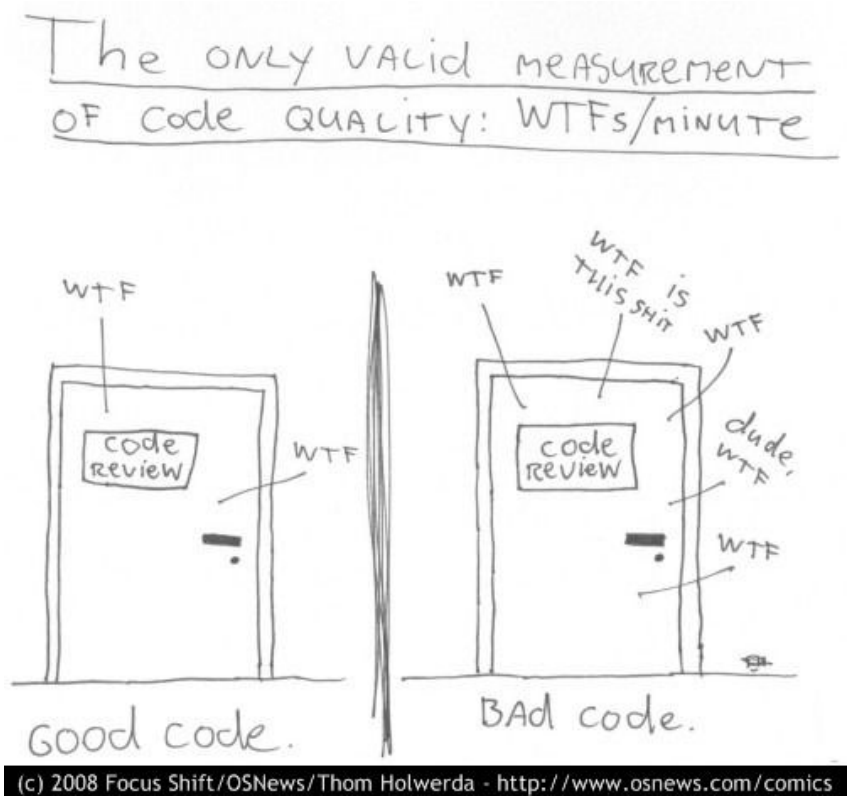
Haifa::C++ presents

# TYPICAL TYPE TYPOS

by Amir Kirsh

# About me

# 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

# *Typical Type Typos*

Common mistakes that relate to (but is not limited to) bad types

- bad design
- efficiency
- undefined behavior
- bugs

# Just before we start

- **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

# Also before we start

**We don’t have time for all slides, so some are annotated with:**


**if you see this ^ on the slide and I’ve missed it, let me know :-)**

# 1. Not hiding 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?**

# 1. Not hiding 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", e.g. a pair that behaves like std::pair but initialized with a single number: first would be the number and second its square, with lazy evaluation
(yet, doable but not straightforward: http://coliru.stacked-crooked.com/a/4c31320c394bcbb5)

# 2. Not using the Rule of Zero

```
// MyClass holds only managed objects
MyClass(const MyClass& m) {
    // increment some counter
    // and do shallow copy
}
```

**What's the problem?**


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

# 2. Not using the Rule of Zero

```
// MyClass holds only managed objects
MyClass(const MyClass& m) {
    // increment some counter
    // and do shallow copy
}
```

```
std::vector<MyClass> vec;
// ...
vec.push_back(my_class_obj);
```


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

# 3. Implementing *move* forgetting *noexcept*

```
MyClass(MyClass&& m) {
    // implement
}
```

**What's wrong?**

# 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

# 4. Forgetting to use std::move or std::forward

**What's wrong here:**

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

**?**

# 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, when you can: Rule of Zero!**

# 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!"); }
```

**T&& above is not Rvalue, it is Forwarding Reference (aka Universal Reference)**

**Below Seems OK, but would not compile**

problem: missing <T>

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

# 5. Not passing by value…

## Pass by value? why? (when?)

# Passing 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

It is potentially more efficient - based on copy elision!

Code: http://coliru.stacked-crooked.com/a/a663b66199f9d25c

See also:

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

# 6. Not using 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
```

# 7. Forgetting 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

# ...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 compile
};
```

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

# ...const iterators

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

**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

# 8. 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

# 9. auto

How much *auto* is too much?

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)

# not using auto ...

What's the problem with the code below:

```
map<Person, int> personCount;
for(const std::pair<Person, int>& pCount : personCount) {
     cout << pCount.first << ": " << pCount.second << endl;
}
```

http://coliru.stacked-crooked.com/a/19731b4611ac2a57

# 10. Beware when returning const lvalue ref!

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);
}
```

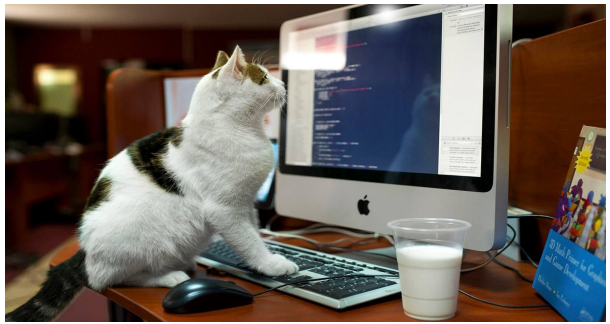


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

# ...Beware of your return type!

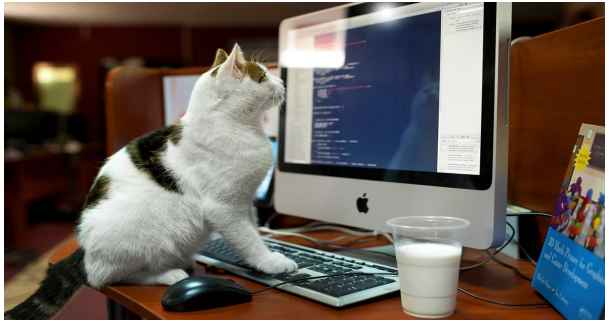


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

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

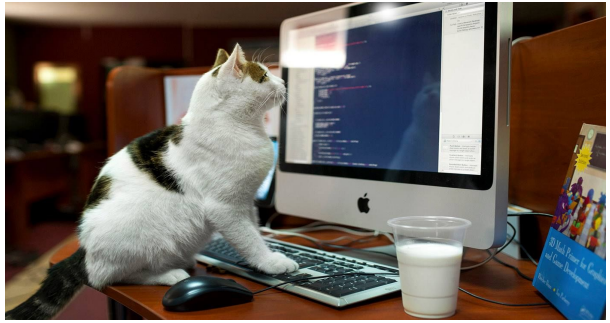


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

# ...Beware of your return type!

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

Add documentation note!

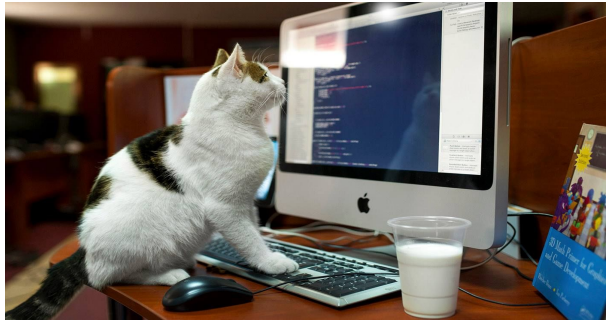


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

```
// 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??

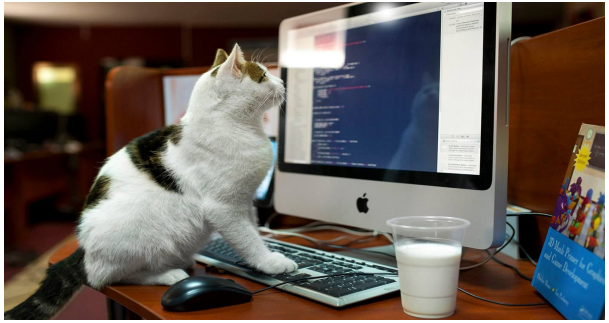


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

```
// 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!

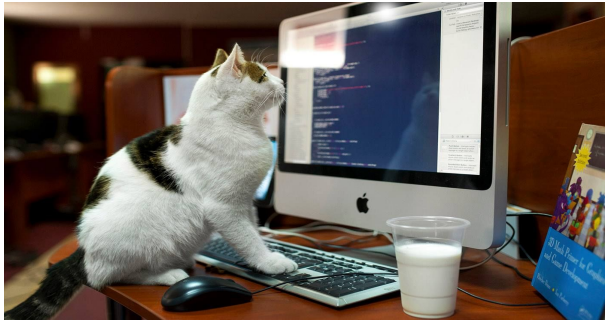


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

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

# ...note also: *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 also: *rvalue* unique_ptr is bug prone, beware

```
auto& ref = *std::make_unique<int>(7);
std::cout << ref << std::endl;
```

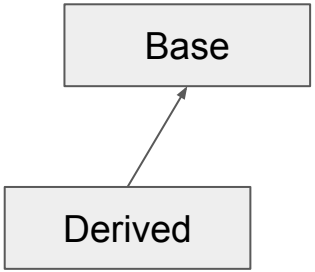
See:
https://stackoverflow.com/questions/57185454/why-does-operator-of-rvalue-unique-ptr-return-an-lvalue

**But this is OK:**

```
std::cout << *std::make_unique<int>(7) << std::endl; // still alive
```

# 11. C-Style Casting on Incomplete types

**What's wrong here?**



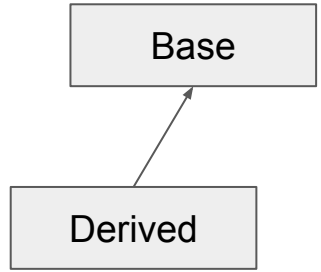

```
void foo(const Base& b);

void foo1(const Derived& d) {
    // foo(d); // can't use polymorphism on incomplete type
    foo((const Base&)d);
}
```

# 11. C-Style Casting on Incomplete types

**What's wrong here?**

```
foo((const Base&)d);
```




The address of *Derived* is not necessarily the same as *Base*

e.g. if Derived has an additional base

http://coliru.stacked-crooked.com/a/e9197e5f37959463

# 12. 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));
```

# 13. Undefined Behavior


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

What's wrong here??

```
int x = foo();
int y = x + 1;
if (x < y) {
    std::cout << "x is smaller";
} else {
    std::cout << "y is smaller or equal";
}
```

See:
gcc: http://coliru.stacked-crooked.com/a/01daf1f23ef832a1
clang: http://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

# 14. Documenting magic numbers requirements

What's wrong here?

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

# use static assert in .cpp  (“protective programming”)

```
// FooWidgetFlags.cpp
//-----------------------------------------------------------
// FooWidgetFlags values must conform to IEEE spec 9927331
//-----------------------------------------------------------
    ASSERT_FooWidgetFlagValue(FooWidgetFlags::NORTH, 100);
    ASSERT_FooWidgetFlagValue(FooWidgetFlags::SOUTH, 2000);
```


```
// in FooWidget.h
#define ASSERT_FooWidgetFlagValue(flag, expected_value) \
    static_assert((int)flag == expected_value, \
      "FooWidgetFlags values must conform to IEEE spec 9927331 flag "#flag \
      " got wrong value, expected: "#expected_value)
```

http://coliru.stacked-crooked.com/a/5941092bed68ba95

# 15. 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)

Use 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>
```

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

# 16. 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; // Clear Object Slicing
  std::vector<Base> vec;
  vec.push_back(d); // Clear Object Slicing
}
```

# 16. Beware of object slicing

```
void foo(Base& b, bool bar) {
     if(bar) b = Derived{};  // <== Slicing?
}

int main() {
  Derived d;
  foo(d, true);
}
```

http://coliru.stacked-crooked.com/a/f52b2058cbe0a896

See also: https://www.learncpp.com/cpp-tutorial/12-8-object-slicing/
https://stackoverflow.com/questions/274626/what-is-object-slicing

# slicing - unique_ptr deleter

```
unique_ptr<A, DeleterA> ptr =
        unique_ptr<B, DeleterB>{new B(), deleterB};
```

**deleterB will not be called when ptr dies**

Code: http://coliru.stacked-crooked.com/a/1a09853c5ec784e3


There is a lecture on this given at Core C++ Meetup, September 2019

# 17. 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;
};
```

# 18. 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

# 19. 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

# 20. Beware of cyclic reference of shared_ptrs

```
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: http://coliru.stacked-crooked.com/a/0bdb6587db374fa7

# 21. Beware of unnecessary copies

What's wrong here?

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

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

# 21. Beware of unnecessary copies

We pass a copy when we can (and should) pass the original:

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

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

# 21. Beware of unnecessary copies

The proper way:

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

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

See: http://coliru.stacked-crooked.com/a/07310a5b7ea353be

# 21. Beware of unnecessary copies

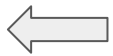
In general, const reference “encourages” copies (by *allowing* them):

```
void foo(const Godzilla& g);

int main() {
    Godzilla_Like g; // g isn't a Godzilla but can cast to it
    foo(g); // Godzilla is born
}
```

## Consider using *templates* (or just *overloading*):

```
template<BigThingy G> void foo(const G& g) {}
  // or
template<BigThingy G> void foo(G&& g) {}
```



assume we use C++20 with a concept for *BigThingy*

# 21. Beware of unnecessary copies

In general, const reference “encourages” copies (by *allowing* them):

```
void foo(const Godzilla& g);

int main() {
    Godzilla_Like g; // g isn't a Godzilla but can cast to it
    foo(g); // Godzilla is born
}
```

**and also - ?**

# 21. Beware of unnecessary copies

In general, const reference “encourages” copies (by *allowing* them):

```
void foo(const Godzilla& g);

int main() {
    Godzilla_Like g; // g isn't a Godzilla but can cast to it
    foo(g); // Godzilla is born
}
```

**and also - <u>avoid silent casting by adding *‘explicit’* to expensive castings</u>**

# 21. Beware of unnecessary copies

last one...

# 21. Beware of unnecessary copies

In call to lambdas

```
std::vector<Godzilla> vec;
auto finder = [vec] () { /* do something with vec */ };
```

# Thank you!

```
void conclude(auto greetings) {
    while(still_time() && have_questions()) {
        ask();
    }
    greetings();
}

conclude([]{ std::cout << "Thank you!"; });
```