaves in a consistent way.

Follow the principle of least astonishment:
https://en.wikipedia.org/wiki/Principle_of_least_astonishment

post-fix vs. pre-fix

const vs. non-const

symmetry

(**discussion**: *operator*== symmetry vs. Liskov substitution principle)

You may want to view Ben Deane on Operator Overloading in CppCon 2018:
https://www.youtube.com/watch?v=zh4EgO13Etg

9. Use static properly

Easy, nothing new, usually done:

Data that is shared between all instances shall be static

Method that doesn't relate to a specific instance *may* be static

BUT -

You may prefer all methods to be non-static and virtual for easier mocking (unless you base your injection on Templates and Lambda and "duck typing")

10. Consider explicit on constructors

Constructor that do not get the entire state of the object
- should be declared as explicit

```
std::vector<int> vec = 7;  // doesn't compile, justifiably
std::vector<int> vec(7);   // compiles, justifiably
std::string str = "Hello"; // compiles, justifiably
```

11. Use const on methods and parameters

Using const correctly:

- **widens the possible usage of a function**
  and at the same time
- **protects you from indeliberate modification**
  where code should not modify.

...const_cast

Removing const (e.g. with `const_cast`) may lead to undefined behavior as the compiler assumes that a const object cannot be changed.

There are a few examples where const_cast may be safely used. Before you use it check that you are in the legitimate area and not in the undefined behavior zone.

See:
https://stackoverflow.com/questions/18841952/what-are-legitimate-uses-of-const-cast/18842082#18842082
https://stackoverflow.com/questions/2673508/correct-usages-of-const-cast

...East const vs. West const

const on the content

```
const int* ptr      <==>      int const * ptr
```

no difference, it's merely a question of style

const on the pointer

```
int * const ptr
```

...logical const vs. physical const

The compiler protects you on physical const

Preserving logical const is *on you*

```
class Foo {
    int* ptr;
public:
    // … ctor, dtor, all the gang
    int& get1() const { return *ptr; } // compiles but smelly
    void foo1() const { *ptr = 42; }   // compiles but smelly
    int*& get2() const { return ptr; } // doesn't compile
    void foo2() const { ++ptr; }       // doesn't compi
};
```

don't do that, remove the const qualifier from the method, or the method itself

...const iterators and const smart pointers

Note that iterators and smart pointers can also be const.

Use it correctly!

```
void printNumbers(const list<int>& numbers) {
  list<int>::const_iterator itr = numbers.cbegin();
  while( itr != numbers.end() ) {
    std::cout << *itr++ << ' ';
  }
}
```

* const smart pointers would be presented later

12. constexpr

Use constexpr for constants that are assigned with a value in compile time

- efficiency
- correctness

Note that functions and constructors can also be marked as constexpr

C++17 also adds ‘if constexpr’ as a replacement for SFINAE
and preprocessor **`ifdef/ifndef`** directives

13. auto

http://stackoverflow.com/questions/6434971/how-much-is-too-much-with-c11-auto-keyword

google style guide on auto:
https://google.github.io/styleguide/cppguide.html#auto - **use only for complex types**

the “big shots” on auto:
https://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Scott-Andrei-and-Herb-Ask-Us-Anything#time=25m03s - **use practically always**

(also discussed in Effective Modern C++ / Scott Meyers - Item 6)

✎ auto - Exercise

Show that ‘auto’ may have efficiency effect by deducing the correct type of pair when running on a map, compared to a common bad manually selected type.

Solution (don’t peek): http://coliru.stacked-crooked.com/a/19731b4611ac2a57

14. Beware of your return type!

What can go wrong with this code?

```
template<class Map, typename Key>
const typename Map::mapped_type& get_or_default(
    const Map& map,
    const Key& key,
    const typename Map::mapped_type& defaultVal
) {
    auto pos = map.find(key);
    return (pos != map.end() ?
            pos->second: defaultVal);
}
```

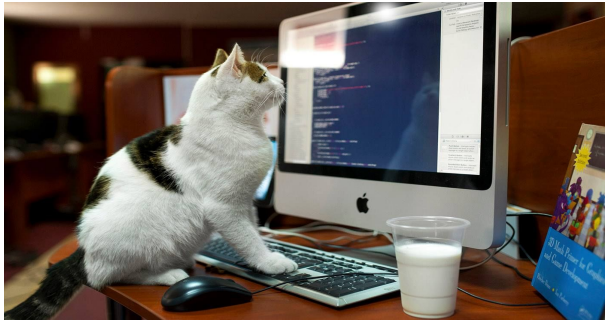


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

...Beware of your return type!

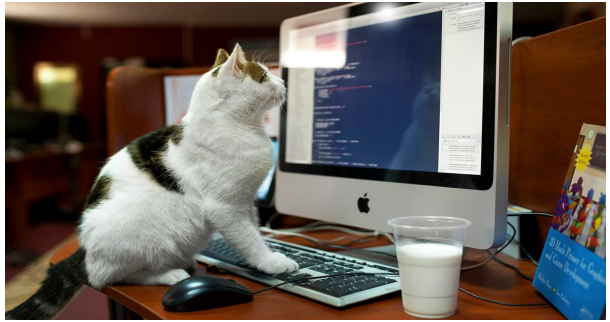


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

What's wrong?

```
template<class Map, typename Key>
const typename Map::mapped_type& get_or_default(
    const Map& map,
    const Key& key,
    const typename Map::mapped_type& defaultVal
) {
    auto pos = map.find(key);
    return (pos != map.end() ?
            pos->second: defaultVal);
}

const string& str = get_or_default(mymap, "pikotaro", "pineapple");
std::cout << str;
```

...Beware of your return type!

Code presenting the problem:
http://coliru.stacked-crooked.com/a/e7983b00ebb59520

We can compile the code with ASAN sanitize flag:
https://github.com/google/sanitizers/wiki/AddressSanitizer
-fsanitize=address

This locates the problem right ahead!
http://coliru.stacked-crooked.com/a/74d5b2e2d0876226

And now the problem is fixed!
http://coliru.stacked-crooked.com/a/d6c8516fe362aeae

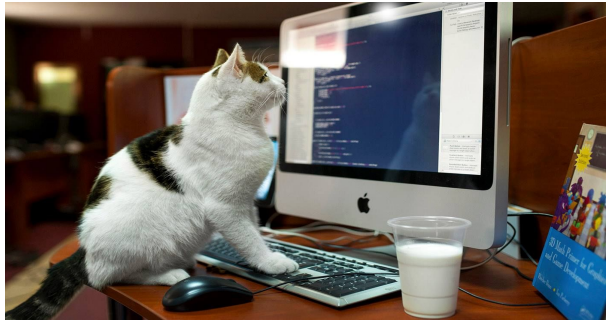


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

...Beware of your return type!

Someone may try to fix it back to const&...

Add documentation note!

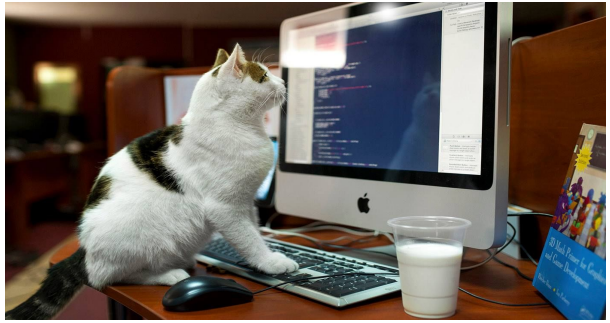


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

```
// we return by value in purpose as returning const reference
// might be a const reference to a temporary which is a bug
// (don't believe it? see: https://www.youtube.com/watch?v=lkgszkPnV8g&t=14m35s)
template<class Map, typename Key>
typename Map::mapped_type get_or_default(
    const Map& map,
    const Key& key,
    const typename Map::mapped_type& defaultVal
) {
  ...
```

by value

CppCon 2017: Louis Brandy
"Curiously Recurring C++ Bugs at Facebook"

...Beware of your return type!

Can we keep it as const& and be safe?

Is there a way??

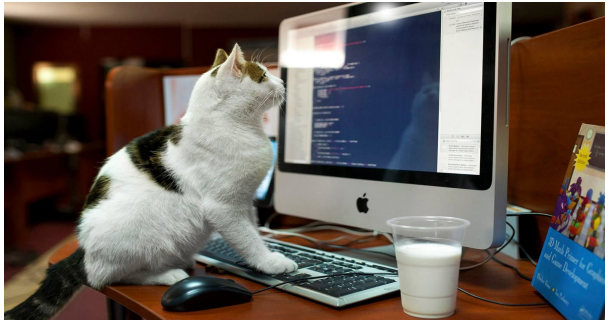


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

```
// we return by value in purpose as returning const reference
// might be a const reference to a temporary which is a bug
// (don't believe it? see: https://www.youtube.com/watch?v=lkgszkPnV8g&t=14m35s)
template<class Map, typename Key>
typename Map::mapped_type get_or_default(
    const Map& map,
    const Key& key,
    const typename Map::mapped_type& defaultVal
) {
  ...
```

by value

...Beware of your return type!

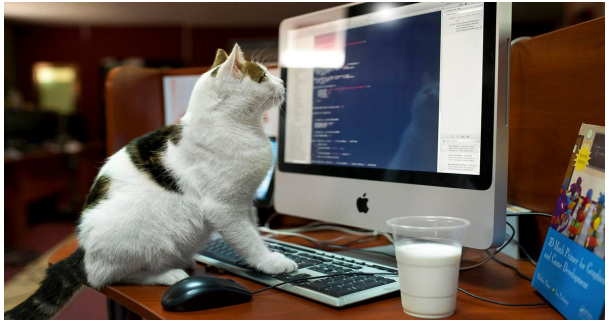


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

Yes, there a way!

```
template<class Map, typename Key>
const typename Map::mapped_type& get_or_default(...)
```

```
// add this overload
// don't allow temporary (rvalue) defaultVal
template<class Map, typename Key>
const typename Map::mapped_type& get_or_default(
    const Map& map,
    const Key& key,
    typename Map::mapped_type&& defaultVal
) = delete;
```

http://coliru.stacked-crooked.com/a/0a9bcbac92b5a891

15. Use forward declaration instead of include

Wherever possible prefer forward declaration over include:

- reduces compilation time
- less compilation issues, trying later to break circular dependencies

Notes:

- google C++ style guide says the opposite, ignore it :-)
- if you get compilation error on ‘incomplete type’ - you may either need an include or need to move code from header to cpp
- beware of multiple inheritance casting on incomplete type!!!
  (if you always use C++ cast operators properly you are fine).

  Code example presenting the problem: http://coliru.stacked-crooked.com/a/e285fb4d59a3754c

16. Avoid ‘using’ entire namespaces

```
using namespace std;    // <= BAD - namespace pollution

using std::cout;        // <= Better

std::cout << "hello"; // <= full name in actual usage, also OK or even better
```

https://stackoverflow.com/questions/1452721/why-is-using-namespace-std-considered-bad-practice

Above rule applies to *your* namespaces as well

17. Types and Type Aliases

https://en.wikipedia.org/wiki/Mars_Climate_Orbiter#Cause_of_failure

Google style guide on type aliases - there are pros and cons:
https://google.github.io/styleguide/cppguide.html#Aliases

Is type aliases an actual solution? not really...

```
using meteres = double;
// caller
meters distance_in_meters = 7.5;
doSomething(distance_in_meters);
```

```
// the method that we call
void doSomething(float distance) {
  // we assume distance is in inches
}
```

...UDL (user defined literals)

Chrono is a great example for type literals:

https://en.cppreference.com/w/cpp/header/chrono

But you can define your own:

```
Length length = 12.0_km + 120.0_m;
```

http://coliru.stacked-crooked.com/a/050d20cbbdccbcc2

See also:

https://en.cppreference.com/w/cpp/language/user_literal

https://akrzemi1.wordpress.com/2012/08/12/user-defined-literals-part-i/

https://stackoverflow.com/questions/237804/what-new-capabilities-do-user-defined-literals-add-to-c

...Fluent

Consider using Fluent for Strong Types

https://github.com/joboccara/NamedType

```
using Meter = NamedType<double, struct MeterParameter>;

using Width = NamedType<Meter, struct WidthParameter>;
using Height = NamedType<Meter, struct HeightParameter>;

Meter operator"" _meter(unsigned long long length) {
  return Meter(length);
}

Rectangle r(Width(10_meter), Height(12_meter));
```

18. Avoid Undefined Behavior

https://memegenerator.net/instance/63896485/spongebob-rainbow-undefined-behavior

Undefined Behavior of overflow

The exact behavior on overflow / underflow is only specified for unsigned types.
For signed integer types the C++ standard simply says that anything can happen.

✎ Exercise: prove the problem of undefined behavior for signed integer overflow

Solution (don't peek):
Undefined behavior (gcc): https://coliru.stacked-crooked.com/a/01daf1f23ef832a1
Undefined behavior (clang): https://coliru.stacked-crooked.com/a/e02aa734ce68aaad
Undefined behavior analysis: https://taas.trust-in-soft.com/tsnippet/t/76626d2a
https://taas.trust-in-soft.com/tsnippet/t/689e4f65

More on signed vs. unsigned, overflow and undefined behavior

http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

boost::numeric_cast: https://www.boost.org/doc/libs/1_70_0/libs/numeric/conversion/doc/html/index.html

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1879.htm

https://www.us-cert.gov/bsi/articles/knowledge/coding-practices/safe-integer-operations

https://stackoverflow.com/questions/30371505/add-integers-safely-and-prove-the-safety

https://www.jwwalker.com/pages/safe-compare.html

http://soundsoftware.ac.uk/c-pitfall-unsigned.html

https://stackoverflow.com/questions/22587451/c-c-use-of-int-or-unsigned-int

https://stackoverflow.com/questions/7488837/why-is-int-rather-than-unsigned-int-used-for-c-and-c-for-loops

https://stackoverflow.com/questions/199333/how-do-i-detect-unsigned-integer-multiply-overflow

https://stackoverflow.com/questions/10011372/c-underflow-and-overflow

https://www.google.com/search?q=cppcon+undefined+behavior ⇐ a popular topic in CppCon

19. Error Handling

Develop an error-handling strategy early in a design
(https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#e1-develop-an-error-handling-strategy-early-in-a-design)

If you decide not to use exceptions (as the decision made by google) - Note that some of the libraries you use, including the standard library, may throw exceptions.

See: https://stackoverflow.com/questions/5184115/google-c-style-guides-no-exceptions-rule-stl

...Avoid using exceptions?

Is it really that expensive to use exceptions?

Benchmark:

```
With exceptions, nice data    :  (1000000,0) in 1553us.
With exceptions, nasty data   :  (0,1000000) in 3677976us.
With optional, nice data      :  (1000000,0) in 2031us.
With optional, nasty data     :  (0,1000000) in 3696us.
```

https://wandbox.org/permlink/48ZQhDSRlZssK6uK

In: CppCon 2018: Patrice Roy “Pessimistic Programming”

20. RAII

Resource Acquisition is Initialization

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-raii

Local object

- cannot escape being born via a ctor
- cannot escape being destroyed via the dtor

Use ctor and dtor to manage the lifecycle of local objects

This is the best tool for resource management

21. User Input and Validation

Never trust user input

Never trust input that might be altered or
processed externally before arriving to your code

All paths that rely on user input must be reviewed (including “dead code”)

Protective Programming - trusted input may yet be revalidated:

- ➔ protect your code against bugs (‘this should never happen’, well unless a bug...)*
- ➔ traceability - logging**

* though, consider performance ** do not revalidate to *repeat* error reporting in logs

...Protective Programming Example

```
if(isInRange(num, allowedMin, allowedMax)) {
    // happy path
    // do things
}
else {
    // bizarre, unexpected:
    // throw an exception / return error code
    // and or print to log (info, warn, error)
}
```

Consider having this check just for protection, even if not required for the flow (‘this should never happen’ cases).
When you already have the ‘if’ as part of your logic, make sure not to forget the ‘else’ (‘this may happen’ cases).

22. Use compile time assertions

Validate when necessary expected constexpr values (size of array, other constant values), types in macro and template expansions

Exercise: improve the code below to use static_assert:

```
// note: do not change the values below!
// FooWidgetFlags values must conform to IEEE spec 9927331
enum class FooWidgetFlags {NORTH = 100, SOUTH = 2000};
```

Solution (don't peek): http://coliru.stacked-crooked.com/a/5941092bed68ba95

See also: https://en.cppreference.com/w/cpp/language/static_assert

23. Security

It’s a topic by its own

You should NOT ignore it.

Make sure one of your team members reads this excellent free book:
https://dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html

Agenda

What and Why

Classes, Types and Flow

Hierarchies

Documentation, Naming and Style

Memory

Concurrency

Templates and Lambda

Complex code and Refactoring

24. Inheritance is “overrated” - don’t rush for it

Postpone your inheritance to the point it is actually required:

WorkerByHour and **MonthlyPaidWorker** *are the same*

They are BOTH just a **Worker**
(and not only for social reasons)

You may add a field in worker for the different behavior (state / strategy pattern)

Advantages:

- No need to kill and recreate object when a worker moves from one status to another
- Can use concrete object Worker (while polymorphism requires pointers)
- Reduced complexity: consider all different attributes of a worker

...before inheriting - ask yourself

Is the relation “is a” or “has a” - **maybe it’s composition**?

What do we model here, is it actually a different thing?

if it has only some different behavior in some specific aspects use **strategy / state**

Can we model the different behavior with **template** parameters?

Advantage: performance, compile time check while still very generic (“duck typing”)
Example (“Static Polymorphism”):

```
NetworkConnection<typename Protocol, typename ServerClientTag>
```

OOP without Inheritance / Bjarne Stroustrup, ECOOP 2015:
https://www.youtube.com/watch?v=xcpSLRpOMJM

25. Prefer to avoid *protected* data members

Using *protected* accessibility breaks the rule of managing data in one place (https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-protected)

To manage different validation on data members or any other logical difference between base and derived, use:

- Virtual functions
- Policy based design / Strategy design pattern

26. Use interfaces

The fact that C++ doesn't have interfaces should not stop you from creating them

Interface in C++ would be an abstract class with no data and usually without a constructor (but with an empty virtual destructor).

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-abstract

Real life: in many cases we do have abstract classes with actual data serve as interfaces and this usually can work (e.g. base class for Shape)

27. Beware of object slicing

```
// Usually Slicing is an accident and not what you meant

class Base { int x, y; };

class Derived : public Base { int z, w; };


int main() {
  Derived d;
  Base b = d; // Object Slicing, z and w of d are sliced off
}
```

28. Make non-leaf classes abstract

The known advice by Scott Meyers, item 33 in ”More Effective C++”.

To put the same thing differently: **do not inherit from concrete classes**

Why?

Remember that a class should have single responsibility? (the ‘S’ in SOLID)

Being a *Base* while also being *concrete class* serves two purposes which may contradict, thus hurting your design and making it harder to later modify this class.

29. Avoid virtual functions if not needed

There is a reason C++ decided not to have dynamic linking as a default.

Virtual functions are costly:

- **runtime 2 jumps instead of 1 / 0**
- **code cache miss**
- **memory requirements (usually not an issue)**

See: https://stackoverflow.com/questions/449827/virtual-functions-and-performance-c

In some cases dynamic polymorphism may be replaced with static polymorphism

(Would be discussed later under “Templates”)

30. virtual destructor

Polymorphic hierarchies MUST have a virtual destructor at the top!

(Even if no class in the hierarchy needs a destructor at all!!)

✎ Exercise: prove the note above 👆

Base class destructor should be either:
- public and virtual (for polymorphic hierarchy), or
- protected and nonvirtual (for non-polymorphic hierarchy)

(Herb Sutter - http://www.gotw.ca/publications/mill18.htm)

...virtual destructor in real life

In real life you may *also* encounter the following cases:

1. **no user destructor declared** - there is a default non-virtual destructor
   this class is not meant for polymorphic usage, you may want to document it
2. **public non-virtual destructor** - not following Herb Sutter's advice
   this class is not meant for polymorphic usage, you may want to document it
   actual examples: std::string, std containers
3. **private dtor** - use where all objects are managed by smart pointer with a custom deleter.
   Possibly for: singleton, objects in object pool, see for example:
   https://stackoverflow.com/questions/19274058/singleton-with-private-destructor-using-stdunique-ptr

(Then come the guys from Herb Sutter's rule, the good and "should be more common" guys:)

4. **public virtual dtor** - allows polymorphic inheritance
5. **protected non-virtual dtor** - *enforced* non-polymorphic

...virtual destructor - a simple rule

If your class has declared another virtual function ⇒

You should have a virtual dtor

If no other virtual function in class and you still hesitate, read this:
https://softwareengineering.stackexchange.com/questions/284561/when-not-to-use-virtual-destructors

31. Use public inheritance just for polymorphism

If polymorphism is not required, i.e. no virtual functions in base - inherit privately

Do not inherit publicly from std containers and std::string

Why?

to make sure no one would point to derived with base pointer, as we do not have a virtual dtor.

(There is a reason C++ has private inheritance and Java doesn't => in C++ we may have classes without a virtual table).

...public inheritance without polymorphism

In practice we do see public inheritance in non-polymorphic hierarchies.

When you have such a case:

- document that the inheritance is not polymorphic (i.e. no virtual dtor in base)
- consider blocking new operators for this class
  see:
  https://en.wikibooks.org/wiki/More_C++_Idioms/Requiring_or_Prohibiting_Heap-based_Objects
  https://stackoverflow.com/questions/1941517/explicitly-disallow-heap-allocation-in-c

("Free delegation" of base methods, achieved in public inheritance, is not a good reason for public inheritance, as you can easily achieve delegation with generated code created automatically by most modern IDEs, or do that manually).

32. Do not call virtual functions from ctor or dtor

Calling virtual functions from ctor or dtor is not polymorphic

We would reach the method at the level that is currently being created (Base)

This may be confusing, and may even lead to runtime errors in case the method in the base is pure virtual.

Don't overcome this by calling a non-virtual method that would call the virtual method, it doesn't help…

Instead:

Create the method and then call virtual 'init' on it

Or wrap your hierarchy with a managing wrapper that would do that

33. Use virtual or override [or final]

virtual = a new virtual method

A new virtual method should have the **virtual** keyword

override and final =
an implementation of an already declared virtual method

An implemented virtual method that was declared virtual in the base
should *NOT have* the virtual keyword but *should have* the **override** keyword

(In rare cases where the method should not be changed in further derived classes, put only the final keyword)

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-override

34. Avoid default parameters in override function

Default parameters are a compile time thing

While virtual functions linking is dynamic

Bad, confusing behavior:

```
struct Base {
    virtual void foo(int i = 0);
};

struct Derived: Base {
    void foo(int i = -1) override;
};
```

Better:

```
class Base {
    virtual void fooImpl(int i);
public:
    void foo(int i = 0) {fooImpl(i);}
};

class Derived: public Base {
    void fooImpl(int i) override;
};
```

35. Consider making virtual functions private

There are arguments in favor

Herb Sutter: http://www.gotw.ca/publications/mill18.htm

The C++ FAQ is reluctant

https://isocpp.org/wiki/faq/strange-inheritance#private-virtuals

It is mostly relevant when you want to hide your polymorphic behavior from the user and make it something more internal to the class hierarchy.

36. Avoid overloading in polymorphic hierarchy

Non-virtual methods shall be used for generic algorithms.

Overloading methods in polymorphic hierarchy is confusing.

Bad, confusing behavior:

```
struct Base {
  virtual ~Base() {}
  void foo() const;
};

struct Derived: Base {
  void foo() const;
};
```

```
void doSomething(const Base& b) {
  b.foo();
}

int main() {
  auto pb = std::make_unique<Derived>();
  pb->foo(); // to which foo would it go?
  doSomething(*pb); // and this one?
}
```

https://coliru.stacked-crooked.com/a/cf433f7795dedee2

37. Beware of inheritance hiding rules

Hiding rules are cruel, but justified. Just be aware!

Bad, confusing behavior:

```
struct Base {
  void foo(double d);
};

struct Derived: Base {
  void foo(int i);
};
```

```
int main() {
  Derived d;
  double num = 2.5;
  d.foo(num); // which foo do we call here?
}
```

https://coliru.stacked-crooked.com/a/cc155516558fea3f

The rationale behind the cruel rule: https://stackoverflow.com/questions/4837399/c-rationale-behind-hiding-rule

Agenda

What and Why

Classes, Types and Flow

Hierarchies

Documentation, Naming and Style

Memory

Concurrency

Templates and Lambda

Complex code and Refactoring

38. Good documentation

Code Tells You How, Comments Tell You Why

Jeff Atwood (co-founder of stackoverflow)

https://blog.codinghorror.com/code-tells-you-how-comments-tell-you-why/
https://blog.codinghorror.com/coding-without-comments/

Why:

- why this loop must come before the next one, even though it seems odd
- why we didn’t do here the obvious thing (that doesn’t really work, we tried...)

...Bad documentation - avoid the obvious

There is a famously bad comment style:

```
i=i+1;                    /* Add one to i */
```

Rob Pike (one of the creators of the “Go” language)
https://www.lysator.liu.se/c/pikestyle.html

but there are worse ways to do it:

```
/**********************************
*                                 *
*          Add one to i           *
*                                 *
**********************************/

i=i+1;
```

Rob Pike (one of the creators of the “Go” language)
https://www.lysator.liu.se/c/pikestyle.html

Don't laugh now, wait until you see it in real life.

Rob Pike (one of the creators of the “Go” language)
https://www.lysator.liu.se/c/pikestyle.html

...Good documentation - TODO comments

```
// TODO: need to complete this slide
// do that before 14-05-2019 or delete this comment
// @Amir Kirsh, 13-05-2019
```

Next slide contains explicit and unpleasant views

Be warned

...Commented code is NOT documentation

- **Commented code is noise**
- **It is silently getting rot**
- **Someone may accidentally use it**

```
// ====================
// CODE-FOR-DELETION
// TODO: code below is not relevant anymore
// can be deleted if not required by 14-Nov-2019
// @Artemy Vysotsky 14-05-2019
//    … <some code> ...
// END-OF-CODE-FOR-DELETION
// ====================
```

39. Have proper file header documentation

Including @author notes on file, methods and changed code

This contradicts google style guide advice
https://google.github.io/styleguide/cppguide.html#File_Comments

Why?

Because in many cases the contributor listed in source control is not the actual contributor (private merge and integrations from other branch)

+ of course

Proper class description above all classes!

40. Document code points that may be overseen

Empty loop - it takes a moment to see the semicolon at the end

```
for ( ; *itr != look_for; ++itr);
```

Better with documentation:

```
// empty loop, moving itr to the item we look for or to the end
for ( ; *itr != look_for; ++itr);
```

Or:

```
for ( ; *itr != look_for; ++itr) {
    // do nothing, we just move itr to 'look_for' position or to the end
}
```

41. Naming

“There are only two hard things in Computer Science: cache invalidation and naming things” — PHIL KARLTON

Let’s just focus on one thing: **method names shall say what they do**:

```
if(checkDate(person))           -- not so good
if(validateBirthDate(person))  -- better
if(hasValidBirthDate(person))  -- even better
```

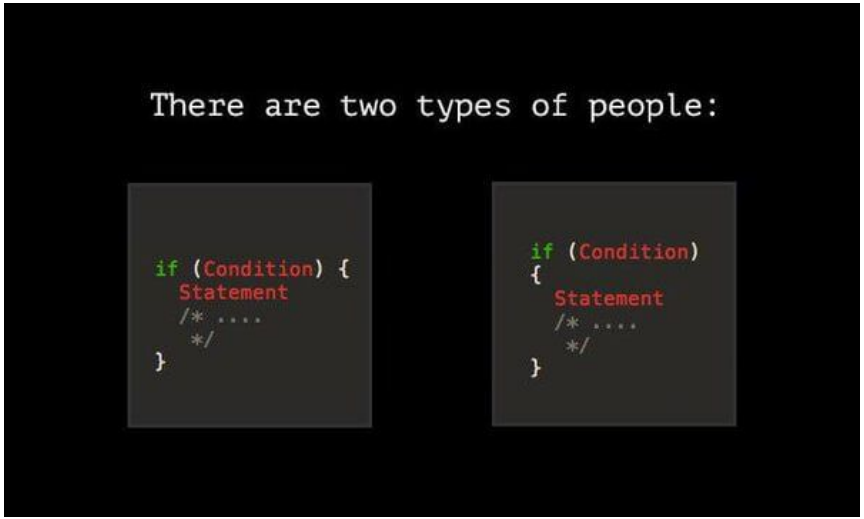
42. Style

Don’t be hard when not required

{ Don’t fight over curly brackets position }

Don’t fight over space after ‘if’

if (test) == if(test)



Don’t change style when you edit existing code

→ extra unnecessary noise in source control

→ your colleagues will pay you back on your code...

...Spaces or Tabs

Why should it matter?

- indentation
- source control changes

Google C++ style guide:

use spaces, make your IDE replace tab with 2 spaces

https://google.github.io/styleguide/cppguide.html#Spaces_vs._Tabs

Why else should it matter?

Developers Who Use Spaces

Make More Money Than Those Who Use Tabs

https://stackoverflow.blog/2017/06/15/developers-use-spaces-make-money-use-tabs/

...East or West const?

as long as you know that:

```
const char* s;
```

is the same as:

```
char const * s;
```

but different from:

```
char * const s;
```

and you remember who is who - you are fine!

Some companies allow both East and West, some conform to one of them.

There are pros and cons for each.

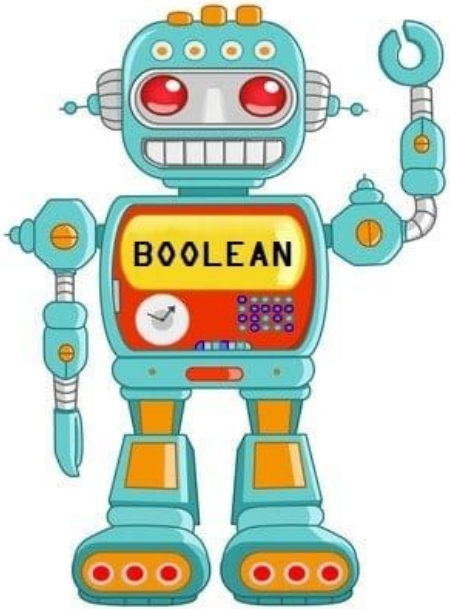
...Simplify boolean!

boolean expressions shall be expressions, not if-else

```
return (a==b && i > 10);
```

is better than

```
if (a==b && i > 10)
    return true;
return false;
```



ternary operator: `test? true-value : false-value` -- is OK when simple and returns exactly the same type (no hidden casting), otherwise use if-else

Agenda

What and Why

Classes, Types and Flow

Hierarchies

Documentation, Naming and Style

Memory

Concurrency

Templates and Lambda

Complex code and Refactoring

43. Use Smart Pointers

Use unique_ptr, shared_ptr and weak_ptr to manage pointers ownership

Do not transfer ownership based on raw pointer or reference

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#i11-never-transfer-ownership-by-a-raw-pointer-t-or-reference-t

- C++ smart pointers are cheap - use them
- You get RAII which if you remember is a good thing
- It's the accepted industry standard, anywhere, by anyone who uses C++

On the performance of C++ smart pointers:

http://blog.davidecoppola.com/2016/10/performance-of-raw-pointers-vs-smart-pointers-in-cpp/

https://stackoverflow.com/questions/22295665/how-much-is-the-overhead-of-smart-pointers-compared-to-normal-pointers-in-c

...What's so bad with raw pointers?

- is it a pointer to a single object or to array of objects?
- who owns it?
- is it set already? was it released? can I use it?
- how can I make sure only one holder will release it?
- in long twisted code, how can I make sure all paths release the pointer? (who guarantees RAII for my pointers?)
  - forgetting to release in a certain branch => memory leak
  - releasing more than once => undefined behavior, usually a crash

...Use unique_ptr for single ownership

unique_ptr is moveable and not copyable thus allowing single ownership

- it is cheaper than shared_ptr - so if there is no need for sharing, prefer unique_ptr
- unique_ptr can turn to shared_ptr (ctor of shared_ptr accepts unique_ptr)

Prefer to use std::make_unique over direct allocation

```
// better (since C++14)
auto a = std::make_unique<A>();
```

```
// less good
auto a = unique_ptr<A>{new A{}};
```

Herb Sutter with a case in which make_unique saves from a leak: https://herbsutter.com/gotw/_102/

...Use shared_ptr for shared ownership

shared_ptr is moveable and copyable thus allowing shared ownership

- it is more expensive than unique_ptr - but if you need it use it
- shared_ptr *cannot* turn into unique_ptr

Prefer to use std::make_shared over direct allocation

```
// better, almost always
auto a = std::make_shared<A>();
```

```
// usually less good
auto a = shared_ptr<A>{new A{}};
```

The left is more efficient. There is one rare case in which you may consider the right - to be discussed later.

...Use weak_ptr for “weak ownership”

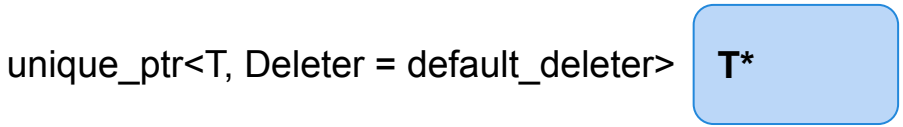
weak_ptr doesn’t increase the ref count of the shared_ptr thus when you try to get the owned object it might already be released, but no harm is done

- useful for sharing a resource (e.g. between threads) where the actual owner may release the resource
- once a weak_ptr was safely turned into shared, by calling lock, it is owned safely as shared_ptr and *cannot* expire while in use:

```
auto s1 = std::make_shared<A>();
auto weak = std::weak_ptr(s1);
auto s2 = weak.lock(); // returns shared_ptr which may hold nullptr
```

...unique_ptr - what's inside

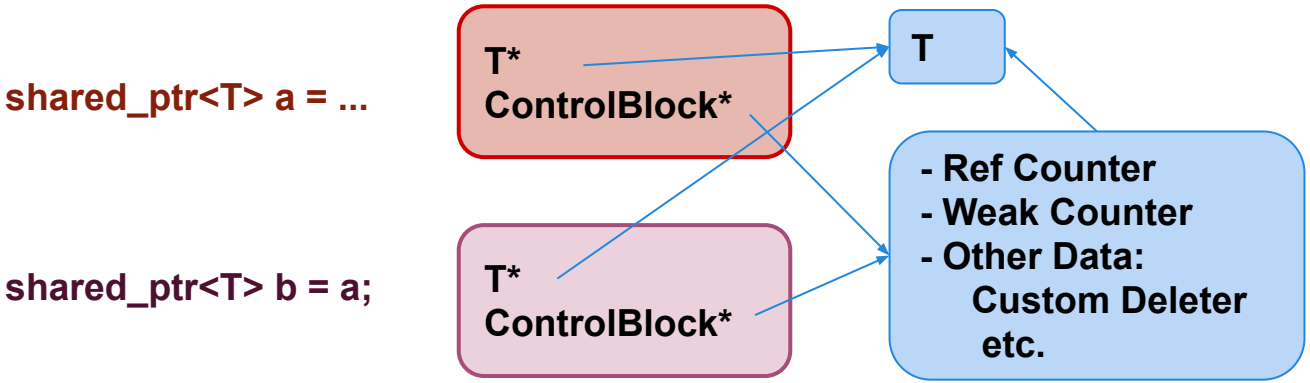
unique_ptr with default deleter holds ONLY the pointer ⇒ size = size of pointer



- **In case of Pimpl (holding unique_ptr to incomplete type):**
  - **Must declare all special member functions - ctors/dtors/assignment - in header and implement them in cpp even if not required at all! - See Item 22 of Modern Effective C++**

On the size of unique_ptr with custom deleter + a trick to reduce this extra size to zero, see: http://www.bourez.be/?p=19

...shared_ptr - what's inside



Control Block

- Manages the ref counter
- Holds the deleter
- Control block actions are thread safe (counting, deleting, getting weak_ptr)
  - Note: **actions on T are not thread safe!** (depends on T itself)

44. Avoid using ‘new’ and ‘delete’ in your code

Once you start using smart pointers, together with *make_unique* and *make_shared* there is no need anymore to use directly ‘new’ and ‘delete’!

The above rule has almost no exceptions, see:
https://stackoverflow.com/questions/46991224/are-there-any-valid-use-cases-to-use-new-and-delete-raw-pointers-or-c-style-arr

- You get immediate RAII which if you remember is a good thing
- You avoid the asymmetry of having ‘new’ into a smart pointer, without a matching ‘delete’

Note: you may still use raw pointers, when relevant, this rule doesn’t prohibit that

...can still break code with smart pointer, beware

Programmer can still cause runtime damages:

- pass the same raw pointer to two different smart pointers (BAD!)
- get the raw pointer from a smart pointer and delete it (BAD!)
- release the pointer from the smart pointer and keep using it (BAD!)

Don’t do the above!

Bad code examples with unique_ptr: http://coliru.stacked-crooked.com/a/ee80f109ee8aaddb

Same bad code, with shared_ptr: http://coliru.stacked-crooked.com/a/4a3a18f2b715fe8e

45. Carefully design your API with smart pointers

Considerations on how to design your internal API with smart pointers is delicate, make sure to use the correct signatures.

- API design should consider the life time of objects and in some cases the threading model
- The actual usage should be aware of the way smart pointers work

See next slides...

API design for smart pointers (1)

Usage	**Proposed API**	**Note**
Factory	unique_ptr<T> factory(params...);	can later be used as shared
Sink **(gets pointer and owns it)**	void sink(unique_ptr<T> ptr);	would move into
Share with someone	void process(shared_ptr<T> ptr); void monitor(weak_ptr<T> ptr);	

Share back	shared_ptr<T> getWidget();
	shared_ptr<const T> getWidget() const;
	weak_ptr<T> getWeakPtrToWidget();
		You may want to share back the content as const
		>> See an important note on rvalue shared_ptr later on <<

API design for smart pointers (2)

Usage	**Proposed API**	**Note**
Let method change our smart_pointer (output param)	void modify(unique_ptr<T>& ptr); void modify(shared_ptr<T>& ptr);	allow the method to change the pointer *that we hold* - be very careful with that!

Work on our resource without actually sharing (we know that we are still alive)	void process(T& ptr);
	void process(T* ptr); // if null is allowed
	void process(const T& ptr);
		better than the alternatives:
		void process(shared_ptr<T>& ptr);
		-- or -- (const shared_ptr<T>& ptr);

Share back
as lvalue, as const lvalue
or as a copy
	T& getWidget(); // prefer not
	const T& getWidget();
	T getWidget();
		Sharing copy back has no issues except for copying
		Sharing reference requires the object shared back to still be alive

See: https://herbsutter.com/2013/05/30/gotw-90-solution-factories | https://herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters
https://stackoverflow.com/questions/3310737/should-we-pass-a-shared-ptr-by-reference-or-by-value

...const smart pointers

To protect the *content* of the smart pointer, the 'const' should be on the type:

```
void foo1(shared_ptr<const A> ptra); // the content is const, cannot change
void foo2(shared_ptr<A> ptra); // may change the inner value

int main() {
    shared_ptr<A> ptr = make_shared<A>(3);
    foo1(ptr); // ok!
    foo2(ptr); // ok (foo2 takes non-const A)
    shared_ptr<const A> const_ptr = make_shared<A>(13);
    foo1(const_ptr); // ok!
    // foo2(const_ptr); // error (foo2 takes only non-const A)
}
```

Code: http://coliru.stacked-crooked.com/a/b97b53c9db7ece98

...sharing from this

Class that wishes to return this pointer as shared_ptr needs to be cautious

This is BAD:

```
struct Bad {
    shared_ptr<Bad> get_ptr() {
        return shared_ptr<Bad>(this); // THIS IS BAD
    }
};
```

The better way:

http://en.cppreference.com/w/cpp/memory/enable_shared_from_this

See a nice example: http://aldrin.co/esft.html

A possible alternative: https://stackoverflow.com/questions/38238876/enable-shared-from-this-vs-direct-assignment/38244552#38244552

...note: *rvalue* shared_ptr is bug prone, beware

Dereferencing shared_ptr returned by value,
without taking it into a local shared_ptr variable:

```
auto& ref = *returns_a_shared_ptr();
ref.boom(); // ref may be dead here
            // not managed anymore by the shared_ptr
```

Source - CppCon 2017: Louis Brandy “Curiously Recurring C++ Bugs at Facebook”:
https://www.youtube.com/watch?v=lkgszkPnV8g&t=28m30s

But this is OK:

```
returns_a_shared_ptr()->boom(); // this is OK, still alive
```

...note: cyclic reference of shared_ptrs, beware

```
class A {
    shared_ptr<B> pb;
    // ...
};
```

```
class B {
    shared_ptr<A> pa;
    // ...
};
```

Cyclic references would never be released…

It may happen also with a single class holding self reference as shared_ptr (e.g. Person holding spouse)

Solution: use weak_ptr

Code example of cyclic shared_ptr reference: https://coliru.stacked-crooked.com/a/0bdb6587db374fa7

Agenda

What and Why

Classes, Types and Flow

Hierarchies

Documentation, Naming and Style

Memory

Concurrency

Templates and Lambda

Complex code and Refactoring

46. Have a clear threading model

Your concurrency design shall be led by a clear threading model:

- Tasks in a Queue with a thread pool
- Queue per thread with messages between threads
- Producer-Consumer with multiple or single threads
- Fork-Join, Map-Reduce
- Stickiness of Session / User / Action to hosting thread

47. Most of your code shouldn't be *'thread aware'*

Given your threading model, parts of the code, can be unaware of concurrency issues (in a good way)*

- An isolated task running in a single thread
- A producer generating tasks into a queue or firing messages to other threads
- A joining task, waiting for results from other threads, but when ready runs on stable data owns by itself alone

* or be almost unaware

48. Object Model, Threading Model, Flyweight

The closer your object model is to your threading model, the easier it would be to manage your concurrent code.

- If an object is being born and dies only in a single thread and no other thread knows this object - concurrency is easy
  Note that it doesn't mean "thread per session" - same thread may handle many sessions and still each session may live only in a single thread
- Constant data that is shared between threads is not an issue
- Try to separate between shared data and self data, write code that works only on self data (and reads constant shared data or a snapshot copy)

...Stateless is precious and priceless

It is easier to manage concurrency for a stateless process, let the state traverse on the data that travels between services.

e.g. Microservice architecture

It may seem a bit counter OOP - pushing for classes that do things but do not manage the data that they process. On the other hand the big gain is dynamic elasticity.

...Actor Model

Actor Model - messages travel through actors, actors may change their own state and send messages to others, but cannot *directly* change the state of another actor - thus each actor only modifies its own data and can run in its own thread.

https://en.wikipedia.org/wiki/Actor_model

49. Prefer atomic variables over locking (if possible)

Atomic variable are potentially implemented with Compare and Swap spin-lock, which in most cases is more efficient than locking.

Exercise: find the bug in the code below and fix it with atomic variable

http://coliru.stacked-crooked.com/a/d3deefd2bf6e79e2

Benchmark: http://demin.ws/blog/english/2012/05/05/atomic-spinlock-mutex/
Discussion: https://stackoverflow.com/questions/15056237/which-is-more-efficient-basic-mutex-lock-or-atomic-integer

...know the difference between volatile and atomic

Will not discuss it here. But they are not the same.

volatile ≠ atomic

volatile

removal (optimization) is not allowed:

```
volatile int a;
// ... a might be accessed also in other threads
a = 3; // compiler is not allowed to remove this line
a = 4;
```

reordering (optimization) is not allowed:

```
a = 3; // compiler is not allowed to reorder even if "it seems" unrelated
b = foo();
```

50. Use lock guards

lock_guard
RAII management for locks - possibly acquired in ctor, released in dtor

unique_lock
RAII management for unique locking - possibly acquired in ctor, released in dtor

shared_lock
RAII management for shared locking - possibly acquired in ctor, released in dtor

lock / try_lock
generic locking algorithm for locking several mutexes without deadlock
can work with lock guards and locking guard policy tag - defer_lock, adopt_lock

See: http://en.cppreference.com/w/cpp/thread/lock_guard
http://en.cppreference.com/w/cpp/thread/lock
http://en.cppreference.com/w/cpp/thread/try_lock

...be careful with your lock guards

What's the bug in the code below?

```
void Obj::update() noexcept {
    unique_lock<mutex>(m_mutex);
    do_the_mutation();
}
```

...be careful with guards in general...

Give ’em a name

```
void Obj::update() noexcept {
    unique_lock<mutex> g(m_mutex);
    do_the_mutation();
}
```

It’s one of the smallest bug fixes ever, but with production core dumps behind it.

Source - CppCon 2017: Louis Brandy “Curiously Recurring C++ Bugs at Facebook”:
https://www.youtube.com/watch?v=lkgszkPnV8g&t=32m20s

51. std::thread shall be *usually* joined

It is an undefined behavior if main exists while there are still undetached threads running. Detached threads would just finish - which is not the proper action for applicative threads (OK for service threads, such as monitoring, logging, cleanup).

```
int main() {
    thread t([]{for(int i=0; i<1'000; ++i) std::cout << i << ' ';});
    t.join(); // we wait to join t (current thread waits till t finishes)
}
```

An additional simple example: http://coliru.stacked-crooked.com/a/de4ffb6e9f0db97c

52. Use thread_local but carefully, not too much

thread_local variable is a static variable created separately per each thread

- It’s a good tool for avoiding unnecessary locks on resources that can be managed per thread

However,

- It’s a static variable, doesn’t belong to an instance (even if it has a context)
- Too many thread locals is a mess (more than 1...)
- Better manage object per thread

53. Pass references carefully to threads

When passing reference to a thread method, C++ ‘ignores‘ it and passes a copy. To actually pass a reference there is a need to wrap the reference with std::ref (or std::cref for const reference)

Example from: http://en.cppreference.com/w/cpp/thread/lock_tag

```
std::thread t1(transfer, std::ref(my_account), std::ref(your_account), 10);
std::thread t2(transfer, std::ref(your_account), std::ref(my_account), 5);
```

* The actual call to a thread methods goes as:

- a thread is created
- std::thread tries to invoke the method passed to it with std::decay on each parameter
- std::decay: remove references, changes array to pointer, etc.
- std::decay on std::ref leaves the original reference

✎ Pass references to threads - Exercise

Find the bug in the code below, prove that it is a bug and fix it:

```
void func(const Godzilla& godzi);

int main(){
  Godzilla g;
  std::thread t(func, g);
  t.join();
}
```

Solution (don’t peek): http://coliru.stacked-crooked.com/a/07310a5b7ea353be

54. Ask: is it thread safe?

Is shared_ptr thread safe? NO!

Well the ref counter is, but the data pointed to is not!

https://stackoverflow.com/questions/9127816/stdshared-ptr-thread-safety-explained

Are the std containers thread safe? NO! (well, partially)

http://en.cppreference.com/w/cpp/container#Thread_safety

Are the basic C++ operations thread safe?

Copy-Ctor, Default Copy-Ctor? NO!

https://stackoverflow.com/questions/25055328/is-the-default-copy-constructor-thread-safe-in-c

55. Call blocking operations with timeout

When calling blocking operations look for the method version that takes timeout and use it

- if timeout passes write to log
- loop on the operation

Why?

⇒ Easier tracing

⇒ May help in avoiding deadlocks

⇒ Support for graceful shutdown

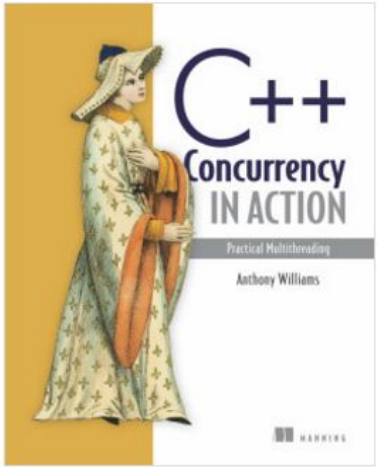
...Concurrency, it’s a big topic

Read more:

C++ Concurrency in Action - Practical Multithreading

Anthony Williams, February 2012, Manning

https://www.manning.com/books/c-plus-plus-concurrency-in-action



...Concurrency, it’s a big topic

See also:
https://channel9.msdn.com/Shows/Going+Deep/C-and-Beyond-2012-Herb-Sutter-Concurrency-and-Parallelism

Concurrency race conditions may be caught with google’s runtime ThreadSanitizer tool
https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

Race condition manual analysis example with tread tables:
https://www.bogotobogo.com/cplusplus/C11/8_C11_Race_Conditions.php

Agenda

What and Why

Classes, Types and Flow

Hierarchies

Documentation, Naming and Style

Memory

Concurrency

Templates and Lambda

Complex code and Refactoring

56. Use Templates for Classes and Algorithms

Google style guide is a bit reluctant about templates:
“Avoid complicated template programming” -
https://google.github.io/styleguide/cppguide.html#Template_metaprogramming

^ Ignore it ^

Go with CppCoreGuidelines which embrace templates
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-templates

Don’t be afraid of templates...
Templates should be your friend for code reuse and generic programming

Read and learn about: SFINAE, Variadic Templates, Specialization, ...

57. Use std containers and algorithms

Do not invent the wheel

- Be familiar with std containers
  https://en.cppreference.com/w/cpp/container
- Be familiar with std algorithms
  https://en.cppreference.com/w/cpp/algorithm

Use them when needed

...get to know std containers

Which of the following is NOT an existing std container?

(a) std::deque
(b) std::forward_list
(c) std::unordered_multiset
(d) all are existing std containers

...get to know std algorithms

Which of the following is NOT an existing std algorithm?

(a) std::find_sequence
(b) std::move_backward
(c) std::rotate_copy
(d) std::adjacent_find

58. Learn from std containers and algorithms

When implementing your own algorithms and data structures

- Explore how similar things were done in the standard library
- Try to align
    - for easier understanding by your users
    - to allow usage of std abilities on your types and algorithms

Implement iterators for your containers

Make your algorithms work with iterators

59. Lambda - capture by name, don’t capture ALL

```
[=] {
   // lambda captured everything around, byval, not recommended
   // (only the variables that we actually use would be copied, but still)
}

[&] {
   // lambda captured everything around, byref, not recommended
   // (only the variables that we actually use would be captured, but still)
}

[&counter, max] {
   // better - captured by name: counter byref, max byval
}
```

60. Lambda - be careful with ref capturing

```
[&counter] {
   // counter is captured byref
}
```

You must be positively sure that the variables that are captured by ref would be alive when the lambda would be invoked.

There is no mechanism of extending the lifetime of captured variables!

Agenda

What and Why

Classes, Types and Flow

Hierarchies

Documentation, Naming and Style

Memory

Concurrency

Templates and Lambda

Complex code and Refactoring

Complex Code

What makes code complex? (1)

- **classes that do more than one thing**
- methods that do more than one thing,
  or **do not use helper methods**
- **too much abstraction**
  (an interface for the interface)
- **managing complex state**

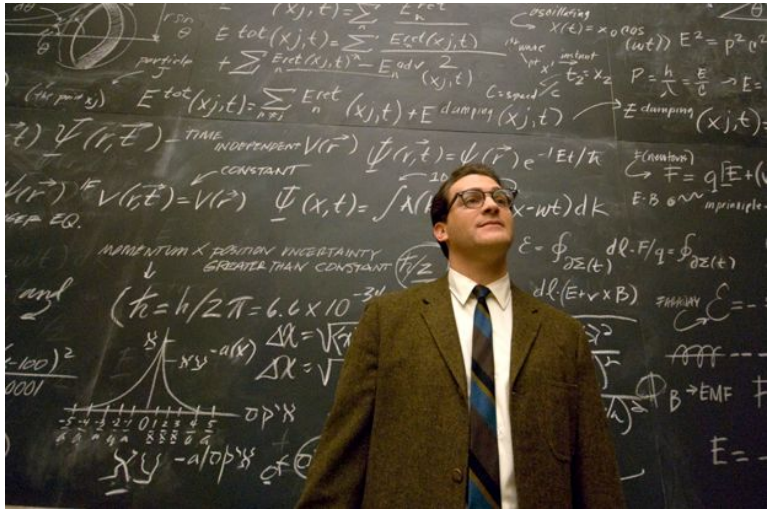


Image Source: http://iscoweb.iut.ac.ir/en/content/adjunct-professor

Complex Code

What makes code complex? (2)

- **multiple inheritance** <= try to avoid
  except for ‘interfaces’
- **indirect flow - messages, triggers**
  <= works well with state machine
- **duplicated code**
  dozens of classes or functions that look the same <= use templates properly
- **templates** <= *do not avoid*, try to isolate the complexities
- **an actual complicated algorithm** <= try to make it a ‘black box’ and test it separately
- **concurrency and race conditions** <= try to isolate, minimize the code that should think about concurrency and race conditions

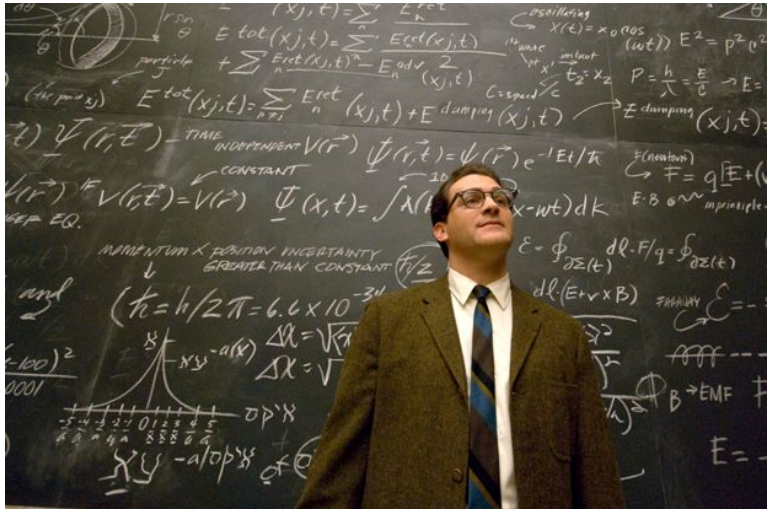
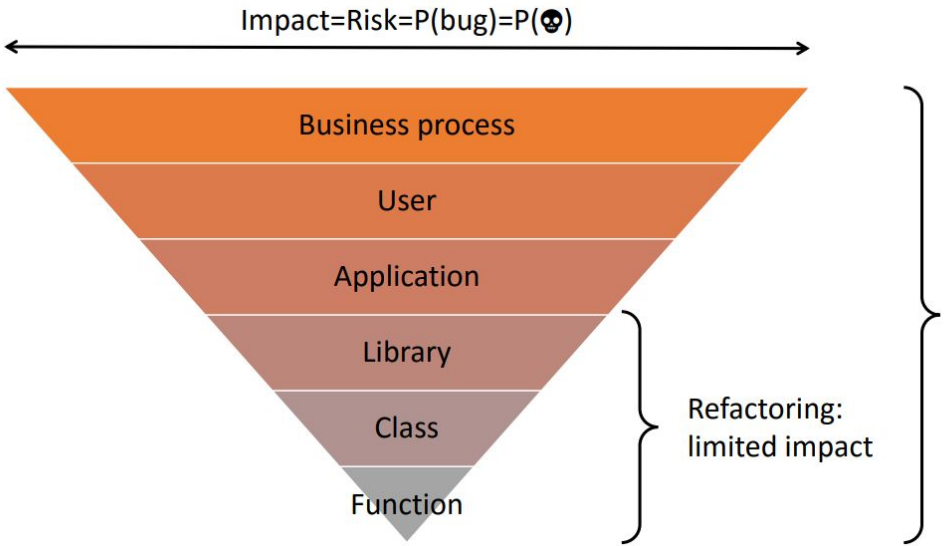
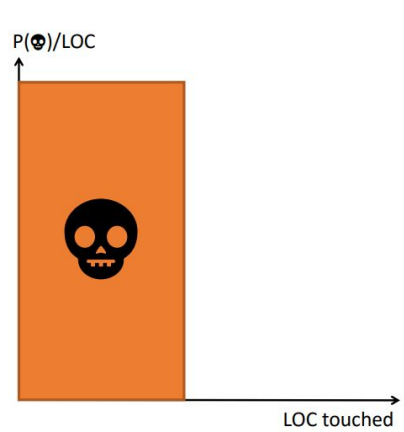
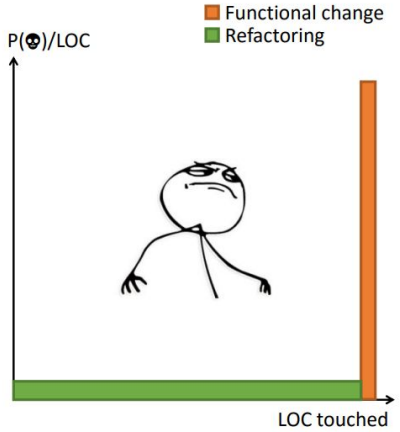


Image Source: http://iscoweb.iut.ac.ir/en/content/adjunct-professor

61. Refactor or Die







CppCon 2017: Mikhail Matrosov “Refactor or die”
https://www.youtube.com/watch?v=fzmjXK9JZ9o

...Refactor - Turn “state hell” into State Machine

```
if(!action.isCancel()) {
 if(queue.activeRequests().containsSimilar(action)) return -1; // already processed, see spe
 if(!is_end_of_day) {
   if(approver.isManager()) {
     if(approver.level >= action.approvingLevel()) {
       int prev_rejections = actionsDAO.getPrevRejections(action, request_date);
       if(prev_rejections < MAX_PREV_REJECTIONS)
         queue.addRequest(action, prev_rejections, request_date);
     } else {
       if(prev_rejections < MAX_ESC_PREV_REJECTIONS) {
          User manager = userDAO.getManager(approver);
          approver.escalate(manager, action, prev_rejections, request_date);
       } else {
         // reject request
       }
     } // handle end of day - add action to next day queue, handle cancel request ...
```

...State Machine

A better way to manage complicated states

- Different states - different behavior
- State is not necessarily mapped based on a single attribute, it is mapped according to the different behavior we wish to model
- A clear context
- Easier communication between Product and Development
- Better tracing
- *Maintenance is not always easier...*

62. Code Review

Review a submit in both directions
(in any order, pros-cons per each order...):

- understand **all diffs in source control**
- understand **the bigger picture**:
  the feature / bug fix - and the related code

Ask questions (read Joel Spolsky first code review by Bill Gates)

Image Soruce:
https://www.reddit.com/r/ProgrammerHumor/comments/2pofnu/extreme_pair_programming

What to look for in Code Review (1)

Image Soruce:
https://www.reddit.com/r/ProgrammerHumor/comments/2pofnu/extreme_pair_programming

Author added own name to submitted changes
inside the code itself (this is not the time for being shy!)

You must understand **how the code was tested**,
which scenarios were covered and how - follow the tests

Reduce complexity - **look for required small refactoring**

You may **add comments** together during the code review but **do not fix the code**

What to look for in Code Review (2)

The best practices! *get them out…*

... rule of 0/3/5, virtual dtor, smart pointers, ...

Follow both the flow and the high level structures
Understand the exact *system state* that is handled
What are the assumed prerequisites?

If **concurrency is involved** you must follow
and **<u>understand the threading model</u>**

Read also: https://medium.com/@mrjoelkemp/giving-better-code-reviews-16109e0fdd36

Image Soruce:
https://www.reddit.com/r/ProgrammerHumor/comments/2pofnu/extreme_pair_programming

Code Review - make *yourself* an example

Polish your code, make it the best ever, clean shine, neat and witty!

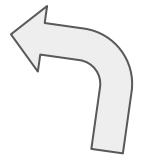
Don’t forget tests.

Don’t forget documentation.

Then summon your team to review your code!

Let the team see how good code looks like.

⇒ Be open for comments - maybe your code is not that perfect…

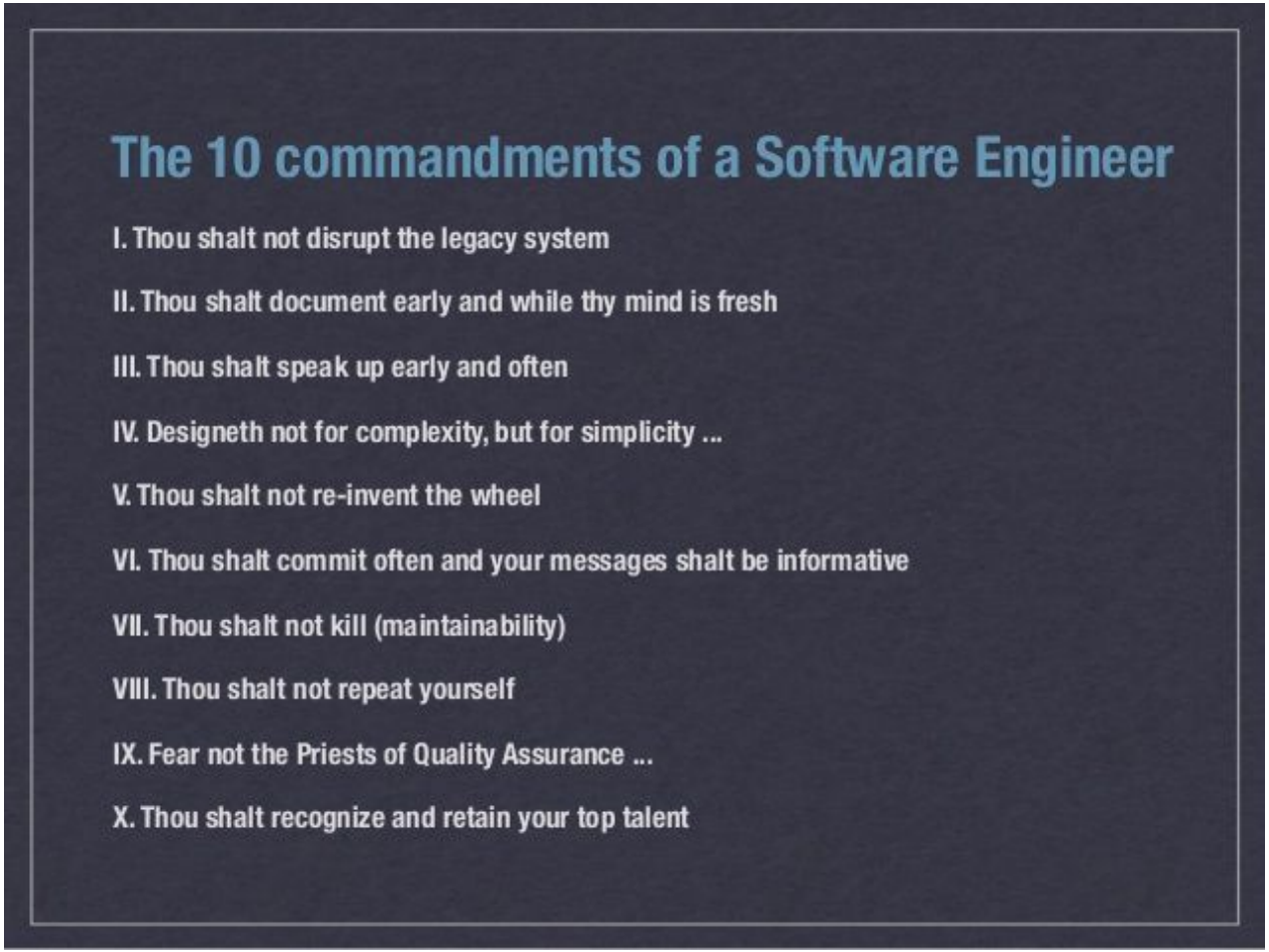


if you don’t have time, let some other expert in the team lead this,
good code reviews to learn from are highly valuable.
Good code is a matter of ongoing education and discipline.

Summary

[10] commandments for Good [C++] Code

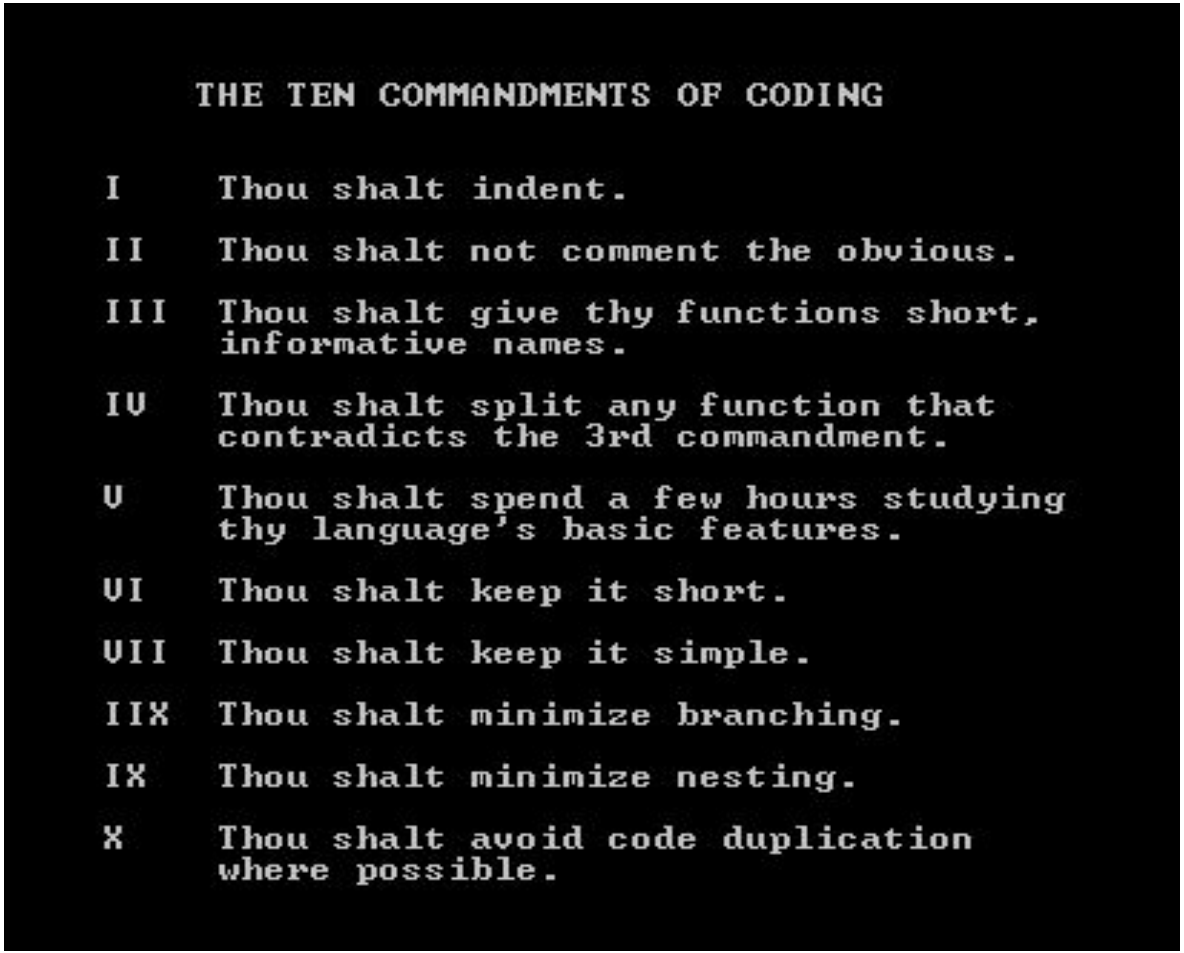
(selected versions)



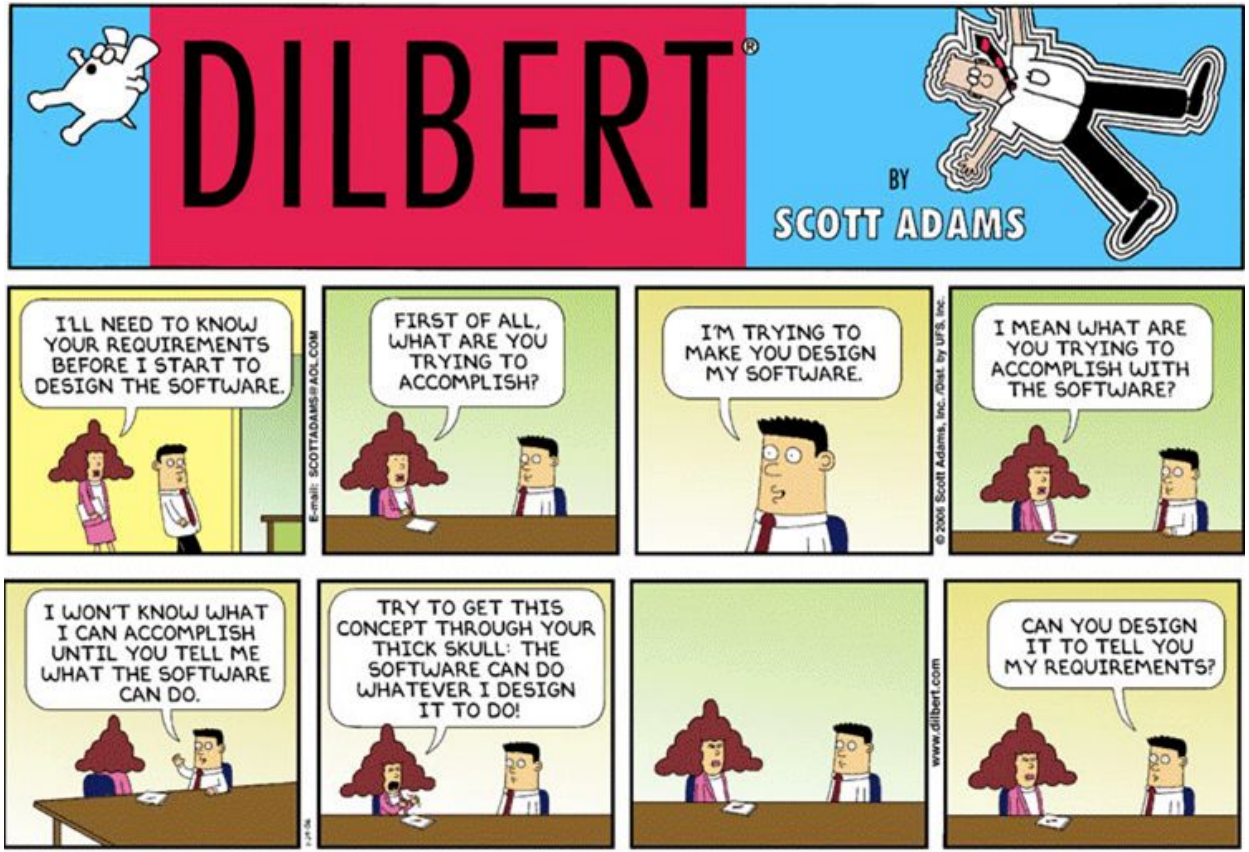
@Sebastian Marek
https://www.slideshare.net/proofek/ten-commandments-of-a-software-engineer-16716894

1. Follow the Rule of Five
2. Almost always use auto (judiciously)
3. Use smart pointers
4. Use RAII for managing resources
5. Use std::string
6. Use standard containers
7. Use standard algorithms
8. Use namespaces
9. Use const
10. Use virtual and override (and final)

https://mariusbancila.ro/blog/2014/10/01/the-cpp-ten-commandments/



http://sourcecodematters.blogspot.com/2014/07/the-ten-commandments.html



The single commandment:

Know the requirements!

for that you need to have requirements

Thank you!

C++ Best Practices Revisited | **Amir Kirsh** | **May 14th**