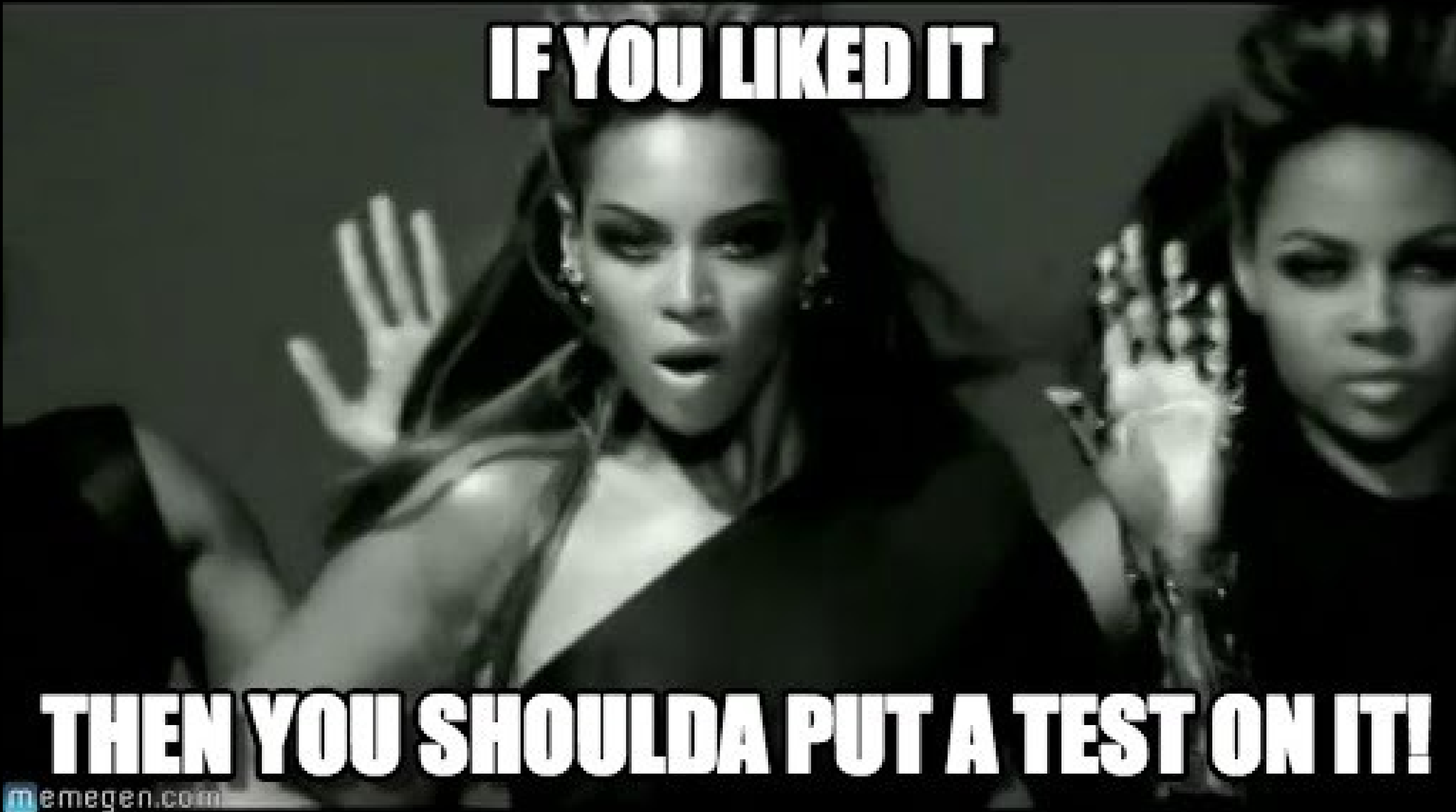
Mocking C++

Peter Bindels – @dascandy42 – github.com/dascandy

Creating software:

1. Make it work
2. Make it reliable
3. Make it fast

Unit testing by Beyonce



TDD/Unit testing principles

- Only features that you test work
- Things without test could work or break
- At this point focus only on functional testing

TDD/Unit testing principles

- Tests are fast
- Tests are reliable

TDD/Unit testing principles

How do you make fast & reliable tests?

TDD/Unit testing principles

- Multiple kinds of test approach
  - Test in small units
  - Test in integrated subsystems
  - Test in production-like environment
- Separate test types
  - Functional first
  - Performance, reliability, security, ... later

TDD/Unit testing principles

- Decouple
  - Separate out things that
    - Are not part of your module’s responsibility
    - Make your module hard to test
  - Test its role only
  - Stub/mock out dependencies
    - If any

Basic mocking example

```
class IBank {
public:
  virtual void Transfer(account_t from,
account_t to, size_t amount) = 0;
};

void BuyABook(IBank* bank,
   account_t client) {
  bank->Transfer(client, 345, 1500);
}
```

Basic mocking example

```
TEST(CanBuyABook) {
  MockRepository mocks;
  IBank* myBank = mocks.Mock<IBank>();
  account_t myAccount = 42;
  account_t merchant = 345;
  mocks
    .ExpectCall(myBank, IBank::Transfer)
    .With(myAccount, merchant, 1500);
  BuyABook(myBank, myAccount);
  mocks.VerifyAll();
}
```

Mocking in C++

- No compile-time reflection on classes
- No way to use reflection output to create a class / object / function

What can we reflect already

- **`Mock<MyClass>();`**
  - Type of MyClass
  - Size of MyClass
  - Alignment of MyClass

  - Could SFINAE to find out more specific functions
  - No generic exploration (C++03)
  - Very limited generic exploration (C++11)

What can we reflect already

- **`ExpectCall(myObj, MyClass::MyFunc)`**
  - Type of myObj
  - Type that contains MyFunc
    - May not be myObj’s class nor MyClass
  - Return value of MyFunc
  - Argument(s) of MyFunc
  - Const/volatileness of MyFunc

What can we reflect already

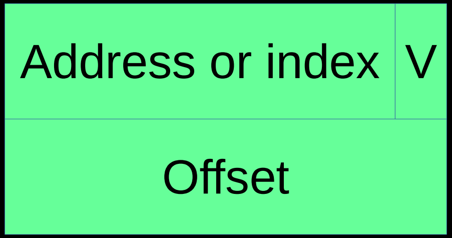
- **`ExpectCall(myObj, MyClass::MyFunc)`**
  - File in which this call is done
  - Line on which this call is done

What can we reflect already

- **`ExpectCall(myObj, MyClass::MyFunc)`**
  - What offset from myObj should be applied before calling
  - Whether myFunc is virtual
    - If it is, what index
    - If it is not, what address it’s at

Anatomy of a member function pointer

```
struct MFP {
   union {
     CODEPTR funcaddr;
     int vtable_index;
     bool isVirtual;
   };
   int offset_from_base;
};
```



Address or index | V

Offset

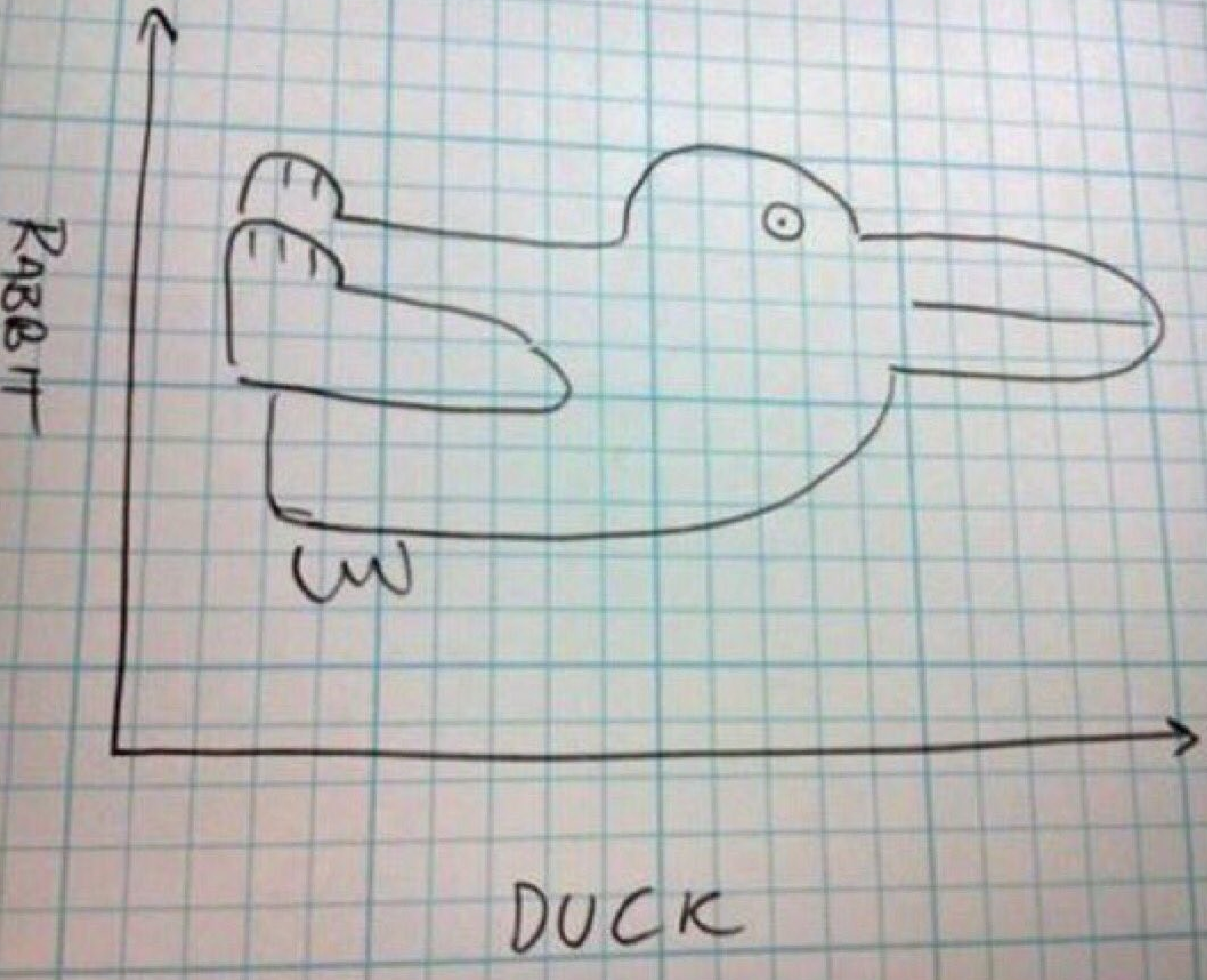
How to make a mock

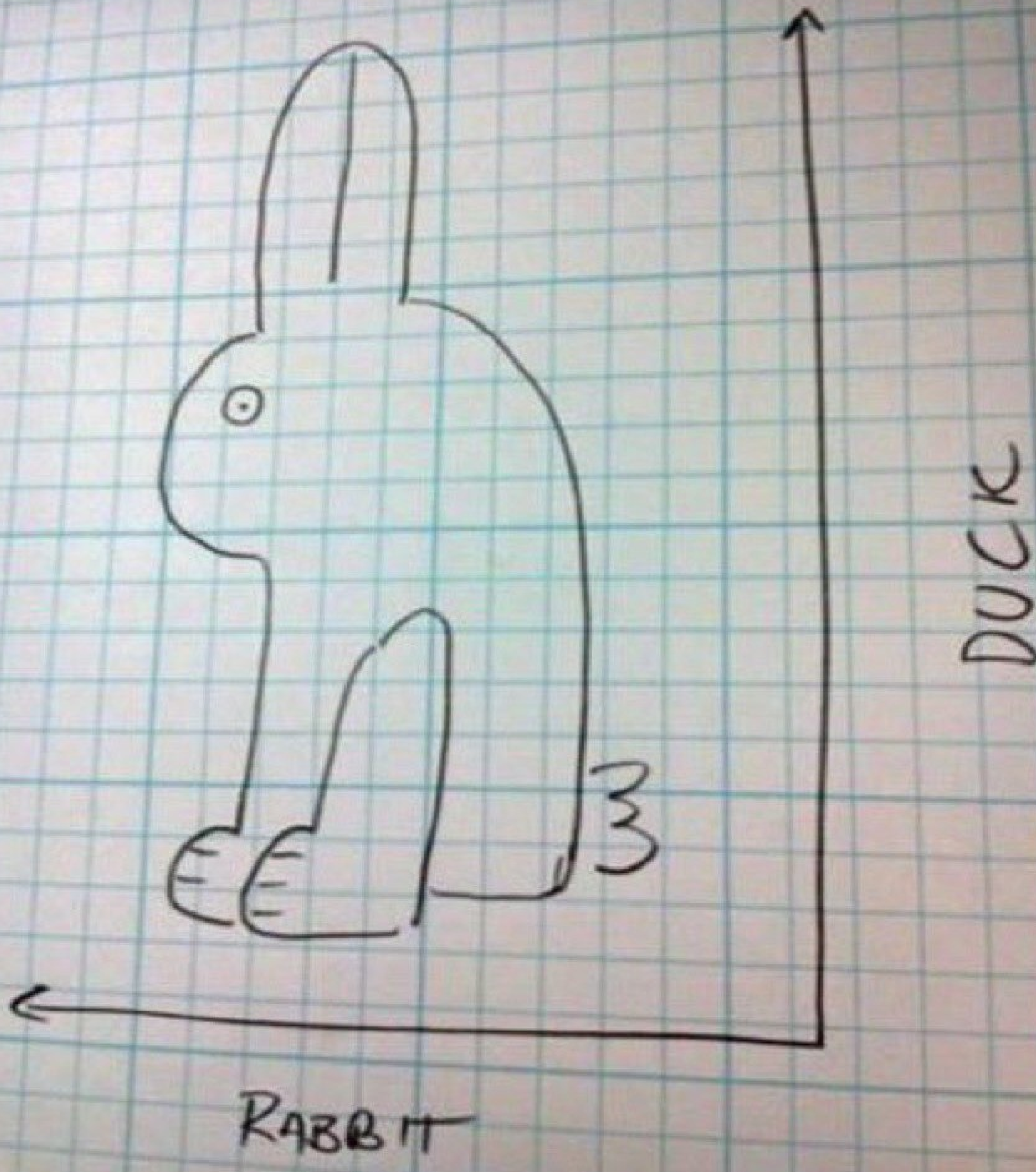
- Create new mock class
- Inherit from interface to be mocked
- Implement all functions with a generic call into a library

How to make a mock

- DRY violation
- Maintenance overhead
  - Add a function, add it to all the mocks too
- Latent bugs
  - Accidentally forgetting const, minor typos
  - Subverted since C++11 by override
    - If you use it

But *why* ?


RABBIT
DUCK


DUCK
RABBIT

So what is an object?

```
class X {
public:
   X();
   virtual ~X();
   virtual void func();
   int x;
};
```

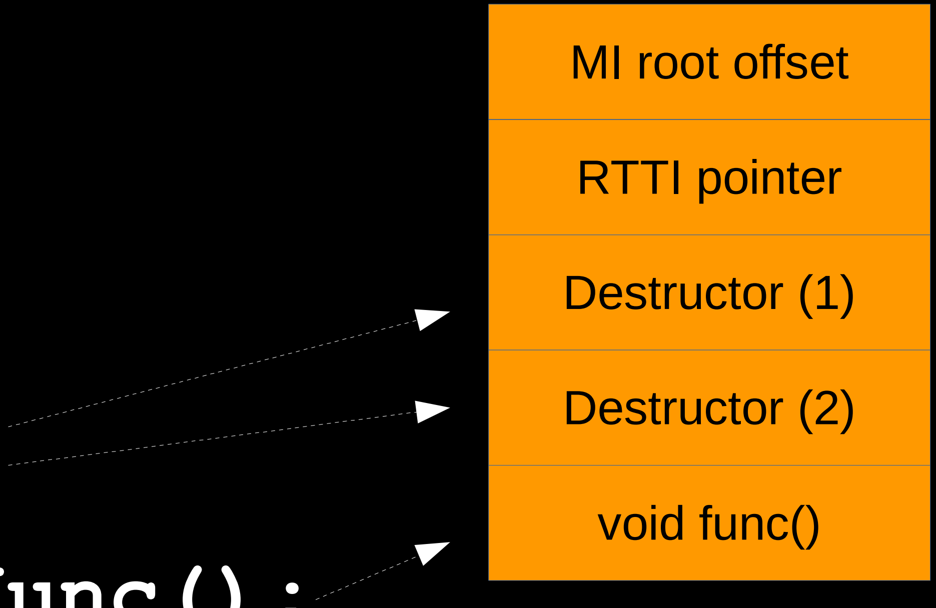
So what is an object?

```
class X {
public:
  X();
  virtual ~X();
  virtual void func();
  int x;
};
```

Vtable pointer
int value

So what is an object?

```
class X {
public:
   X();
   virtual ~X();
   virtual void func();
   int x;
};
```



MI root offset

RTTI pointer

Destructor (1)

Destructor (2)

void func()

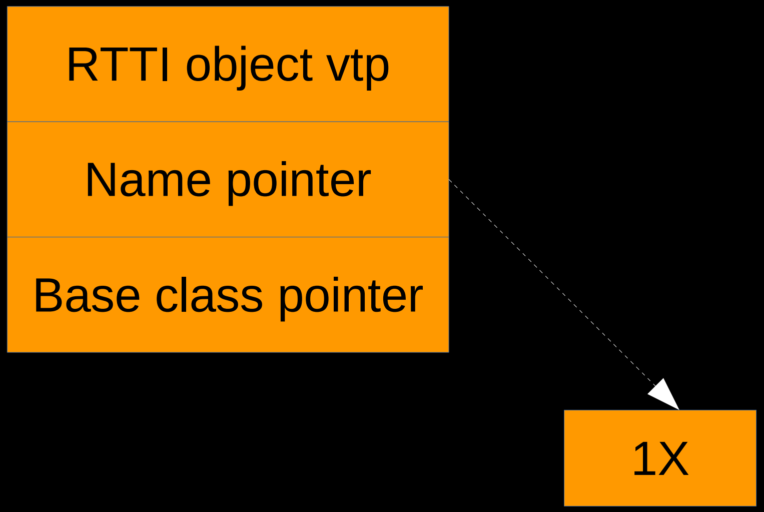
Virtual Function Table

- Two entries above the “actual table”
  - MI object offset
  - RTTI object pointer
- Flat table with a mishmash of virtual member functions
  - One entry per function
  - Two entries for destructor

    (everywhere except MSVC)
- Everything’s read-only and per-class

So what is an object?

```
class X {
public:
   X();
   virtual ~X();
   virtual void func();
   int x;
};
```



RTTI (non-MSVC)

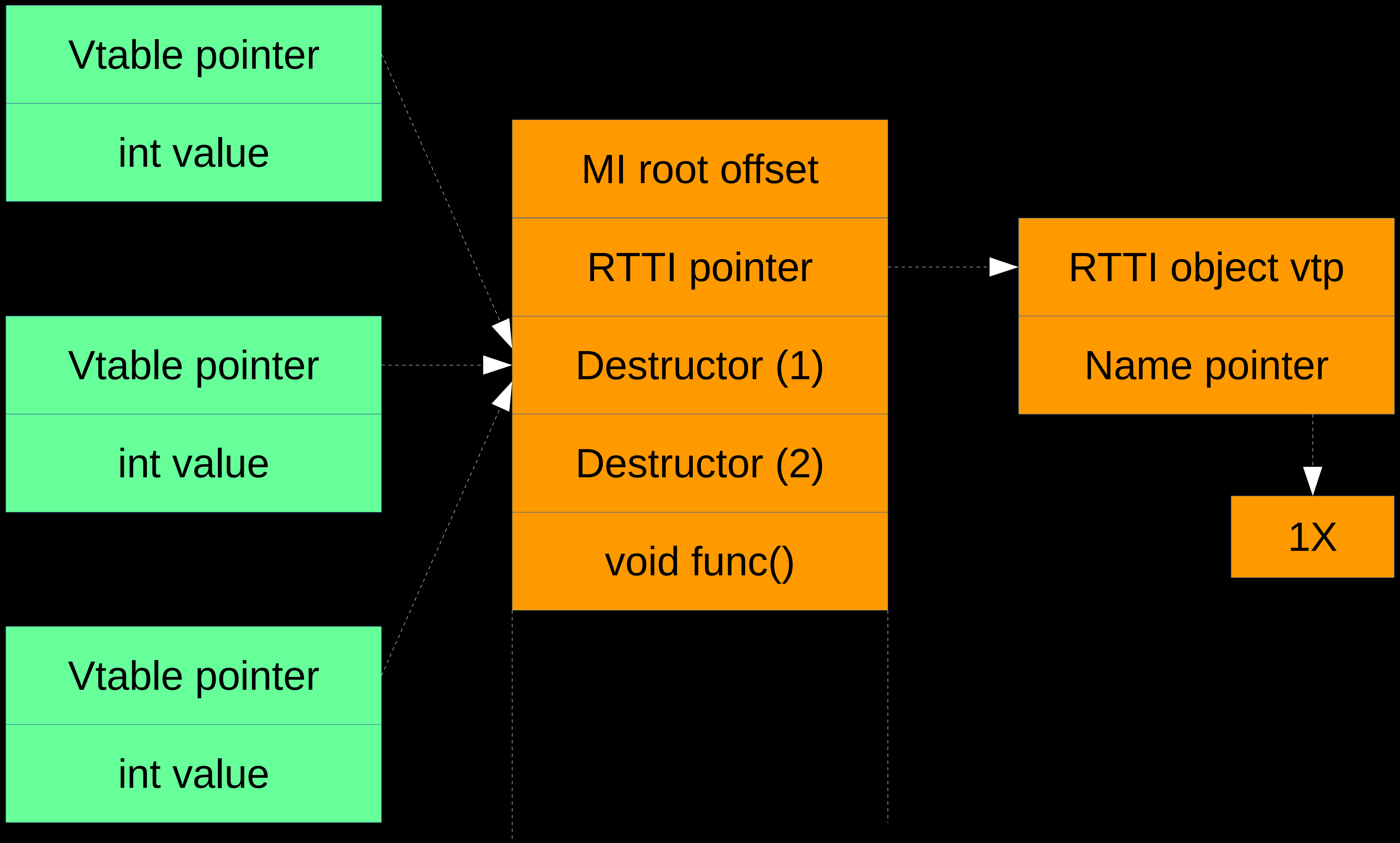
- Contains
  - Virtual fuctions to runtime use type
  - Name of class
  - 0 or more base class’ type_info pointers

RTTI (non-MSVC)

- Only used for
  - Dynamic cast
  - Runtime typeid
  - Exception handler matching

- Only read-only data

Graphically



Let’s implement a C++ class in assembly!

Mangling

`_ZN1XC1Ev`

`_Z:`	This is a C++ identifier
`N...E:`	This is a nested list (think **`X::Y::Z`**)
`1X:`	Literal name “X”
`C1:`	Class constructor
`v:`	No argument list

So this is **`X::X()`**

Mangled names because there are subtleties not visible in unmangled

```
_ZN1XC1Ev:
  ; Vtable pointer plus 2 machine words
  mov    $_ZTV1X + $0x10,%rdx
  mov    %rdx,(%rdi)
  retq

_ZN1XD1Ev:
  retq

_ZN1X4funcEv:
  retq
```

```
_ZN1XD0Ev:
  ; Defer to regular destructor
  callq  _ZN1XD1Ev
  mov    $0x10,%esi
  ; Call operator delete
  callq  _ZdlPvm
  retq
```

```
_ZTV1X:     ; TV == vtable
dq      0,
        _ZTI1X,
        _ZN1XD1Ev,
        _ZN1XD0Ev,
        _ZN1X4funcEv
```

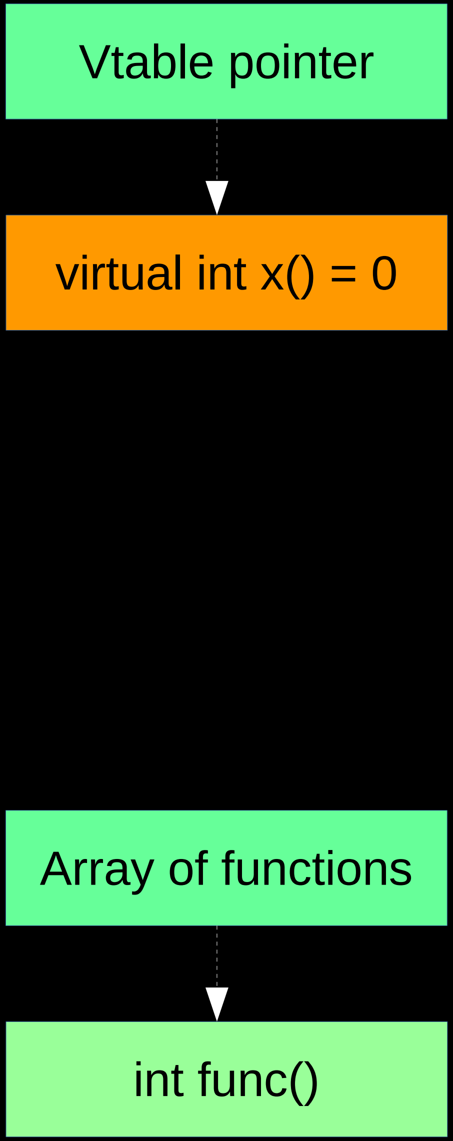
```
_ZTI1X:     ; TI == RTTI info object
dq          _ZTVN10__cxxabiv1
            17__class_type_infoE + 0x10,
            _ZTS1X

_ZTS1X:     ; TS == RTTI class name
db          "1X",0
```

Demo

Can we do this in C++?

- Capture info we need at compile time
- DIY construct the object at run time
- **`reinterpret_cast`** to the intended type
- Use as intended type



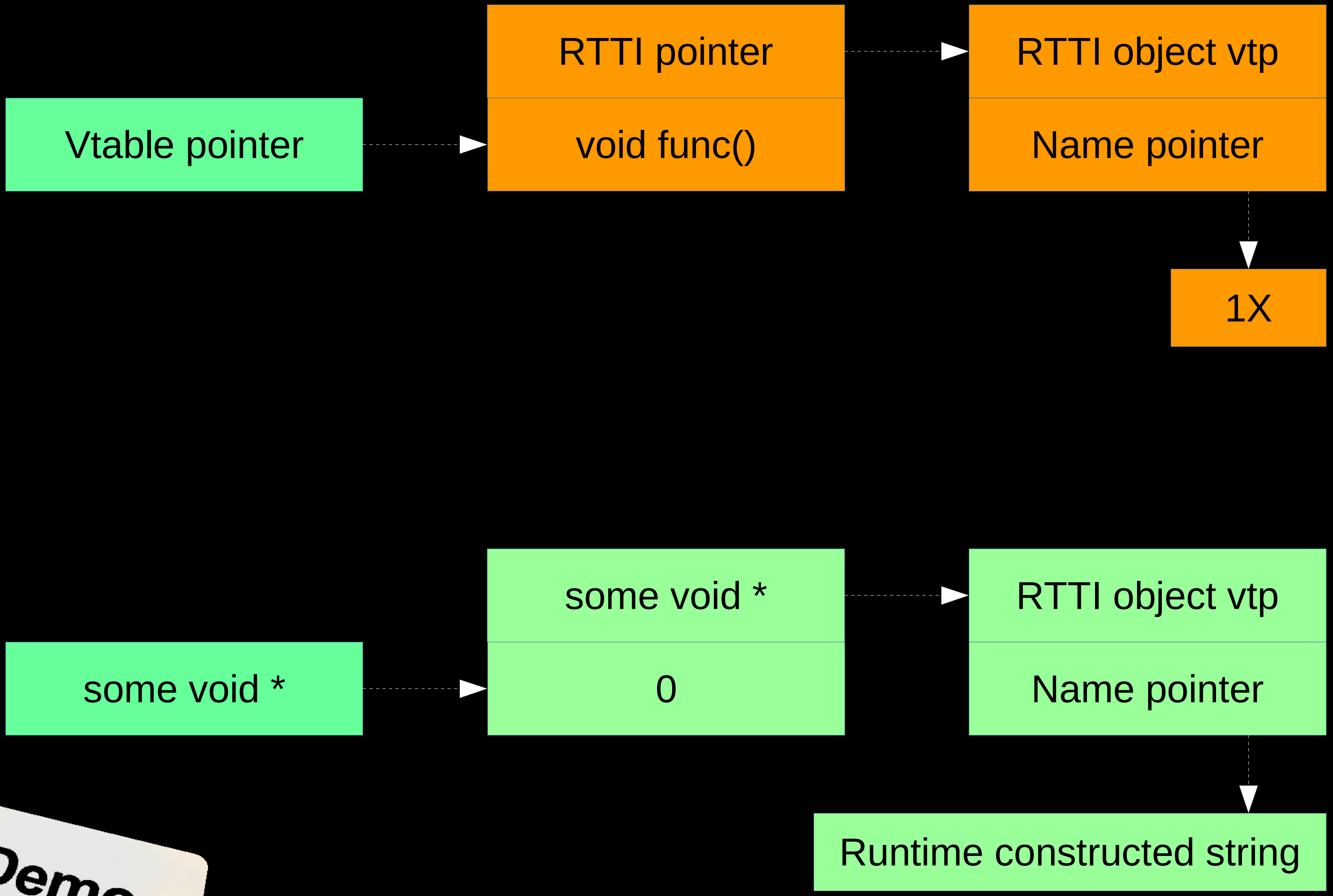
Vtable pointer

virtual int x() = 0

Array of functions

int func()

Demo



Vtable pointer

RTTI pointer

void func()

RTTI object vtp

Name pointer

1X

some void *

some void *

0

RTTI object vtp

Name pointer

Runtime constructed string

Demo

Can we do this in C++?

- No… ?
  - How big is the vtable?
  - We can make it “big enough”

Can we do this in C++?

- No… ?
  - What inheritance graph do we have?
  - Where are the vtables?

  - Put vtables at all possible locations

Can we do this in C++?

- No… ?
  - What kinds of functions are in the vtable?
  - What return values to return?
  - How do we clean up a callee-cleanup stack (Windows) ?

  - We can fill it with functions that only throw

    No return value No argument cleanup

Can we do this in C++?

- No… ?
  - What members to initialize?
  - How to initialize?

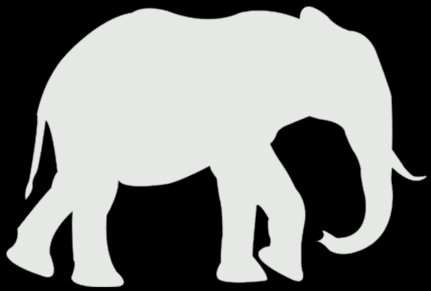
  - Don’t initialize anything
  - Allow the user to request members to be initialized if needed

Can we do this in C++?

- Yes
  - Capture the size
  - Create a large enough vtable
  - Put in place an RTTI object

Can we do this in C++?

- No
  - This is all **undefined behaviour** in C++
  - This is all **defined behaviour** according to the ABI
  - Optimizers can and will avoid conforming to the ABI if it can be done within C++’s scope
    - Don’t use release builds for functional tests
    - Don’t use LTO for functional tests



Advantages

Advantages

- Deleting destructor could… not delete

```
_ZN1XD0Ev:
  ; Defer to regular destructor
  callq  _ZN1XD1Ev
  ; mov    $0x10,%esi
  ; ; Call operator delete
  ; callq  _ZdlPvm
  callq  _ZN10HippoMocks11MarkDeletedEPv
  retq
```

Advantages

- Deleting destructor could… not delete

- Breakpoint or throw exception when you use a dangling pointer
- Halt unit test with a ZombieMockException

Advantages

- Omit base class construction/destruction entirely

- Not possible with normal mocks, can avoid unintended side-effects
- Do watch out for breaking LSP

Advantages

- Test use of an interface without any implementation

- Less code to write to first test

Advantages

- Hook into a DI framework & auto-mock all dependencies

- Makes for very nice testing
- Cannot forget to mock out a dependency
- Strongly encourages good unit testing
- Lowers barrier to entry for new tests

Disadvantages

Disadvantages

- Invokes UB in your test environment

Alternative ways

Alternative ways

- Use macros and ask the user
  - Trompe l’Oeil
  - Google Mock
  - Most others

- More work
- Maintenance issues from DRY

Alternative ways

- Use a script/program to convert header to mock object
  - Google Mock

- Parsing C++ is **very** hard

Alternative ways

- Use a compiler frontend to generate mock classes
  - None that I know of

- Compiler vendor lock-in
- Additional build step

future<C++>

C++23+ dreaming

```
template <typename T>
class mock : public T {
  constexpr {
    for (auto f : $T.functions())
      if (f.is_virtual())
        $reify(f, [this](auto&& args...){
          ...;
        });
  }
};
```

C++23+ dreaming

```
template <typename T>
class mock : public T {
  constexpr {
    for (auto f : $T.functions())
      if (f.is_virtual())
        $reify(f, [this](auto&& args...){
          ...;
        });
  }
};
```

C++23+ dreaming

- No more UB
- Interface is wholly unchanged
- Works very well with all forms of optimizers
- If you don’t link in the mock user, you get conditional devirtualization

Not just for mocking

```
template <typename T>
class proxy : public T {
  constexpr {
    for (auto f : $T.functions())
      static_assert(f.is_virtual());
      $reify(f, [this](auto&& args...){
        remote.call(get_func_id(f),
          args...);
      });
  }
};
```

Not just for mocking

```
template <typename T>
class logger : public T {
  constexpr {
    for (auto f : $T.functions())
      static_assert(f.is_virtual());
      $reify(f, [this](auto&& args...){
        log(args...);
        inner.$name(f)(args...);
      });
  }
  logger(T& inner) {...}
};
```

Legacy code bases

- Not designed for testing
- Large
- Unknown side-effects
- Not very well tested
- Giant risk of breakage when changed

How to start unit testing

- Create a unit test for existing code

- Refactor existing code

How to start unit testing

- Create a unit test for existing code
  - But that’s not possible
  - Code uses C / free functions directly
  - Extract interfaces

- Refactor existing code
  - But you can’t! Too much risk!
  - First put code under test, then refactor & retest
  - But how can I write a test without interfaces?

Solution

- Just mock the free functions

Solution

- Use macros to replace at compile time, to use a different function
  - Not 100% stable
  - One build per replacement set
  - Ugly macro use
  - Does not scale to interfaces

Solution

- Use macros to replace at compile time, to use an invocable object
  - Not 100% stable
  - One build per replacement set
  - Ugly macro use
  - Not C compatible

  - Does not scale to interfaces

Solution

- Replace the function at link or load time
  - Intentional ODR violation
    - Depending on link order may not work / break
  - LD_PRELOAD_PATH
    - Platform specific
    - Hidden and evil

Solution

- Replace the function itself at run time

Solution

- Replace the function itself at run time

```
0000000000000000 <myFunction>:
   0: 48 83 ec 08             sub    $0x8,%rsp
   4: 48 8b 7e 08             mov    0x8(%rsi),%rdi
   8: e8 00 00 00 00          callq  d <myFunction+0xd>
   d: 48 85 c0                test   %rax,%rax
  10: 74 38                   je     4a <myFunction+0x4a>
```

Solution

- Replace the function itself at run time

```
0000000000000000 <myFunction>:
   0: e9 ?? ?? ?? ??       jmp    <myMockFunction>
   5: .. 8b 7e 08          <corrupt>
   8: e8 00 00 00 00       callq  d <myFunction+0xd>
   d: 48 85 c0             test   %rax,%rax
  10: 74 38                je     4a <myFunction+0x4a>
```

Solution

- Replace the function itself at run time

```
byte* pMalloc = (byte*)&malloc;
pMalloc[0] = 0xE9;
...
```

Solution

- Replace the function itself at run time

- Memory protection error
  - Cannot write to code section

- Solution: Ask nicely.

Solution

```
uint8_t* pMalloc = (uint8_t*)&malloc;

mprotect((intptr_t)pMalloc & ~0xFFF,
    0X2000,
    PROT_READ | PROT_WRITE | PROT_EXEC);

pMalloc[0] = 0xE9;

…
```

Solution

This works fine

With three caveats

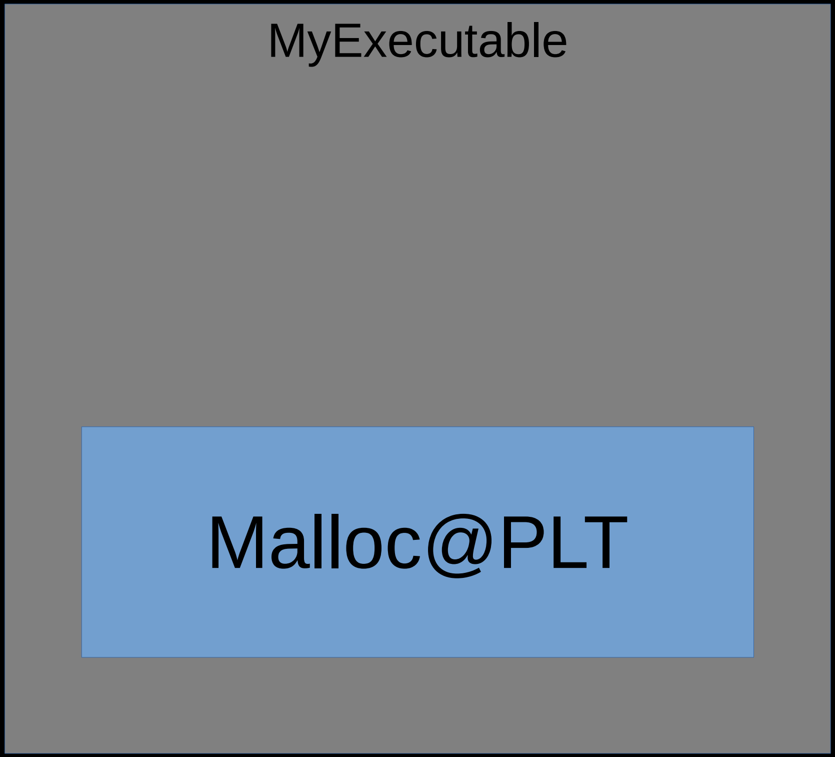
Solution

- #1: The replacement code has to fit
  - X86 jump is 5 bytes, should nearly always fit
  - ARM jump is 12 bytes, should nearly always fit
  - X86-64 jump is 14 bytes, may not always fit...

Solution

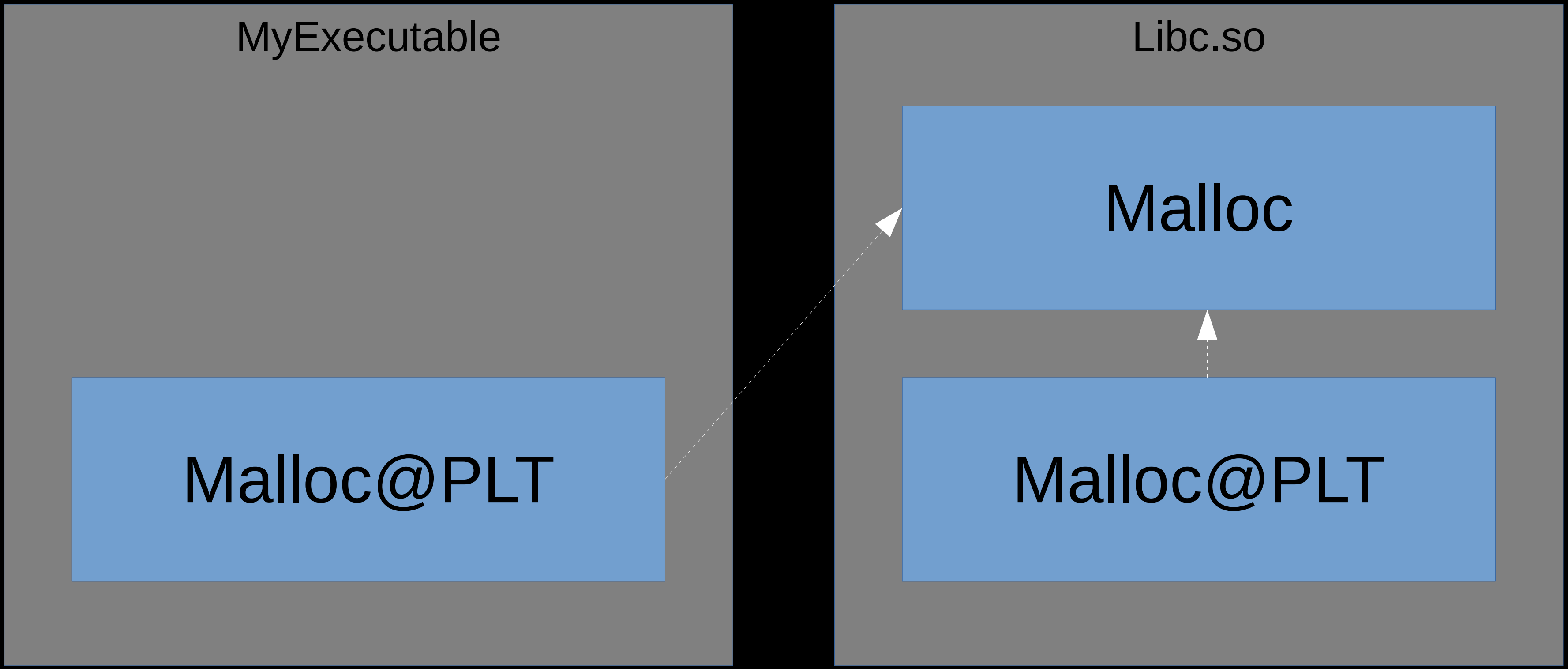
#2: My malloc is not your malloc

- In fact, your malloc is not necessarily your malloc



MyExecutable

Malloc@PLT



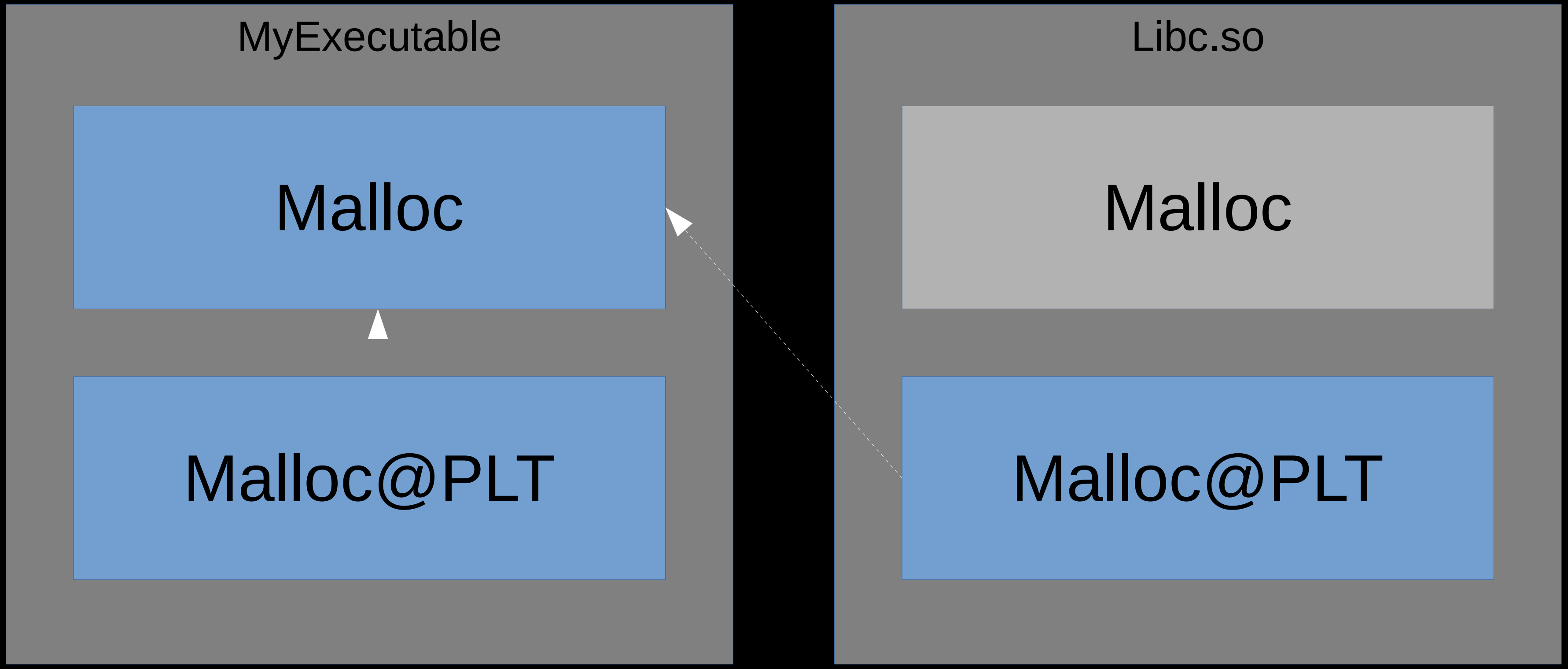
MyExecutable

Malloc@PLT

Libc.so

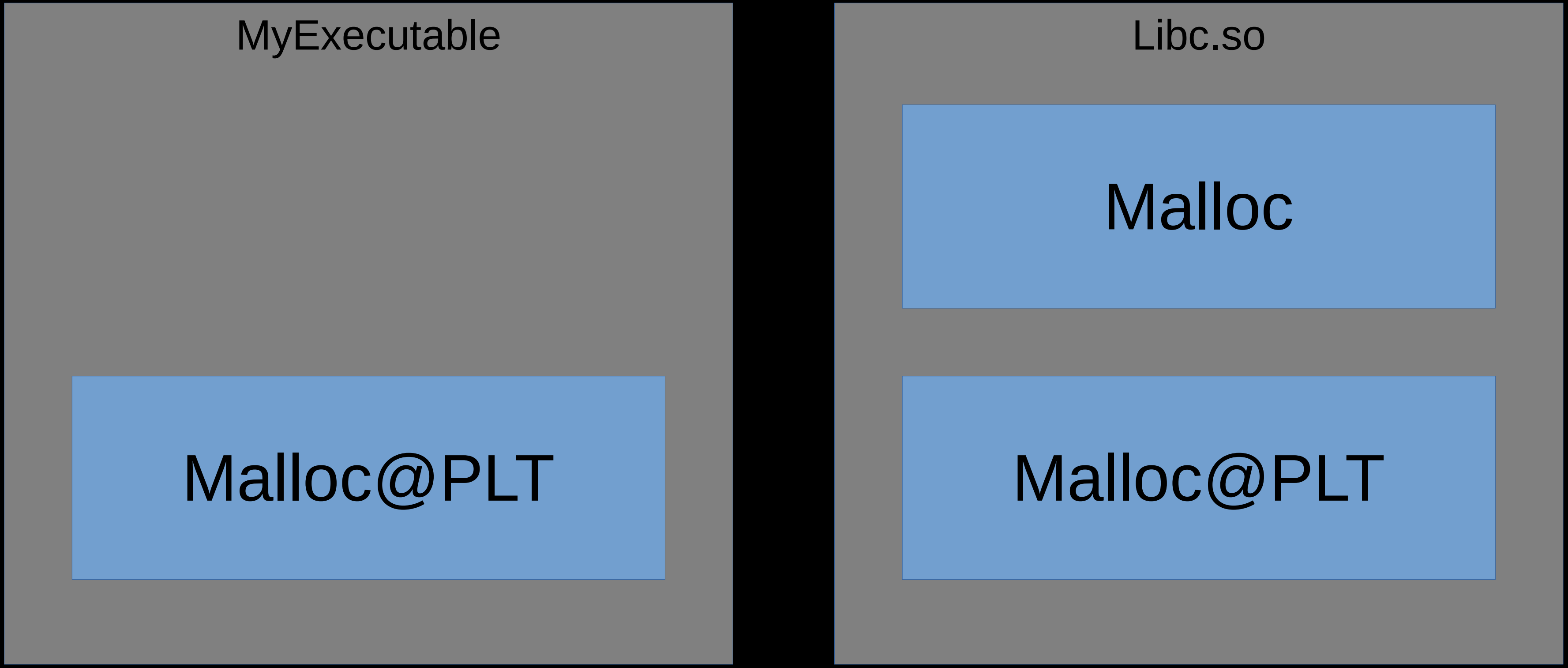
Malloc

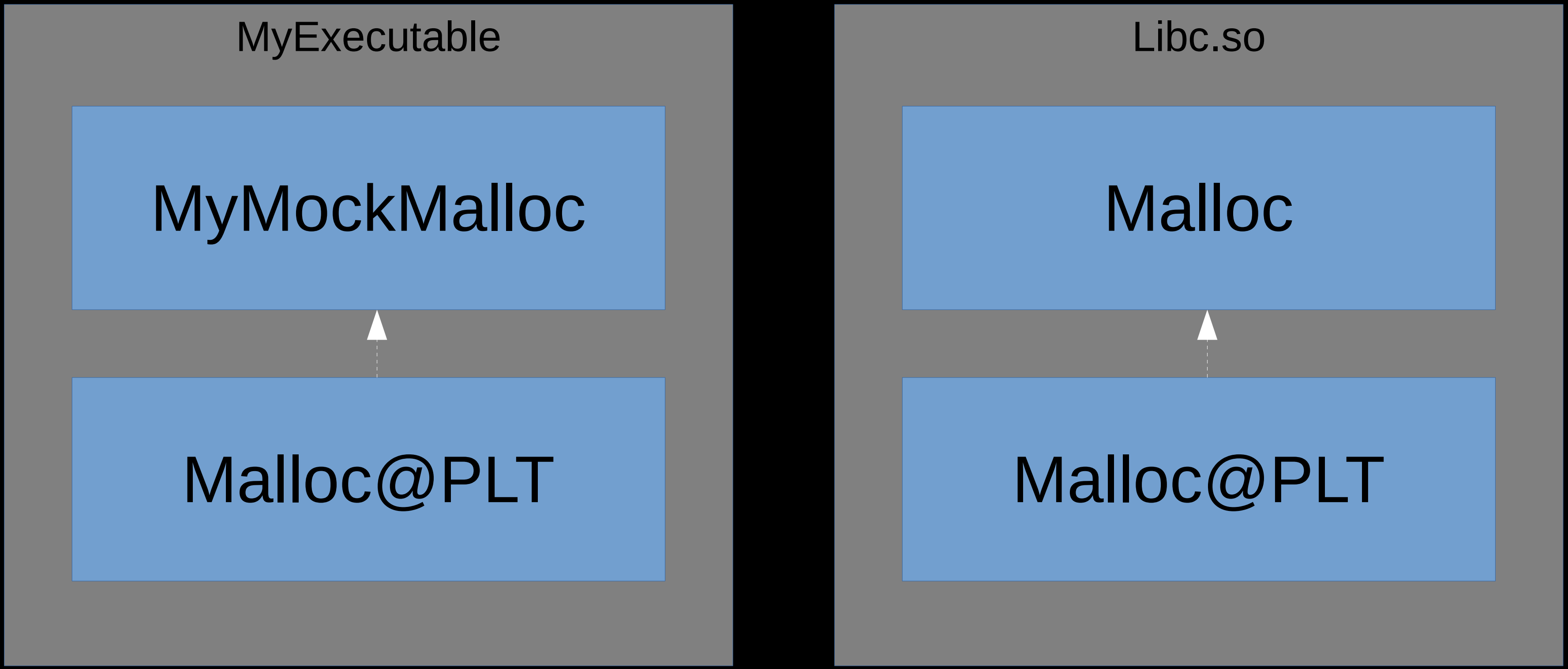
Malloc@PLT


MyExecutable
Malloc
Malloc@PLT
Libc.so
Malloc
Malloc@PLT

- So which is **`&malloc`** ?

- … it’s actually any one of these.




MyExecutable
MyMockMalloc
Malloc@PLT
Libc.so
Malloc
Malloc@PLT

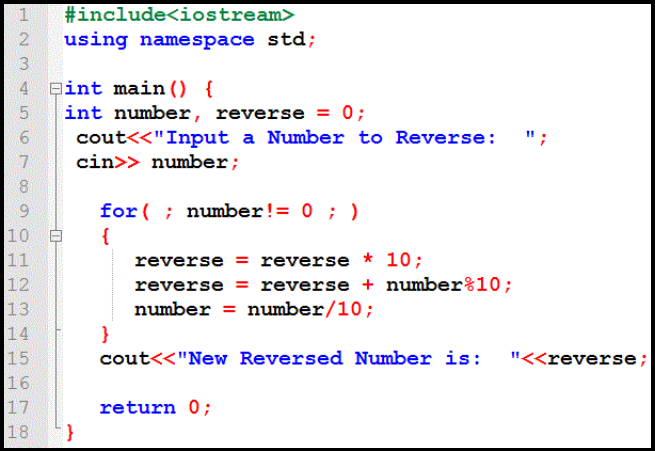
Solution

- #3: My **`f()`** may not actually be my **`f()`**
  - Inlined functions

- Good for testing
- Not 100% replacement

How much to test?

- More tests ensures
  - Less chance of breakage
  - More work to change (resistant to change)
- Less tests ensures
  - The major cases work
  - Cornercases can break undetected
  - Code is modifiable



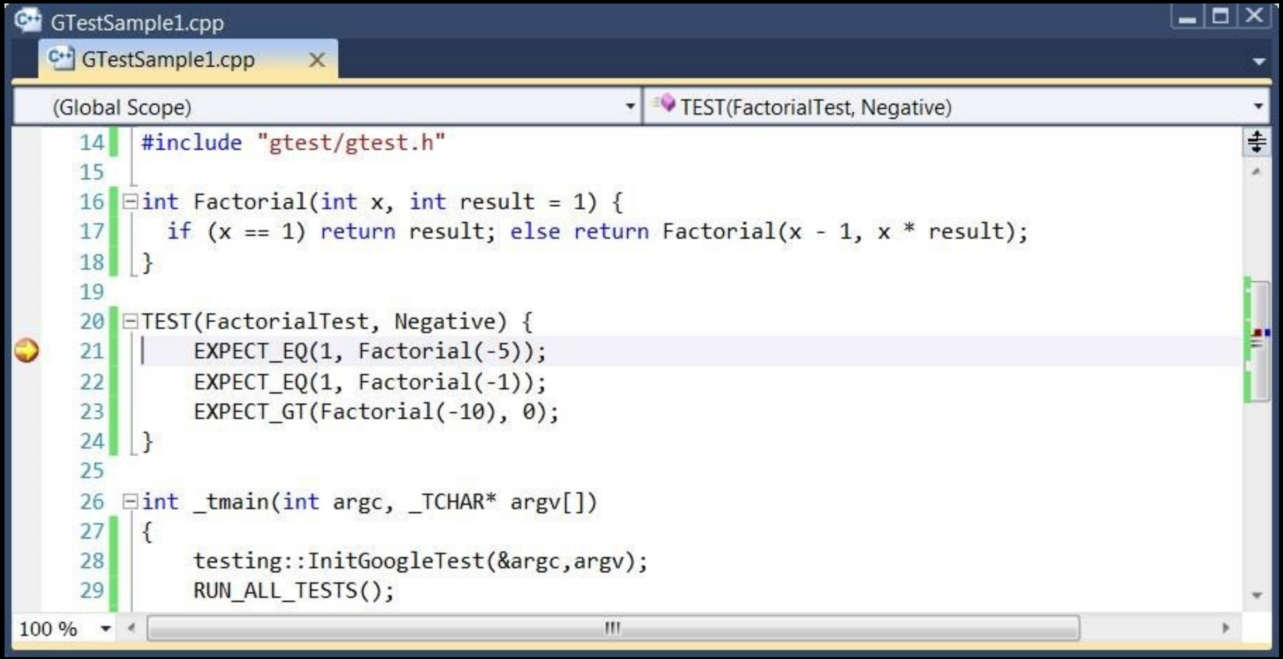
```
#include<iostream>
using namespace std;

int main() {
int number, reverse = 0;
 cout<<"Input a Number to Reverse:  ";
 cin>> number;

   for( ; number!= 0 ; )
   {
      reverse = reverse * 10;
      reverse = reverse + number%10;
      number = number/10;
   }
   cout<<"New Reversed Number is:  "<<reverse;

   return 0;
}
```

==



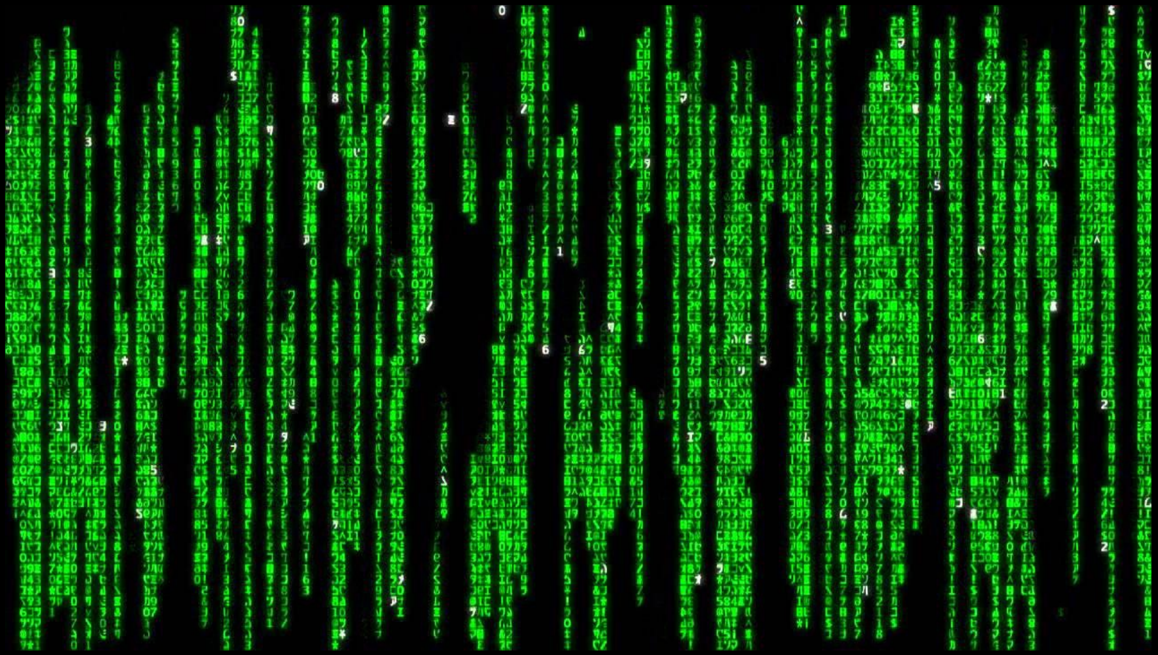
```cpp
#include "gtest/gtest.h"

int Factorial(int x, int result = 1) {
  if (x == 1) return result; else return Factorial(x - 1, x * result);
}

TEST(FactorialTest, Negative) {
    EXPECT_EQ(1, Factorial(-5));
    EXPECT_EQ(1, Factorial(-1));
    EXPECT_GT(Factorial(-10), 0);
}

int _tmain(int argc, _TCHAR* argv[])
{
    testing::InitGoogleTest(&argc,argv);
    RUN_ALL_TESTS();
```

==



==

When to use mocks

- Write code “Lego-style”
  - Small components that are well tested
  - Large components only assemble small components
- .. in effect, try not to
  - More complicated tests
  - Larger tests

Throwing destructor or not?

- Mock context should check all pending calls on function exit → **`VerifyAll()`**
  - Easy to forget, no way to notice
- Mock errors detected at function exit are only relevant if no error already happened
  - Existing error trumps missing calls
- Why not make the destructor run the check at all times?

Questions