TYPICAL TYPE TYPOS

Amir Kirsh

Typical Type Typ🐞s

Feel free to participate, it’s an interactive session.

But please be noted that the session is recorded.

If you unmute, your audio and video stream would be captured together with the name you are logged in with.

Chat messages may also get into the final recording.

About me

Lecturer
Academic College of Tel-Aviv-Yaffo
Tel-Aviv University

Member of the Israeli ISO C++ NB

Co-Organizer of the **CoreCpp** conference and meetup group

Developer Advocate for C++
INCREDIBUILD

Previously
Programmer, Dev Manager
Chief Programmer @ Comverse

INCREDIBUILD

Suffering from slow builds?

It’s not just waste of time

It affects your dev cycles
and productivity

Typical Type Typ🐞s

Common errors that relate to *bad use* or *implementation* of types

- bad design
- inefficiency
- undefined behavior
- bugs
- compilation errors

Just before we start

- **Compiler warnings:**
  always solve them, they are stronger than any best practice! (*note:* `-Wall` is not all)

- **Static code analysis tools:**
  use them, they help you conform with best practices
  see for example: https://rules.sonarsource.com/cpp/RSPEC-5912

- **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 may not have time for all slides, so some are annotated with:

if you see this ^ on the slide it means we may skip it

Last note before we start

It's a game and there is a prize

you are requested to count your points

but you MUST submit your answer

before *my* answer* is revealed

* *my* answer would be considered right even if you disagree

We have a few of those
to be sent to the winners

let’s try it...

let’s try it...

```
int main() {
    int i = i * 0; // what’s the value of i?
}
```

don’t count your answer on this one, it’s just a warm up

let’s try it...

```
int main() {
    int i = i * 0; // what’s the value of i?
}
```

A It’s undefined behavior due to bad initialization

B It’s undefined behaviour due to bad initialisation

C It’s undefined behavior due to bad initialisation

D It’s undefined behaviour due to bad initialization

OK, Ready?

OK, Ready?

Let’s start

1. What’s wrong here?

```
template<typename K, typename V>
void print(const std::map<K, V>& m) {
    for(const std::pair<K, V>& p: m) {
        // print p
    }
}
```

1. What’s wrong here?

```
template<typename K, typename V>
void print(const std::map<K, V>& m) {
    for(const std::pair<K, V>& p: m) {
        // print p
    }
}
```

A dangling pointer

B inefficiency

C potential leak

D compilation error

1. What’s wrong here?

```
template<typename K, typename V>
void print(const std::map<K, V>& m) {
    for(const std::pair<K, V>& p: m) {
        // print p
    }
}
```

A dangling pointer

[✓ B inefficiency]

C potential leak

D compilation error

Redundant temporaries due to casting

```
pair<const K, V> => const pair<K, V>&
```

- temporary pair
- temporary copy of K
- temporary copy of V

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

Why?

auto casting to const lvalue reference is allowed

remove of top level const is allowed

The proper way - no redundant copies

```
template<typename K, typename V>
void print(const std::map<K, V>& m) {
    for(const std::pair<const K, V>& p: m) { /* … */ }
}

// Or BETTER:
template<typename K, typename V>
void print(const std::map<K, V>& m) {
    for(const auto& p: m) { /* … */ }
}
```

The proper way - no redundant copies

```
// Or EVEN NICER:
template<typename K, typename V>
void print(const std::map<K, V>& m) {
    for(const auto& [key, val]: m) { /* … */ }
}
```

C++17 structured binding

Related

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)

2. What’s wrong here?

```
template<typename K, typename V>
void print(const std::map<K, V>& m) {
    for(const auto& p: m) {
        // print p
    }
}
```

A const issues

B should use forwarding ref

C not generic enough

D bad style

2. What’s wrong here?

```
template<typename K, typename V>
void print(const std::map<K, V>& m) {
    for(const auto& p: m) {
        // print p
    }
}
```

A const issues

B should use forwarding ref

[✓ **C** not generic enough]

D bad style

2. What’s wrong here?

```
template<typename K, typename V>
void print(const std::map<K, V>& m) {
    for(const auto& p: m) {
        // print p
    }
}

map<std::string, int, std::greater<std::string> > strCount;
print(strCount); // <== compilation error
```

```
    no matching function for call to 'print'
    template argument deduction/substitution failed:
    mismatched types 'std::less<...>' and 'std::greater<...>'
```

The proper way - more generic...

Option 1 - supporting any kind of std::map

```
template<typename K, typename V, typename... AdditionalArgs>
void print (const std::map<K, V, AdditionalArgs...>& m) {
    /* … */
}
```

The proper way - more generic...

Option 2 - supporting any kind of “mapping container”

```
template<template<class, class, class...> class MAP,
    typename K, typename V, typename... AdditionalArgs>
void print (const MAP<K, V, AdditionalArgs...>& m) {
    /* … */
}
```

Problem?

The proper way - more generic...

Option 2 - supporting any kind of “mapping container”

```
template<template<class, class, class...> class MAP,
    typename K, typename V, typename... AdditionalArgs>
void print (const MAP<K, V, AdditionalArgs...>& m) {
    /* … */
}
```

Problem? it’s too greedy

The proper way - more generic...

Option 3 - add restrictions via SFINAE / C++20 requires / C++ concepts

```
template<typename T>
concept map_type = /* … */

void print (const map_type auto& m) { // C++20
    /* … */
}
```

See:

https://stackoverflow.com/questions/64087934/how-to-write-a-c-concept-restricting-the-template-to-stdmap-and-stdunorder
https://stackoverflow.com/questions/25749917/how-can-i-make-a-function-that-takes-either-a-map-or-an-unordered-map

3. What’s wrong here?

```
// using C++20 auto as parameter type
void printPair(const auto& p) {
    std::cout << p.first << ", " << p.second;
}
```

3. What’s wrong here?

```
void printPair(const auto& p) {
    std::cout << p.first << ", " << p.second;
}
```

A dangling pointer

B inefficiency

C potential leak

D bad design

3. What’s wrong here?

```
void printPair(const auto& p) {
    std::cout << p.first << ", " << p.second;
}
```

A dangling pointer

B inefficiency

C potential leak

[✓ **D** bad design]

Issue #1: language issue!

not hiding your privates is wrong

Data members should be private

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

Why?

Because it doesn’t properly allow different behaviors, e.g. a pair initialized with a single number, with the second being lazy evaluated to its square

(yet, doable but not straightforward: http://coliru.stacked-crooked.com/a/4c31320c394bcbb5)

Issue #2: too generic && not generic enough!

Too generic

```
void printPair(const auto& p) {
    std::cout << p.first << ", " << p.second;
}
```

Not generic enough
What about std::tuple of two
(i.e. “twople”)

Specifically, a better implementation

```
void printPair(const auto& p) {
    std::cout << std::get<0>(p) << ", " << std::get<1>(p);
}
// works for std::pair, std::tuple, std::array
```

or even better

```cpp
template<class P> concept Pair = requires(P p) {
  requires std::tuple_size<P>::value == 2;
  std::get<0>(p);
  std::get<1>(p);
};
```

```cpp
void print(const Pair auto& p) {
    std::cout << std::get<0>(p) << ", " << std::get<1>(p);
}
```

4. What’s wrong here?

```
// using C++20 auto parameter
void zero_initialize_all( auto& container ) {
    for( auto& val : container ) {
        val = {};
    }
}
```

4. What’s wrong here?

```
// using C++20 auto parameter
void zero_initialize_all( auto& container ) {
    for( auto& val : container ) {
        val = {};
    }
}
```

A 1st auto should be: auto&&

B 2nd auto should be: auto&&

C 2nd auto should be by value!

D 1st auto should be by value!

4. What’s wrong here?

```
// using C++20 auto parameter
void zero_initialize_all( auto& container ) {
    for( auto& val : container ) {
        val = {};
    }
}
```

A 1st auto should be: auto&&

[✓ B 2nd auto should be: auto&&]

C 2nd auto should be by value!

D 1st auto should be by value!

Rvalues can appear on the left

```
// using C++20 auto parameter
void zero_initialize_all( auto& container ) {
    for( auto&& val : container ) {
        val = {};
    }
}


vector<bool> vb = {true, false, true};
zero_initialize_all(vb);
```

to support
this creature

Beware of Specialization...

`std::vector<bool>` **is considered a language accident as it doesn't behave as other vector types**

^ don't do such things in your code!

One should be able to use the specialized version, the same as using the base template, without being aware of the exact type being used

* Liskov Substitution Principle rephrased for templates

5. Beware of specialization and inheritance

```cpp
// Base template
template<class T> struct Foo {
  static void print() {
    std::cout << "Something";
  }
};
```

```cpp
struct Pet {};
struct Dog : public Pet {};

// Specialized version
template<> struct Foo<Pet> {
  static void print() {
    std::cout << "Pet";
  }
};
```

```cpp
// M A I N
int main() {
  Foo<Dog>::print();
}
```

What would this main print?

Source:
https://stackoverflow.com/questions/7928871/good-practices-regarding-template-specialization-and-inheritance

5. Beware of specialization and inheritance

```
// Base template
template<class T> struct Foo {
  static void print() {
    std::cout << "Something";
  }
};
```

```
struct Pet {};
struct Dog : public Pet {};

// Specialized version
template<> struct Foo<Pet> {
  static void print() {
    std::cout << "Pet";
  }
};
```

```
// M A I N
int main() {
  Foo<Dog>::print();
}
```

What would this main print?

A Something **B** Pet **C** Dog **D** Program does not compile

5. Beware of specialization and inheritance

```
// Base template
template<class T> struct Foo {
  static void print() {
    std::cout << "Something";
  }
};
```

```
struct Pet {};
struct Dog : public Pet {};

// Specialized version
template<> struct Foo<Pet> {
  static void print() {
    std::cout << "Pet";
  }
};
```

```
// M A I N
int main() {
  Foo<Dog>::print();
}
```

What would this main print? →

[✓ A Something]   B Pet   C Dog   D Program does not compile

By the way, same result:

```
// Base template
template<class T> struct Foo {
  static void print() {
    std::cout << "Something";
  }
};
```

```
struct Pet {};
struct Dog : public Pet {};

// Specialized version
template<> struct Foo<Pet*> {
  static void print() {
    std::cout << "Pet";
  }
};
```

What would this main print?

```
// M A I N
int main() {
  Foo<Dog*>::print();
}
```

[✓ **A** Something]   **B** Pet   **C** Dog   **D** Program does not compile

6. What’s wrong here?

```
class MyClass {
    // MyClass holds only “RAII objects” (i.e. which manage their own lifetime)
public:
    MyClass() = default;
    MyClass(const MyClass& m) {
        // increments a static counter counting copies then copies all members
    }
    // other methods, but no other c’tors / d’tor
};
```

6. What’s wrong here?

```
class MyClass {
     // MyClass holds only “RAII objects” (i.e. which manage their own lifetime)
public:
     MyClass() = default;
     MyClass(const MyClass& m) {
          // increments a static counter counting copies then copies all members
     }
     // other methods, but no other c’tors / d’tor
};
```

A dangling reference

B inefficiency

C potential leak

D compilation error

6. What’s wrong here?

```
class MyClass {
    // MyClass holds only “RAII objects” (i.e. which manage their own lifetime)
public:
    MyClass() = default;
    MyClass(const MyClass& m) {
        // increments a static counter counting copies then copies all members
    }
    // other methods, but no other c’tors / d’tor
};
```

A dangling reference

[✓ B inefficiency]

C potential leak

D compilation error

Not using the Rule of Zero

```
std::vector<MyClass> vec;

// ...

vec.push_back(my_class_obj); // no move :-/
```

```
// defaulting the move operation is ok...
MyClass(MyClass&&) = default;
MyClass& operator=(MyClass&&) = default;
// but rule of zero is better!*
```

* See also:
The Rule of Zero revisited: The Rule of All or Nothing by Arne Mertz

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

How to do it right? (for example…)

```
class MyClass : Counter<MyClass> {
    // MyClass holds only “RAII objects”
public:
    // Use rule of zero!
};
```

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

7. What’s wrong here?

```
MyClass(MyClass&& m) {

    // this type needs to implement move
    // actual implementation comes here
    // assume constructor initialization list is used if relevant

}
```

7. What’s wrong here?

```
MyClass(MyClass&& m) {
    // this type needs to implement move
    // actual implementation comes here
    // assume constructor initialization list is used if relevant
}
```

A dangling reference

B inefficiency

C missing “const”

D compilation error

7. What’s wrong here?

```
MyClass(MyClass&& m) {

    // this type needs to implement move
    // actual implementation comes here
    // assume constructor initialization list is used if relevant

}
```

A dangling reference

[✓ B inefficiency]

C missing “const”

D compilation error

Implementing *move* forgetting *noexcept*

vector's push_back implementation is allowed to use *move ctor* only if it is declared as noexcept:

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

Why? to avoid possible bad scenario of exception during move

- we call push_back to add a Godzilla to vector<Godzilla>
- capacity of vector is exhausted, so vector capacity shall be enlarged
- new bigger allocation is made, old Godzillas shall be moved
- while moving Godzilla at index N an exception is thrown
- we have now two broken vectors and cannot rollback

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

Implementing *move* forgetting *noexcept*

Don’t believe there is a difference?

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

vs.

```
A(A&& a) /* oops forgot */ {
    // code
}
```

Implementing *move* forgetting *noexcept*

Don’t believe there is a difference?

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

vs.

```
A(A&& a) /* oops forgot */ {
    // code
}
```

```
in A's empty ctor
in A's move ctor
in A's move ctor
in A's move ctor
in A's move ctor
in A's move ctor
in A's move ctor
in A's move ctor
in A's move ctor
...
```

```
in A's empty ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
...
```

http://coliru.stacked-crooked.com/a/15a89b45b0dcfedd

8. If you want to copy, pass by-value

```
std::set<string> long_strings;
void store(string s) {
  long_strings.insert(std::move(s));
}
```

what’s wrong here?

8. If you want to copy, pass by-value

```
std::set<string> long_strings;
void store(string s) {
   long_strings.insert(std::move(s));
}
```

what's wrong here?

A dangling reference

B inefficiency

C potential leak

D compilation error

8. If you want to copy, pass by-value

```
std::set<string> long_strings;
void store(string s) {
  long_strings.insert(std::move(s));
}
```

what's wrong here?

A dangling reference

[✓ B inefficiency]

C potential leak

D compilation error

Be cautious with passing by value

```
std::set<string> long_strings;
void store(string s) {
    long_strings.insert(std::move(s));
}
```

we copy even if not needed

the rule of “if you need to copy pass by value” needs great care

See: https://stackoverflow.com/questions/10231349/are-the-days-of-passing-const-stdstring-as-a-parameter-over

Related:

The *copy and swap idiom* is elegant (maybe) but *inefficient*…

http://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf

https://stackoverflow.com/questions/24014130/should-the-copy-and-swap-idiom-become-the-copy-and-move-idiom-in-c11/24018053#24018053

Alternatives

```
void store(const string& s) {
  long_strings.insert(s);
}

void store(string&& s) {
  long_strings.insert(std::move(s));
}
```

OR

```
template<typename T> requires
std::convertible_to<T, std::string>
void store(T&& s) {
  long_strings
      .insert(std::forward<T>(s));
}
```

Alternatives

Inserting *existing item* into std::set via our *store* function

	byval	const ref	const ref + rval	forwarding ref
lvalue	**copy**	**---**	**---**	**---**
rvalue	**move**	**copy**	**move**	**move**

GCC (with libstdc++) and Clang (with libc++) both with -O3
https://godbolt.org/z/954KeM

Alternatives

Inserting *existing item* into std::set via our *store* function

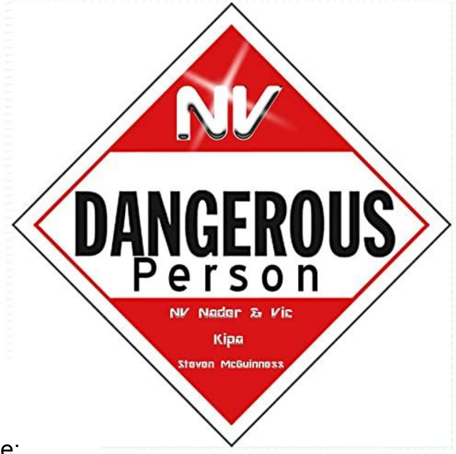
	byval	const ref	const ref + rval	forwarding ref
lvalue	**copy**	**---**	**---**	**---**
rvalue	**move**	**copy**	**move**	**move**

better

GCC (with libstdc++) and Clang (with libc++) both with -O3
https://godbolt.org/z/954KeM

9. What’s wrong here?

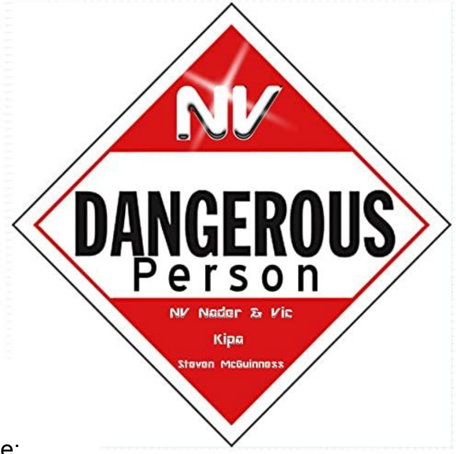
```
Person p = "John";
```



Source:
https://www.youtube.com/watch?v=zBH0wei8pTw

9. What’s wrong here?

```
Person p = "John";
```



Source:
https://www.youtube.com/watch?v=zBH0wei8pTw

A constructor isn’t explicit

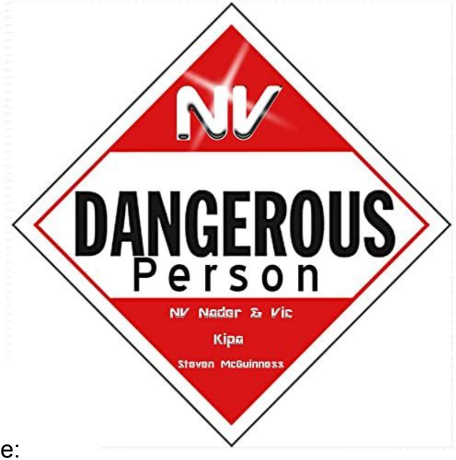
B potential inefficiency

C bad use of char*

D potential leak

9. What’s wrong here?

```
Person p = "John";
```



Source:
https://www.youtube.com/watch?v=zBH0wei8pTw

[✓ A] constructor isn’t explicit

B potential inefficiency

C bad use of char*

D potential leak

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
```

Why?

We want to avoid:

```
    void foo(const vector<int>& v);
    foo(7); // doesn’t compile, ctor is explicit
```

10. What’s wrong here?

```
class Foo {
  int* ptr;
public:
  // … proper ctors dtor etc.
  int& get1() const { return *ptr; }
  void foo1() const { *ptr = 42; }
  int*& get2() { return ptr; }
  void foo2()  { ++ptr; }
};
```

assume there is *a good reason* we do not use smart pointers, so “not using smart pointers” is not the answer here!

10. What’s wrong here?

```
class Foo {
  int* ptr;
public:
  // … proper ctors dtor etc.
  int& get1() const { return *ptr; }
  void foo1() const { *ptr = 42; }
  int*& get2() { return ptr; }
  void foo2()  { ++ptr; }
};
```

assume there is *a good reason* we do not use smart pointers, so “not using smart pointers” is not the answer here!

A code doesn’t compile

B ptr should be mutable

C code breaks logical const

D code breaks physical const

10. What’s wrong here?

```
class Foo {
  int* ptr;
public:
  // … proper ctors dtor etc.
  int& get1() const { return *ptr; } // compiles, but smelly
  void foo1() const { *ptr = 42; }   // compiles, but smelly
  int*& get2() { return ptr; } // ok, compiles only if method is not const
  void foo2()  { ++ptr; }      // ok, compiles only if method is not const
};
```

- A code doesn’t compile
- B ptr should be mutable
- [✓ C code breaks logical const]
- D code breaks physical const

...logical const vs. physical const

The compiler protects you on physical const

Preserving logical const is *on you*

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

```
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
};
```

...const iterators

Note that iterators and smart pointers can also be const.

Use them correctly!

```
class AnotherFoo {
    std::list<int> numbers;
public:
    list<int>::iterator get1() { /*...*/ }
    list<int>::const_iterator get2() const { /*...*/ }
};
```

SKIP

...const smart pointers

To protect content owned by a smart pointer, use ‘const’ with the inner type:

```
void foo1(shared_ptr<const A> ptra); // the content is const
void foo2(const shared_ptr<A> ptra); // the pointer is const, not the content

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

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

Forgetting const on methods and parameters

Keeping const correctness:

- **widens the possible usage of a function**
- **protects us from indeliberate modifications**

Remember that there is also constexpr

Note that constexpr when relevant is even better than just const
(e.g. for constants that are assigned with a value in compile time)

- efficiency
- correctness

Note also that functions and constructors can also be marked as constexpr

C++17 also adds ‘if constexpr’ as a possible replacement for SFINAE

11. What can go wrong here?

```
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);
}
```

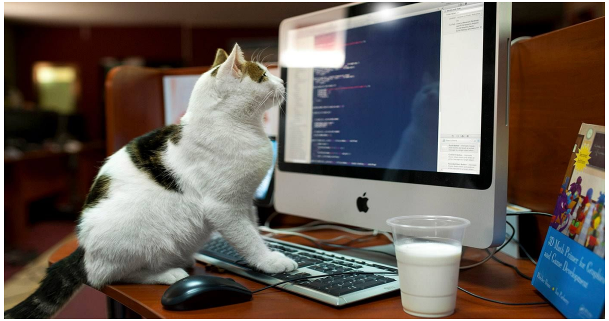


Image Source: http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

11. What can go wrong here?

```
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);
}
```

A the map can be empty

C inefficiency

B dangling reference

D code is too generic

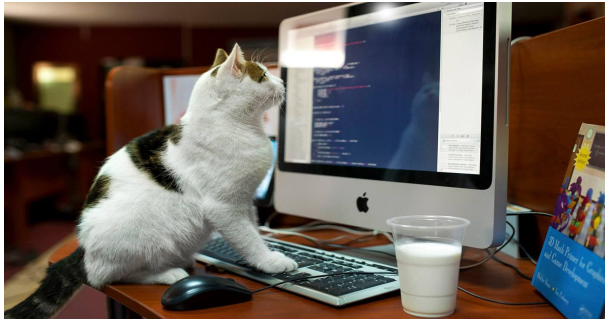


Image Source: http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

11. What can go wrong here?

```
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);
}
```

A the map can be empty

[✓ B dangling reference]

C inefficiency

D code is too generic

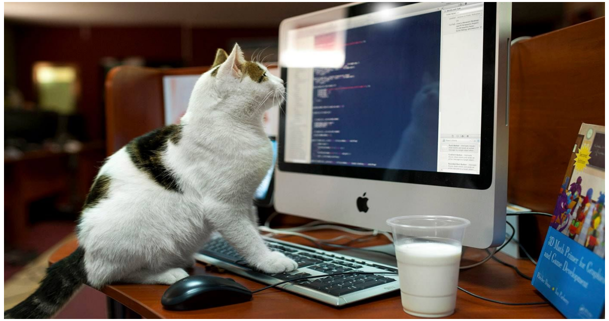


Image Source:
http://www.magicindie.com/magicblog/wp-content/uploads/2013/12/cat_programmer.jpg

...Beware of your return type!

```
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;
```

Note that ASAN locates the problem

Code presenting the problem:

http://coliru.stacked-crooked.com/a/e7983b00ebb59520

We can compile the code with ASAN sanitize flag
(see: https://github.com/google/sanitizers/wiki/AddressSanitizer -fsanitize=address)
which identifies the problem right ahead!
http://coliru.stacked-crooked.com/a/74d5b2e2d0876226

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

...Beware of your return type!

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

Add documentation note!

```
// 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 const& and still be safe?

Is there a way??

```
// 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!

Yes, there’s 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
```

12. What’s wrong here?

```
// [a]
for(char c: std::string{"hello"}) {
  // do something with c
}

// [b]
for(const char& c: std::string{"hello"}) {
  // do something with c
}

// [c]
for(char c: Person{"John"}.name()) {
  // do something with c
}
```

12. What’s wrong here?

```
// [a]
for(char c: std::string{"hello"}) {
  // do something with c
}

// [b]
for(const char& c: std::string{"hello"}) {
  // do something with c
}

// [c]
for(char c: Person{"John"}.name()) {
  // do something with c
}
```

A All three loops may be using a dangling ref

B `[a]` is OK, `[b]` and `[c]` may be using a dangling ref

C `[a]` and `[c]` are OK `[b]` may be using a dangling ref

D `[a]` and `[b]` are OK `[c]` may be using a dangling ref

12. What’s wrong here?

```
// [a]
for(char c: std::string{"hello"}) {
  // do something with c
}

// [b]
for(const char& c: std::string{"hello"}) {
  // do something with c
}

// [c]
for(char c: Person{"John"}.name()) {
  // do something with c
}
```

A All three loops may be using a dangling ref

B `[a]` is OK, `[b]` and `[c]` may be using a dangling ref

C `[a]` and `[c]` are OK `[b]` may be using a dangling ref

D `[a]` and `[b]` are OK `[c]` may be using a dangling ref

lifetime of top most expression in range *is* extended

```
for(char& c: std::string{"John"}) {
  // do something with c
}
```

is like:

```
{
  auto&& _range = std::string{"John"};
  auto _begin = std::begin(_range);
  auto _end = std::end(_range);
  for ( ; _begin != _end; ++_begin) {
    char& c = *_begin;
    // do something with c
  }
}
```

lifetime extended, we are fine

...Beware of “dependent temporaries” in a loop

```
for(char c: Person{"John"}.name()) {
  // do something with c
}
```

is like:

```
{
  auto&& _range = Person{"John"}.name();
  auto _begin = std::begin(_range);
  auto _end = std::end(_range);
  for ( ; _begin != _end; ++_begin) {
    char c = *_begin;
    // do something with c
  }
}
```

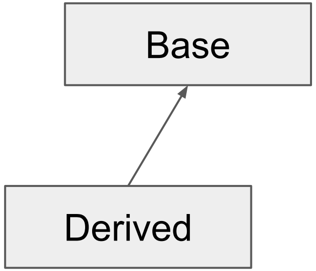
if `name()` returns a ref to a member, that’s a dangling ref

See for yourself:
http://coliru.stacked-crooked.com/a/938cb19812c8dbf8

There is a proposal to fix this behavior:
http://josuttis.com/download/std/D2012R0_fix_rangebasedfor_201029.pdf

13. What may go wrong here?

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

class Derived;

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



13. What may go wrong here?

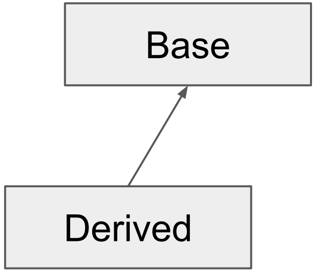
```
void foo(const Base& b);

class Derived;

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

A Base might be abstract

B Runtime bad casting

C Compilation error

D Infinite recursion



Base

Derived

13. What may go wrong here?

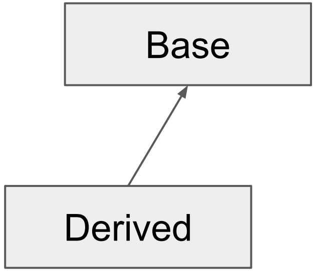
```
void foo(const Base& b);

class Derived;

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

A Base might be abstract

[✓ B Runtime bad casting]

C Compilation error

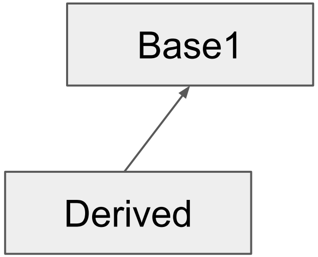
D Infinite recursion



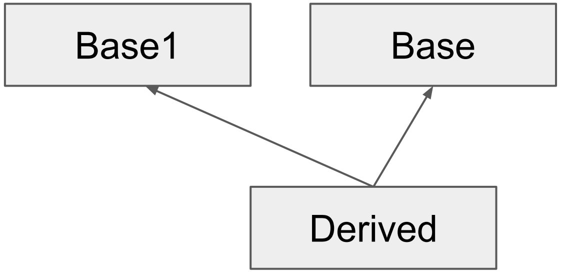
C-Style Casting on Incomplete types

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

1. Base can change, casting on incomplete type still compiles. Oops…



2. 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



Don’t use C-Style casting!

Use here *static_cast* or *dynamic_cast*
(depending on what you actually know at compile time)

14. What can go wrong here?

```
using meters = double;

meters distance = 7.5;
doSomething(distance);
```

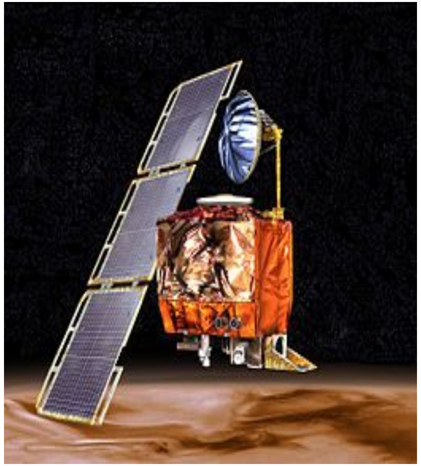


image source:
https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

14. What can go wrong here?

```
using meters = double;

meters distance = 7.5;
doSomething(distance);
```

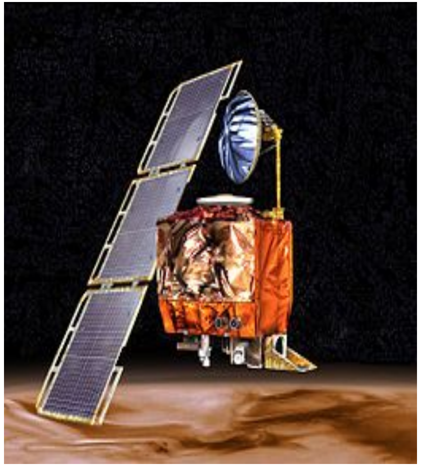


image source:
https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

A Measurement units can get wrong

B Casting from double to int / float

C No type enforcement

D All the above

14. What can go wrong here?

```
using meters = double;

meters distance = 7.5;
doSomething(distance);
```

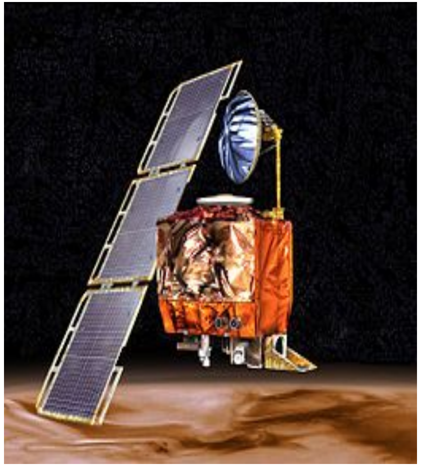


image source:
https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

A Measurement units can get wrong

B Casting from double to int / float

C No type enforcement

[✓ D All the above]

14. What can go wrong here?

```
using meters = double;

meters distance = 7.5;
doSomething(distance);

// the method that we call might be
void doSomething(float distance_feet);
Or:
void doSomething(int distance_cm);
```

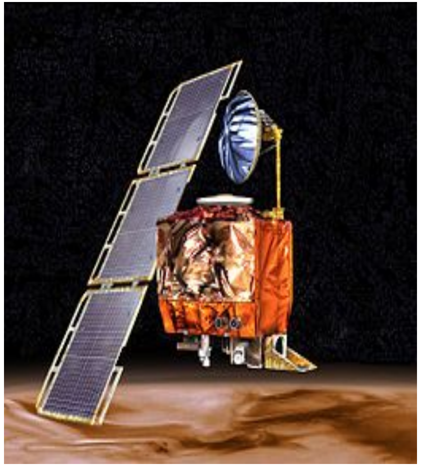


image source: https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

...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

...Strong Types

Consider using:

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));
```

...don’t just wrap it with a struct

[This is NOT a Strong Type]

```
struct Meters { double m; }

Rectangle r(Meters(10), Meters(12));

// but then this would also work:
Rectangle r({10}, {12});
```

Note also that this is against the encapsulation rule
Such structs turn to grow into fully functioning classes with public members...

Wrong Type is *Actually* Crashing

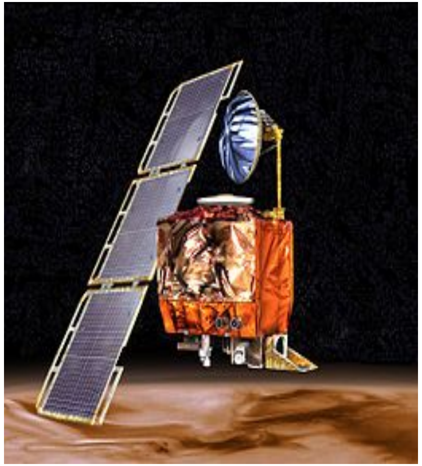
The Ariane 5 crash:

https://en.wikipedia.org/wiki/Ariane_5_Flight_501

https://hownot2code.com/2016/09/02/a-space-error-370-million-for-an-integer-overflow/

Mars Climate Orbiter crash:

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



Safe Types Options

boost::units

CppCon 2015: Robert Ramey “Boost Units”

https://github.com/nholthaus/units

https://github.com/pierreblavy2/unit_lite

https://github.com/bernedom/SI

https://github.com/mpusz/units

https://github.com/joboccara/NamedType

15. What will be printed?

```
int x = foo(0); // foo(0) returns MAX_INT
int y = x + 1;
if (x < y) {
    std::cout << "x is smaller";
} else {
    std::cout << "y is smaller or equal";
}
```

15. What will be printed?

```
int x = foo(0); // foo(0) returns MAX_INT
int y = x + 1;
if (x < y) {
    std::cout << "x is smaller";
} else {
    std::cout << "y is smaller or equal";
}
```

A x is smaller

B y is smaller or equal

C can print anything...

D code doesn’t compile

15. What will be printed?

```
int x = foo(0); // foo(0) returns MAX_INT
int y = x + 1;
if (x < y) {
    std::cout << "x is smaller";
} else {
    std::cout << "y is smaller or equal";
}
```

A x is smaller

B y is smaller or equal

[✓ **C** can print anything...]

D code doesn’t compile

Undefined Behavior

```
int x = foo(0); // foo(0) returns MAX_INT
int y = x + 1;
if (x < y) {
    std::cout << "x is smaller";
} else {
    std::cout << "y is smaller or equal";
}
```

signed integer overflow is undefined behavior

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

Compare:
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 and overflow undefined behavior

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://stackoverflow.com/questions/18195715/why-is-unsigned-integer-overflow-defined-behavior-but-signed-integer-overflow-is

More on Overflow and Safe Numerics

CppCon 2018: Robert Ramey "Safe Numerics"

boost::numeric_cast

boost safe numerics

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

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

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

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

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

Also related: CppCon 2019: Marshall Clow "std::midpoint? How Hard Could it Be?"

Other Resources on Undefined Behavior

It’s a popular topic in CppCon =>

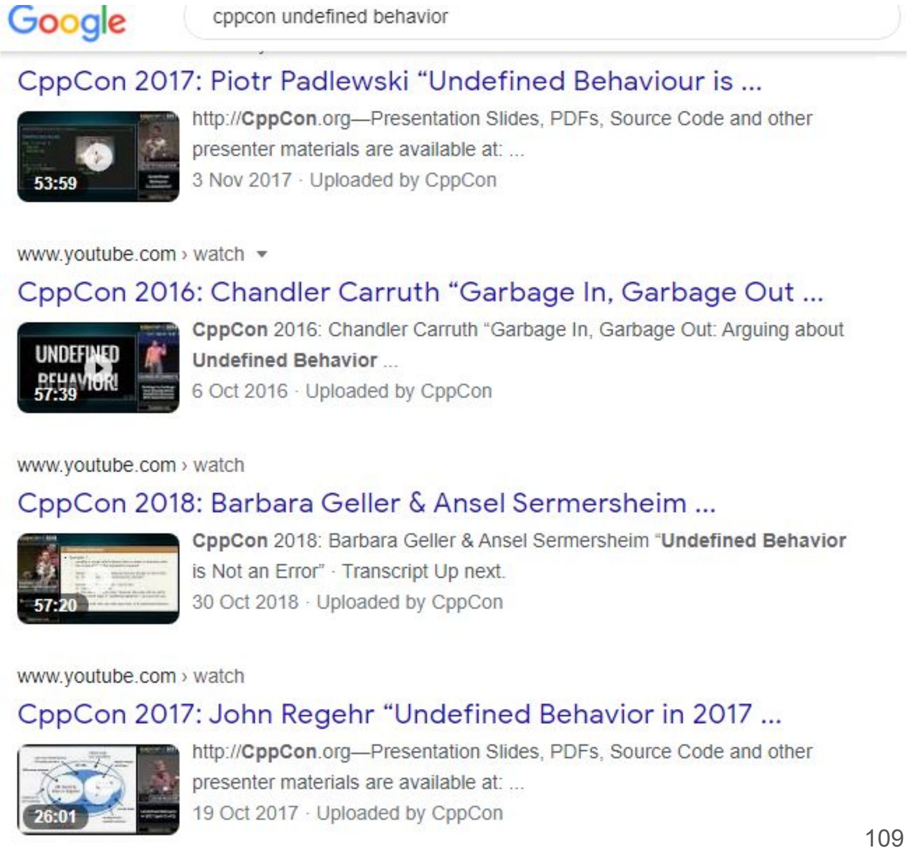
https://www.google.com/search?q=cppcon+undefined+behavior

Featuring also in ACCU:

https://accu.org/content/conf2014/MarshallClowUndefined Behavior-ACCU2014.pdf

The LLVM Project blog, on Undefined Behavior:

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



cppcon undefined behavior

CppCon 2017: Piotr Padlewski “Undefined Behaviour is ...

http://CppCon.org—Presentation Slides, PDFs, Source Code and other presenter materials are available at: ...

3 Nov 2017 · Uploaded by CppCon

www.youtube.com › watch

CppCon 2016: Chandler Carruth “Garbage In, Garbage Out ...

CppCon 2016: Chandler Carruth “Garbage In, Garbage Out: Arguing about Undefined Behavior ...

6 Oct 2016 · Uploaded by CppCon

www.youtube.com › watch

CppCon 2018: Barbara Geller & Ansel Sermersheim ...

CppCon 2018: Barbara Geller & Ansel Sermersheim “Undefined Behavior is Not an Error” · Transcript Up next.

30 Oct 2018 · Uploaded by CppCon

www.youtube.com › watch

CppCon 2017: John Regehr “Undefined Behavior in 2017 ...

http://CppCon.org—Presentation Slides, PDFs, Source Code and other presenter materials are available at: ...

19 Oct 2017 · Uploaded by CppCon

16. What’s wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    void push(T&& t) {
        vec.push_back(std::forward<T>(t));
    }
    // ...
};
```

16. What’s wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    void push(T&& t) {
        vec.push_back(std::forward<T>(t));
    }
    // ...
};
```

A **T&&** in **push** is NOT a forwarding reference, thus **compilation error**

B **T&&** in **push** is NOT a forwarding reference, thus we support only push of rvalues

C **push** may add to the vector a dangling ref

D **push** may inefficiently copy when it can move an item into the vector

16. What’s wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    void push(T&& t) {
        vec.push_back(std::forward<T>(t));
    }
    // ...
};
```

A **T&&** in **push** is NOT a forwarding reference, thus **compilation error**

[✓ B] **T&&** in **push** is NOT a forwarding reference, thus we support only push of rvalues

C **push** may add to the vector a dangling ref

D **push** may inefficiently copy when it can move an item into the vector

The proper way - option 1

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    void push(T&& t) {
        vec.push_back(std::move(t));
    }
    void push(const T& t) {
        vec.push_back(t);
    }
    // ...
};
```

The proper way - option 2

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    template<typename U>
      requires std::convertible_to<U, T>
    void push(U&& u) {
        vec.push_back(std::forward<U>(u));
    }
    // ...
};
```

17. What’s wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    // ...
    T pop() {
        T& e = vec.back();
        vec.pop_back();
        return std::move(e);
    }
};
```

17. What’s wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    // ...
    T pop() {
        T& e = vec.back();
        vec.pop_back();
        return std::move(e);
    }
};
```

A **pop** returns a dangling reference

B **pop** moves from a dangling reference (code would be OK without the call to **`std::move`**)

C **pop** has UB: “moving out” from a vector is impossible

D the reference **e** is being invalidated once we call **pop_back**

17. What’s wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    // ...
    T pop() {
        T& e = vec.back();
        vec.pop_back(); // e’s dtor called
        return std::move(e);
    }
};
```

A **pop** returns a dangling reference

B **pop** moves from a dangling reference (code would be OK without the call to **`std::move`**)

C **pop** has UB: “moving out” from a vector is impossible

[✓ **D** the reference **e** is being invalidated once we call **pop_back**]

The proper way

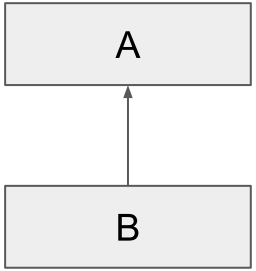
```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    T pop() {
        T e = std::move(vec.back());
        vec.pop_back();
        return e;
    }
    // ...
};
```

Code for items 16-17: http://coliru.stacked-crooked.com/a/b339af287c876ec4

See also: https://stackoverflow.com/questions/6438086/iterator-invalidation-rules
https://stackoverflow.com/questions/12600330/pop-back-return-value
https://stackoverflow.com/questions/40500821/how-to-store-a-value-obtained-from-a-vector-pop-back-in-c

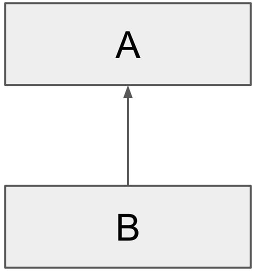
18. What’s the problem here?

```
void conditionalAssign(bool condition, A& a1, const A& a2) {
    if(condition) a1 = a2;
}

B b {1, 1};
conditionalAssign(shouldAssign, b, B{2, 2});
```



18. What's the problem here?

```
void conditionalAssign(bool condition, A& a1, const A& a2) {
     if(condition) a1 = a2;
}

B b {1, 1};
conditionalAssign(shouldAssign, b, B{2, 2});
```



A potentially wrong method call

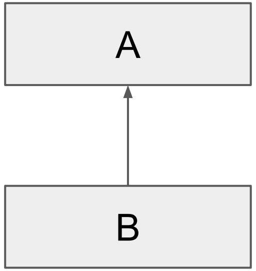
B potential dangling reference

C potential self-assignment

D potential infinite recursion

18. What’s the problem here?

```
void conditionalAssign(bool condition, A& a1, const A& a2) {
     if(condition) a1 = a2;
}

B b {1, 1};
conditionalAssign(shouldAssign, b, B{2, 2});
```



[✓ **A** potentially wrong method call]

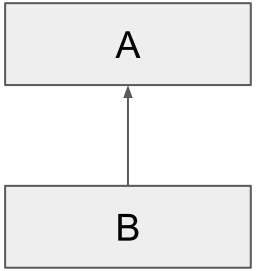
B potential dangling reference

C potential self-assignment

D potential infinite recursion

Assignment is usually not virtual

```
void conditionalAssign(bool condition, A& a1, const A& a2) {
    if(condition) a1 = a2; // default assignment is not virtual
}

B b {1, 1};
conditionalAssign(shouldAssign, b, B{2, 2});
```



http://coliru.stacked-crooked.com/a/c7346fb21e850f6d

Above might be considered as a variant or a special case of *object slicing*.

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

Beware of object slicing in general

```
// 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
}
```

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

See a discussion in stackoverflow on the subject:
https://stackoverflow.com/questions/56308336/why-unique-ptr-doesnt-prevent-slicing-of-custom-deleter

19. What’s wrong here?

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

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

19. What’s wrong here?

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

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

A potential memory leak

B inefficient design

C potential infinite recursion

D code doesn’t compile

19. What’s wrong here?

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

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

[✓ **A** potential memory leak]

B inefficient design

C potential infinite recursion

D code doesn’t compile

Beware of cyclic reference of shared_ptrs

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

```
class B {
    weak_ptr<A> pa; // <= code change
    // ...
};
```

Cyclic references would never be released…

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

Possible solution: use weak_ptr

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

Last One

Last One

Are you ready?

20. What’s wrong here?

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

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

20. What’s wrong here?

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

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

problem is here

20. What’s wrong here?

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

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

problem is here

A potential memory leak

B redundant copying

C creating an unjoinable thread

D thread is not copyable

20. What’s wrong here?

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

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

problem is here

A potential memory leak

[✓ **B** redundant copying]

C creating an unjoinable thread

D thread is not copyable

How can we fix it?

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

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

problem is here

Unnecessary copy passing param to a thread

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

Score Summary

Score Summary

18-20 points

Score Summary

18-20 points

You probably wrote so many bugs, which made you the real C++ pro you are.

Ask for a raise. You deserve it.

(And email me for the prize*: kirshamir@gmail.com)

*** a draw might be conducted**

Score Summary

12-17 points

Score Summary

12-17 points

You are good.

Remember that Bjarne rates himself 7/10 in C++.

(And email me also for the prize*: kirshamir@gmail.com)

*** a draw might be conducted**

Score Summary

6-11 points

Score Summary

6-11 points

You are a bit rusty.

Consider moving to Rust.

Score Summary

0-5 points

Score Summary

0-5 points

Don’t feel too bad.

But, be sure to get your code reviewed, especially if working on life critical systems.

Thank you!

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

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