

CS 105C: Lecture 11

Last Time...

***“ What you don’t use, you don’t pay for.
And further: What you do use, you
couldn’t hand code any better.***

Heterogeneous Collections

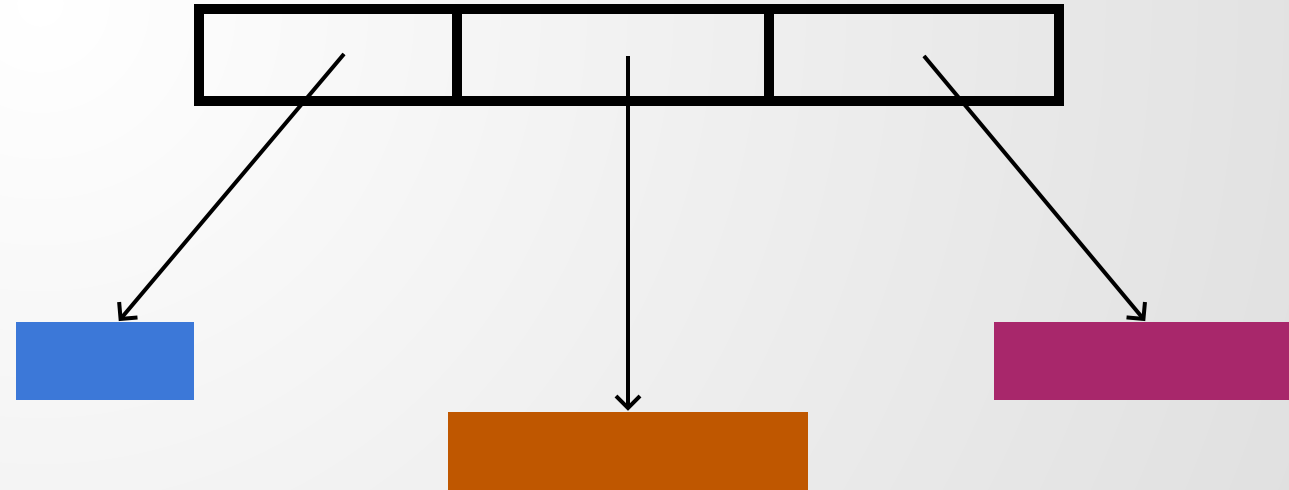
Pay for them even if you don't use them!

```
a = [1, "1", 1.0]
```

$O(N)$ indexing



Indirect storage



Other cases of pay-without-use

Green Threads

User-level threads managed by the language runtime.

Can be left unused, but they have to be included in the language runtime regardless.

Garbage Collection

Garbage-collected systems must have all their memory in garbage collection.

Incur the cost of running gc even if we know exactly when all memory can be freed.

C++ Zero-Cost Abstractions

`std::unique_ptr`

`std::vector`

Compiler	Optimization	new	std::shared_ptr	std::make_shared	std::unique_ptr	std::make_unique
GCC	no	3.03	13.48	30.47	8.74	9.09
GCC	yes	3.03	6.42	3.24	3.07	3.04
cl.exe	no	8.79	25.17	18.75	11.94	13.00
cl.exe	yes	7.42	17.29	9.40	7.58	7.68

Element access is identical assembly code to raw array access, because some parts can be done at compile-time.

Speed difference: tens of seconds in 24 hours.

Vector copy code is 15% *faster* than handwritten naive copy code!

Questions!

Q: Are shared pointers zero cost?

A: What do we mean by zero cost?

Under the definition that Bjarne Stroustrup gives us, they **are** zero cost.

But shared pointers are **not** free! (Not even close!)

Compiler	Optimization	new	std::shared_ptr	std::make_shared	std::unique_ptr	std::make_unique
GCC	no	3.03	13.48	30.47	8.74	9.09
GCC	yes	3.03	6.42	3.24	3.07	3.04
cl.exe	no	8.79	25.17	18.75	11.94	13.00
cl.exe	yes	7.42	17.29	9.40	7.58	7.68

Q: Why do people write programs in other languages if C++ is so fast?

A:

```
~/tmp/cppptest  
> ./a.out  
[1] 77202 segmentation fault ./a.out
```

A:

From Lecture 0: while it is not true that C++ is a language for experts only, it is a language that requires discipline and knowledge from its users in order to be effective.

A:

Performance matters. But it doesn't always matter where we think it does.

CS 105C: Lecture 11


More Zero-Cost Abstractions and Undefined Behavior

Exceptions

A not-quite-zero cost abstraction

What are the rules about where exceptions can occur?

```
1 void operation_1();
2 void operation_2();
3 void operation_3() noexcept;
4
5 int main(){
6     operation_1();
7     operation_2();
8     operation_3();
9 }
```



Pretty much anywhere!

(In C++17, functions labeled 'noexcept' cannot throw)

**How should we implement
exceptions?**

The compiler somehow needs to insert extra code into the compiled program to implement exceptions

Exactly what code it inserts will depend on how we implement

A First Attempt

Just check every instruction!

```
1 int main(){
2     try{
3         op1();
4     }catch(Exception& e){
5         handler1(e);
6     }
7
8     try{
9         op2();
10    }
11    catch(Exception& e){
12        handler2(e);
13    }
14
15    op3();
16 }
```



```
1 Exception_Structure x;
2
3 int main(){
4     op1();
5     if(x.exception_happened){
6         handler1(x);
7     }
8
9     op2();
10    if(x.exception_happened){
11        handler2(x);
12    }
13
14    op3();
15    if(x.exception_happened){
16        terminate();
17    }
18 }
```

Exception Handling

Do we have to pay a cost even if we don't actually throw/catch any exceptions?

Runtime cost

Yes. In the worst case, we have to insert a branch *every other instruction*, which **at least doubles** the runtime of the code

Compile time cost

Yes: we have to insert all these additional exception checks--we may also want to optimize by trying to remove some checks (which could be costly!)

Code Complexity Cost

In source code? No. In binary code? Oh *hell* yes. You won't be able to *tell* what the code is doing underneath all those branches!

C++ requires that an exception triggers!

Even though we're not catching exceptions, we *still* have to include these branches, because we cannot ignore any exceptions.

```
1 int main(){
2     try{
3         op1();
4     }catch(Exception& e){
5         handler1(e);
6     }
7
8     try{
9         op2();
10    }
11    catch(Exception& e){
12        handler2(e);
13    }
14
15    op3();
16 }
```

```
1 Exception_Structure x;
2
3 int main(){
4     op1();
5     if(x.exception_happened){
6         handler1(x);
7     }
8
9     op2();
10    if(x.exception_happened){
11        handler2(x);
12    }
13
14    op3();
15    if(x.exception_happened){
16        terminate();
17    }
18 }
```

Exceptions

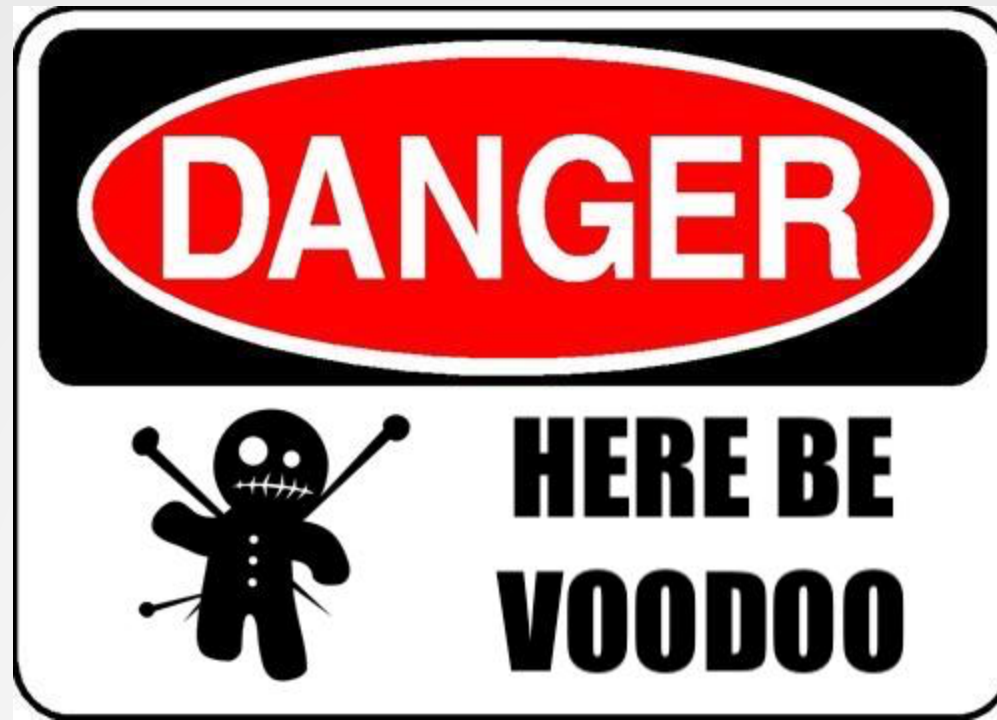
Now do it again, but properly this time!

Goals

We would like an exception handling framework that is:

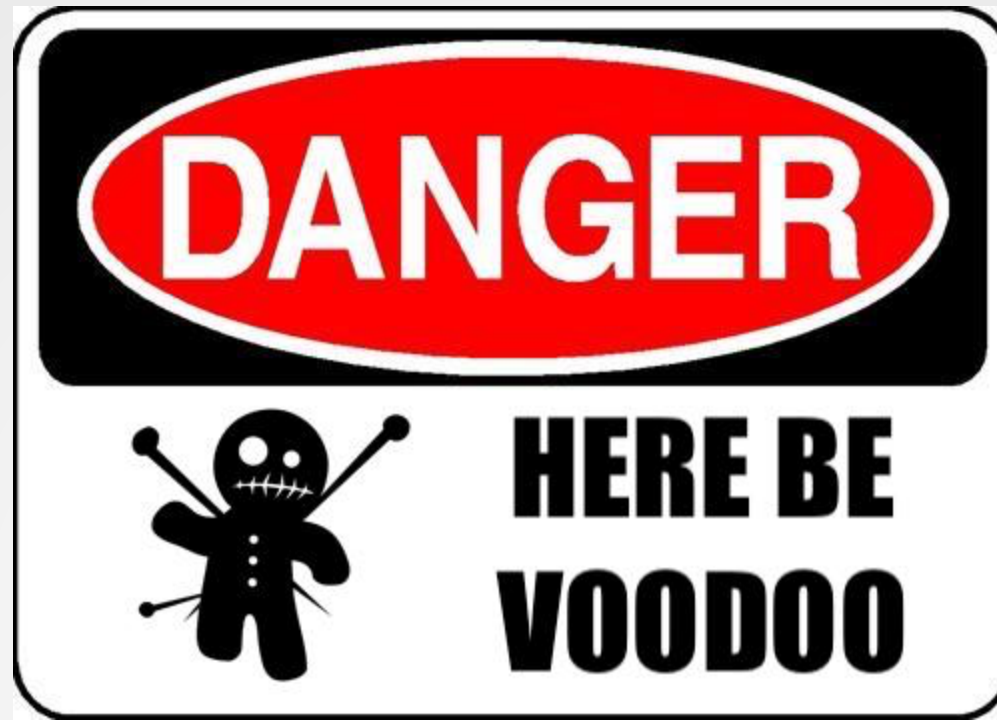
- Fast in the case where no exception is thrown
- Not excessively slow when an exception is actually thrown

But first...a warning!



**We are about to enter an area of C++ that is
very implementation-dependent and often
undocumented!**

The interfaces used for exception handling are defined not by the C++ Standard or the API, but the **ABI**. This means that every compiler could potentially implement its own exception techniques



The implementation we will study is based on x86 and GCC, documented partially on [this post](#).

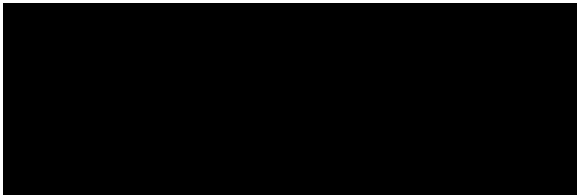

There is *no guarantee* that your compiler implements things the same way!

General Idea

Instead of checking if an exception occurred at every instruction, we should have some routine which runs when an exception is thrown.

This routine will be responsible for executing the exception.

Before

```
1 operation_1();
2 
3
4
5 operation_2();
6 
7
8
```

After

```
1 void operation_1(){
2     if(bad_thing){
3         __cxa_throw();
4     }
5 }
6
7 void operation_2(){
8     if(bad_thing){
9         __cxa_throw();
10    }
11 }
```

But there's a problem...

How does the exception executor know e.g. whether it's been invoked inside of a try-catch block? How does it know which catch to use?

```
1 int main(){
2   try{
3     op1();
4   }catch(Exception& e){
5     handler1(e);
6   }
7
8   try{
9     op2();
10  }
11  catch(Exception& e){
12    handler2(e);
13  }
14
15  op3();
16 }
```

Handle exception?
Call terminate?

```
1 void operation1(){
2   if(bad_thing){
3     __cxa_throw();
4   }
5 }
6
7 void operation2(){
8   if(bad_thing){
9     __cxa_throw();
10  }
11 }
```

Solution: Record state!

At the start of a catch block, call `__cxa_begin_catch` and call `__cxa_end_catch` at the end--this will install information in some global area about what exception handlers are available at the time.

```
1 int main(){
2     try{
3         op1();
4     }catch(Exception& e){
5         handler1(e);
6     }
7
8     try{
9         op2();
10    }
11    catch(Exception& e){
12        handler2(e);
13    }
14
15    op3();
16 }
```



```
1 int main(){
2     __cxa_begin_catch(handler1);
3     op1();
4     __cxa_end_catch(handler1);
5
6     __cxa_begin_catch(handler2);
7     op2();
8     __cxa_end_catch(handler2);
9
10    op3();
11 }
```

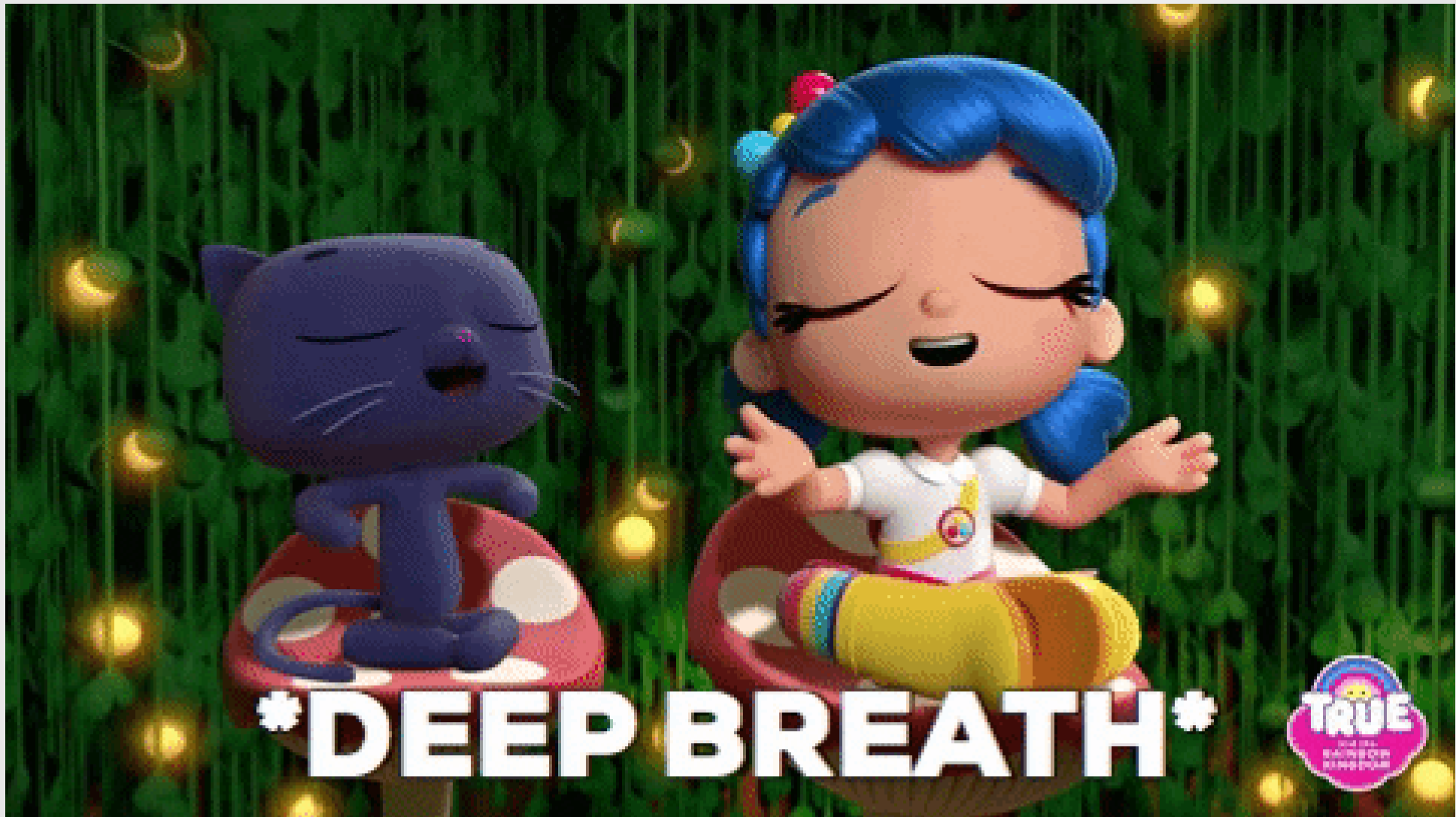

State Records

The state records stored by `__cxa_begin_catch` can be stored in two different locations:

- In a global table
- In the stack itself

GCC on x86 platforms uses the former, variants on ARM use the latter. The choice of location has a minor-to-moderate impact on runtime performance.

The state includes things like "which exception handlers are active", "where control for those handlers jumps to", and "active variables that need to be cleaned up."



DEEP BREATH



What's the overall picture?

1. When we enter a try/catch block, we use `__cxa_begin_catch()` to record information about what to do if an exception triggers.
2. If an exception is triggered, `__cxa_throw()` will use this information to decide which exception handlers to run.

We've actually omitted about 12 steps here, but that should be enough for the basics

An Example

```
1 void op1(){
2     try{
3         MyClass class2;
4         op2();
5     }catch (DumbException& e){
6         handler2(e);
7     }
8 }
9
10
11 int main(){
12     try{
13         MyClass class1;
14         op1();
15     }catch (SillyException& e){
16         handler1(e);
17     }
18 }
```

→ can catch: DumbException
resume execution at: handler2
destroy stack variables: none

→ can catch: SillyException
resume execution at: handler1
destroy stack variables: class1

```

1 void op1(){
2     try{
3         MyClass class2;
4         op2();
5     }catch (DumbException& e){
6         handler2(e);
7     }
8 }
9
10
11 int main(){
12     try{
13         MyClass class1;
14         op1();
15     }catch (SillyException& e){
16         handler1(e);
17     }
18 }

```

Exception Type	Resume At	Destroy Var

```

1 void op1(){
2     try{
3         MyClass class2;
4         op2();
5     }catch (DumbException& e){
6         handler2(e);
7     }
8 }
9
10
11 int main(){
12     try{
13         MyClass class1;
14         op1();
15     }catch (SillyException& e){
16         handler1(e);
17     }
18 }

```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1

```
1 void op1(){
2     try{
3         MyClass class2;
4         op2();
5     }catch (DumbException& e){
6         handler2(e);
7     }
8 }
9
10
11 int main(){
12     try{
13         MyClass class1;
14         op1();
15     }catch (SillyException& e){
16         handler1(e);
17     }
18 }
```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1


```

1 void op1(){
2     try{
3         MyClass class2;
4         op2();
5     }catch (DumbException& e){
6         handler2(e);
7     }
8 }
9
10
11 int main(){
12     try{
13         MyClass class1;
14         op1();
15     }catch (SillyException& e){
16         handler1(e);
17     }
18 }

```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1

```

1 void op1(){
2   try{
3     MyClass class2;
4     op2();
5   }catch (DumbException& e){
6     handler2(e);
7   }
8 }
9
10
11 int main(){
12   try{
13     MyClass class1;
14     op1();
15   }catch (SillyException& e){
16     handler1(e);
17   }
18 }

```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1

```

1 void op1(){
2   try{
3     MyClass class2;
4     op2();
5   }catch (DumbException& e){
6     handler2(e);
7   }
8 }
9
10
11 int main(){
12   try{
13     MyClass class1;
14     op1();
15   }catch (SillyException& e){
16     handler1(e);
17   }
18 }

```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1
DumbException	handler2	class2

```

1 void op1(){
2     try{
3         MyClass class2;
4         op2();
5     }catch (DumbException& e){
6         handler2(e);
7     }
8 }
9
10
11 int main(){
12     try{
13         MyClass class1;
14         op1();
15     }catch (SillyException& e){
16         handler1(e);
17     }
18 }

```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1
DumbException	handler2	class2

```

1 void op1(){
2     try{
3         MyClass class2;
4         op2();
5     }catch (DumbException& e){
6         handler2(e);
7     }
8 }
9
10
11 int main(){
12     try{
13         MyClass class1;
14         op1();
15     }catch (SillyException& e){
16         handler1(e);
17     }
18 }

```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1
DumbException	handler2	class2

op2 throws a SillyException!

```

1 void op1(){
2     try{
3         MyClass class2;
4         op2();
5     }catch (DumbException& e){
6         handler2(e);
7     }
8 }
9
10
11 int main(){
12     try{
13         MyClass class1;
14         op1();
15     }catch (SillyException& e){
16         handler1(e);
17     }
18 }

```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1
DumbException	handler2	class2

op2 throws a SillyException!

We look through our handlers from innermost scope to outermost, trying to find a match. If a match is found, execute!

```

1 void op1(){
2   try{
3     MyClass class2;
4     op2();
5   }catch (DumbException& e){
6     handler2(e);
7   }
8 }
9
10
11 int main(){
12   try{
13     MyClass class1;
14     op1();
15   }catch (SillyException& e){
16     handler1(e);
17   }
18 }

```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1
DumbException	handler2	class2

op2 throws a SillyException!

We look through our handlers from innermost scope to outermost, trying to find a match. If a match is found, execute!

```

1 void op1(){
2     try{
3         MyClass class2;
4         op2();
5     }catch (DumbException& e){
6         handler2(e);
7     }
8 }
9
10
11 int main(){
12     try{
13         MyClass class1;
14         op1();
15     }catch (SillyException& e){
16         handler1(e);
17     }
18 }

```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1 ←
DumbException	handler2	class2

op2 throws a SillyException!

We look through our handlers from innermost scope to outermost, trying to find a match. If a match is found, execute!


```

1 void op1(){
2     try{
3         MyClass class2;
4         op2();
5     }catch (DumbException& e){
6         handler2(e);
7     }
8 }
9
10
11 int main(){
12     try{
13         MyClass class1;
14         op1();
15     }catch (SillyException& e){
16         handler1(e);
17     }
18 }

```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1 ←
DumbException	handler2	class2

op2 throws a SillyException!

I found a match! We need to execute the cleanup routines of all blocks below us (destroy class2 + class1), then jump to the handler.

```

1 void op1(){
2     try{
3         MyClass class2;
4         op2();
5     }catch (DumbException& e){
6         handler2(e);
7     }
8 }
9
10
11 int main(){
12     try{
13         MyClass class1;
14         op1();
15     }catch (SillyException& e){
16         handler1(e);
17     }
18 }

```

Exception Type	Resume At	Destroy Var
SillyException	handler1	class1 ←
DumbException	handler2	class2

op2 throws a SillyException!

I found a match! We need to execute the cleanup routines of all blocks below us (destroy class2 + class1), then jump to the handler.

Is this zero-cost?

Do we pay if we don't actually **catch** any exceptions? (Throwing might happen anyways)

Runtime cost

Very nearly. There is no visible cost if no exception is actually thrown...but the cleanup code occupies space in the instruction cache, potentially slowing down branches a little.

Compile time cost

A small cost. We need to add some of the extra functions and cleanup information.

Code Complexity Cost

A complexity cost to implementing the standard library. Otherwise, not an unthinkably large cost.

Is this zero-cost?

Could you write this better?

...depends on your error handling needs. If you don't need the full power of exceptions (arbitrary distance from throw to catch, pre-emption of executing code), you can probably do better.

Are C++ exceptions zero-cost?

No.

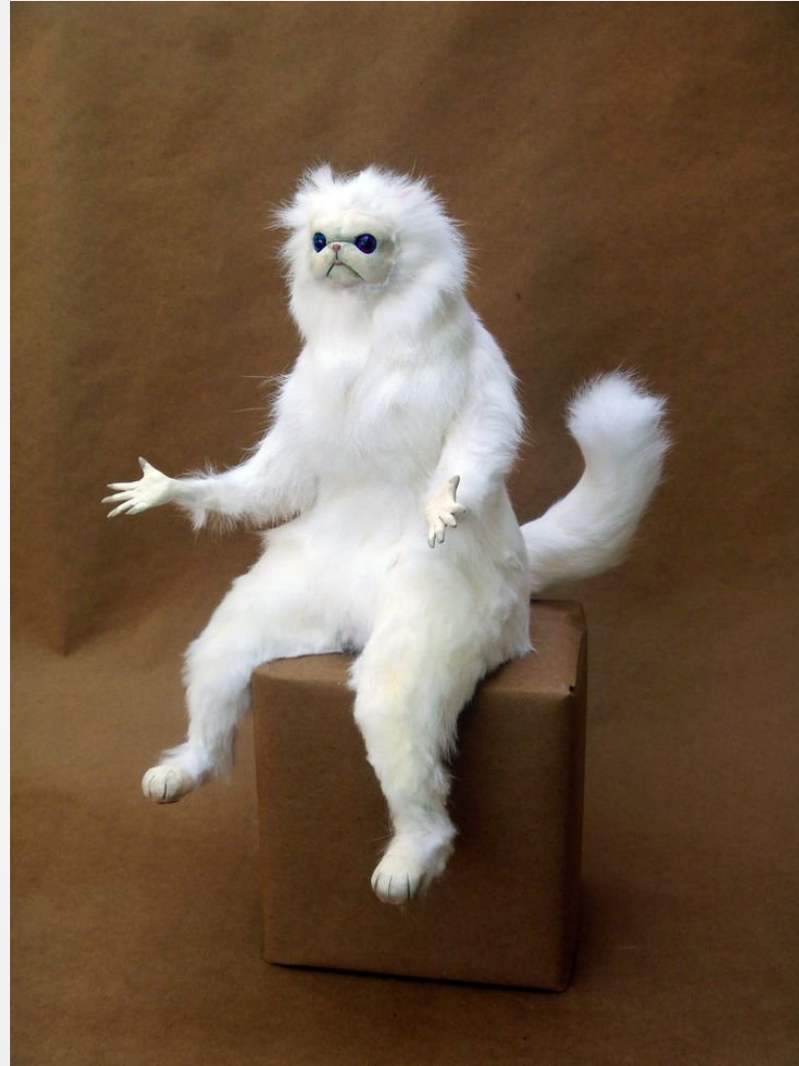
But they turn out to be useful enough in a variety of situations (and close enough to zero-cost) that they're worth including anyways.

Undefined Behavior

Recall from Lecture 1 that Undefined Behavior is triggered whenever any of the **193 conditions listed here occur.**

(Actually, that's just in C. I don't know if anyone has ever tried to list all the UB conditions for C++ in one spot--there's a lot!)

What happens when UB is triggered?



C++ did not spring into the world fully designed

Somebody thought long and hard about what should happen in certain cases, and came to the conclusion that the compiler should be allowed to do **anything it wants to do!**

Why did they come to this conclusion?

The fundamental answer is "speed"

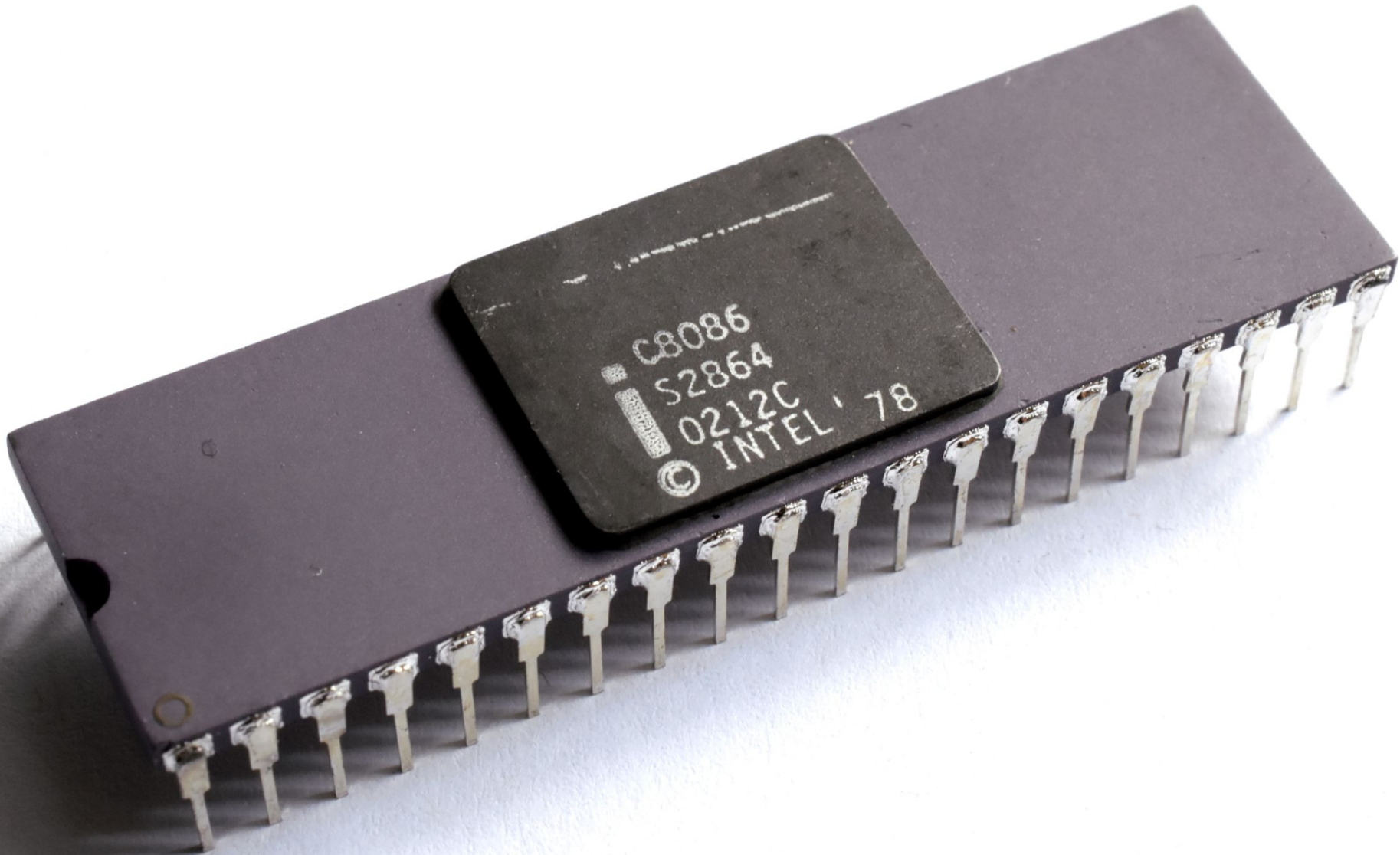
We'll look at three examples of UB to help us understand this a little better:

- Shifting by more than a register width
- Dereferencing a NULL pointer
- Signed integer overflow

Shift Width Overflow

In C++, shifting by more than the width of the register is undefined behavior.

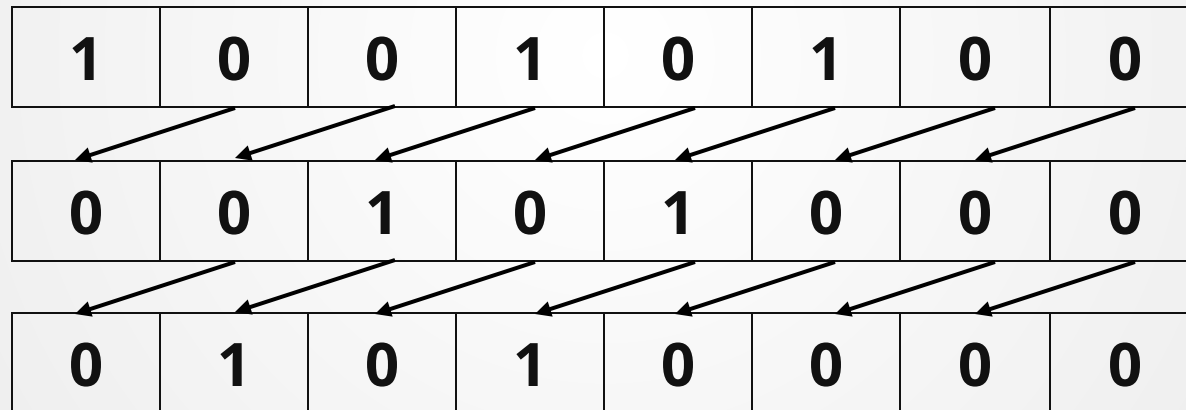
```
1  int32_t x = 5;  
2  x <<= 31; // Just fine!  
3  x <<= 32; // Undefined!
```



8086 Shift Instructions

The 8086 did not have hardware to carry out an arbitrary shift operation.

Instead, it would shift your register one position at a time, using one clock cycle per shift.



You could provide an 8-bit register as the shift amount.

8086 Shift Instructions

You could provide an 8-bit register as the shift amount.

So...you could shift up to 255 times, using 1 shift per clock cycle. Meaning one instruction could take 255 cycles.

Intel later realized this was a terrible idea, and every single x86 processor since then has masked shifts to the lower 5 bits (effectively putting a cap of 31 on the shift amount).

But now we have a problem!

Shifting too much?

```
1 int x = 173, y = 33;  
2 x <<= y;
```

On 8086

0	0	1	0	0	0	0	1
---	---	---	---	---	---	---	---

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

0	0	1	0	0	0	0	1
---	---	---	---	---	---	---	---

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

Shift Register

Mask

Shift Amount

Final value of x

On 80286

0	0	1	0	0	0	0	1
---	---	---	---	---	---	---	---

0	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

0	1	0	1	1	0	1	0
---	---	---	---	---	---	---	---

Same Instruction, Different Outcome

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

Final value of x

0	1	0	1	1	0	1	0
---	---	---	---	---	---	---	---

The same shift instruction results in different outcomes on different processors!

But wait, it gets worse!

Some CPUs treat anything greater than a 31-bit shift as a zero shift (not implementing masking like the 80286).

What can the compiler do?

There are only two things the compiler can do to mitigate this:

Check the result of *every* shift operation for overshifts, and insert fixup code if this occurs.

Pros:

- Consistent behavior across platforms

Cons:

- Very, very slow, with no opportunity to improve performance

Declare that if you shift by too much, you're on your own.

Pros:

- Speedy!

Cons:

- Inconsistent

NULL Pointer Dereference

**Dereferencing a NULL pointer in C++ is
not guaranteed to result in a segfault!**

NULL Pointer Dereference

What happens when you dereference the pointer referring to address 0 on different CPUs?

Architecture	Behavior
x86-64, 64-bit mode	Illegal Page Fault (segfault)
x86-64, real mode	Completely legal!
PDP-11	Always contains value zero.
Other CPUs	Access memory-mapped I/O

...but NULL doesn't even have to be zero! The standard just defines it as a pointer which is "different from a pointer to any object or function"

What can the compiler do?

We really only have two choices here:

- Check the value of every pointer to see if it is NULL on every single access.
- Declare that we don't know what happens if you dereference NULL.

Integer Overflow

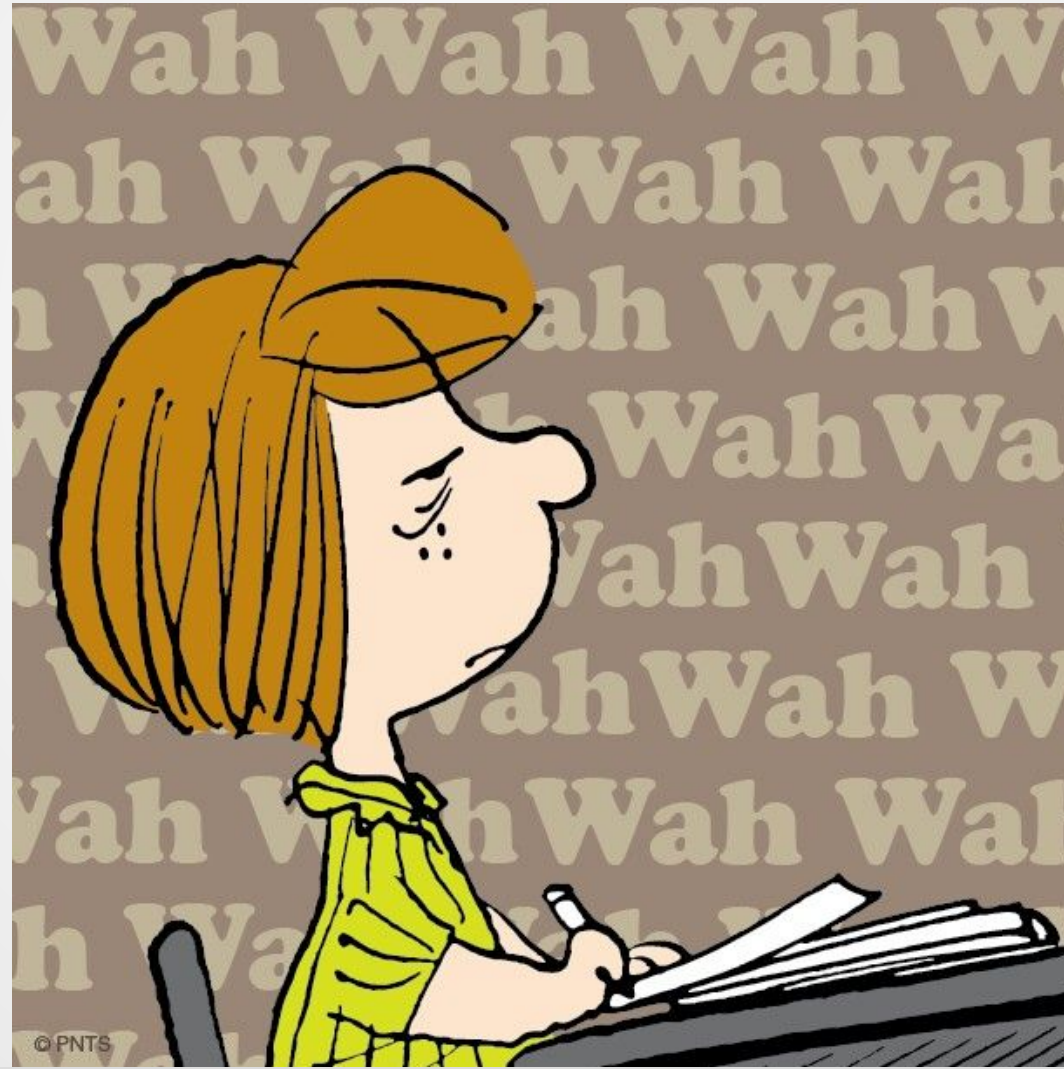
Integer overflow doesn't behave the same on all CPUs.

Most CPUs use twos-complement.

Some use ones-complement.

That one wacky Russian ternary computer
doesn't even use binary representations.

Either have to do runtime checks or
declare behavior to be undefined



Optimizing on Integer Overflow UB

The "as-if" rule

A C++ compiler may transform the program in any way it likes, including ways that break the rules of the standard, as long as all observable behavior of the program is **as if** the rules were obeyed.

Optimizing Division

```
1 int x = foo();  
2 if (x > 0) {  
3     int y = x + 5;  
4     int z = y / 4;  
5 }
```



Division is slow!

What we'd like to be able to do: optimize
division statements into bitshifts

```
1 int x = foo();  
2 int y = x + 5;  
3 int z = y / 4;
```

Is this legal?

```
1 int x = foo();  
2 int y = x + 5;  
3 int z = y >> 2;
```

NO!

`-1 >> 2 == 4611686018427387903`

```
1 int x = foo();
2 if (x > 0){
3     int y = x + 5;
4     int z = y / 4;
5 }
```

Is this legal?

```
1 int x = foo();
2 if (x > 0){
3     int y = x + 5;
4     int z = y >> 2;
5 }
```

Compiler Reasoning:

- At start of if-statement, x is in the range [1, INT_MAX]
- That means y is in the range [6, INT_MAX] (???)
- So y is positive on line 4, so we can do the transformation

```
1 int x = foo();
2 if (x > 0){
3     int y = x + 5;
4     int z = y / 4;
5 }
```

Is this legal?

```
1 int x = foo();
2 if (x > 0){
3     int y = x + 5;
4     int z = y >> 2;
5 }
```

Compiler Reasoning:

- At start of if-statement, x is in the range [1, INT_MAX]
- **That means y is in the range [6, INT_MAX]** →
- So y is positive on line 4, so we can do the transformation

Is this a valid assumption?

- If x + 5 does not overflow, then it is valid.
- If x + 5 **does** overflow, then UB and the program is undefined!

Summary

Reminder: Project 2 Due on 12/4

This is the **Wednesday** after we get back!








(changed from the original date of Monday after Thanksgiving)

Most of the difficulty of the project is in `prime()`, `hamming()`, and `pi()`, so try to have the other functions (particularly `map`, `filter`, and `chain`) working before leaving for break!

Reminder: Bonus Lecture Poll

Currently on Piazza.

Will close sometime after 6:30pm today.

4 (44% of users)		Template Metaprogramming and Multithreading in C++	Show Voters
4 (44% of users)		The games/graphics ecosystem: tools and tricks	Show Voters
3 (33% of users)		A Survey of C++ Use in Distributed Computing, Data Science, and Computational Biology	Show Voters
1 (11% of users)		File Handling and File Systems in C++	Show Voters
1 (11% of users)		Compiler Optimizations (Covers topics like unrolling, copy elision, (N)RVO, and others)	Show Voters
3 (33% of users)		The Insanity of Undefined Behavior	Show Voters
5 (56% of users)		The Standard Library: Useful Classes and Implementations	Show Voters

Notecards

Name and EID

One thing you learned today (can be "nothing")

One question you have about the material. **If you leave this blank, you will be docked points.**

If you do not want your question to be put on Piazza, please write the letters **NPZ** and circle them.



Additional Reading

- <https://www.includehelp.com/embedded-system/shift-and-rotate-instructions-in-8086-microprocessor.aspx>
- <https://stackoverflow.com/questions/6793262/why-dereferencing-a-null-pointer-is-undefined-behaviour>
- <https://kristerw.blogspot.com/2016/02/how-undefined-signed-overflow-enables.html><https://stackoverflow.com/questions/15718262/what-exactly-is-the-as-if-rule>
- <https://nullprogram.com/blog/2018/07/20>