C++ now

This is C++

Jon Kalb

2024

This is


WHAT'S
YOUR
SUPERPOWER?


Uncompromising
performance!

This is


safety
performance

This is

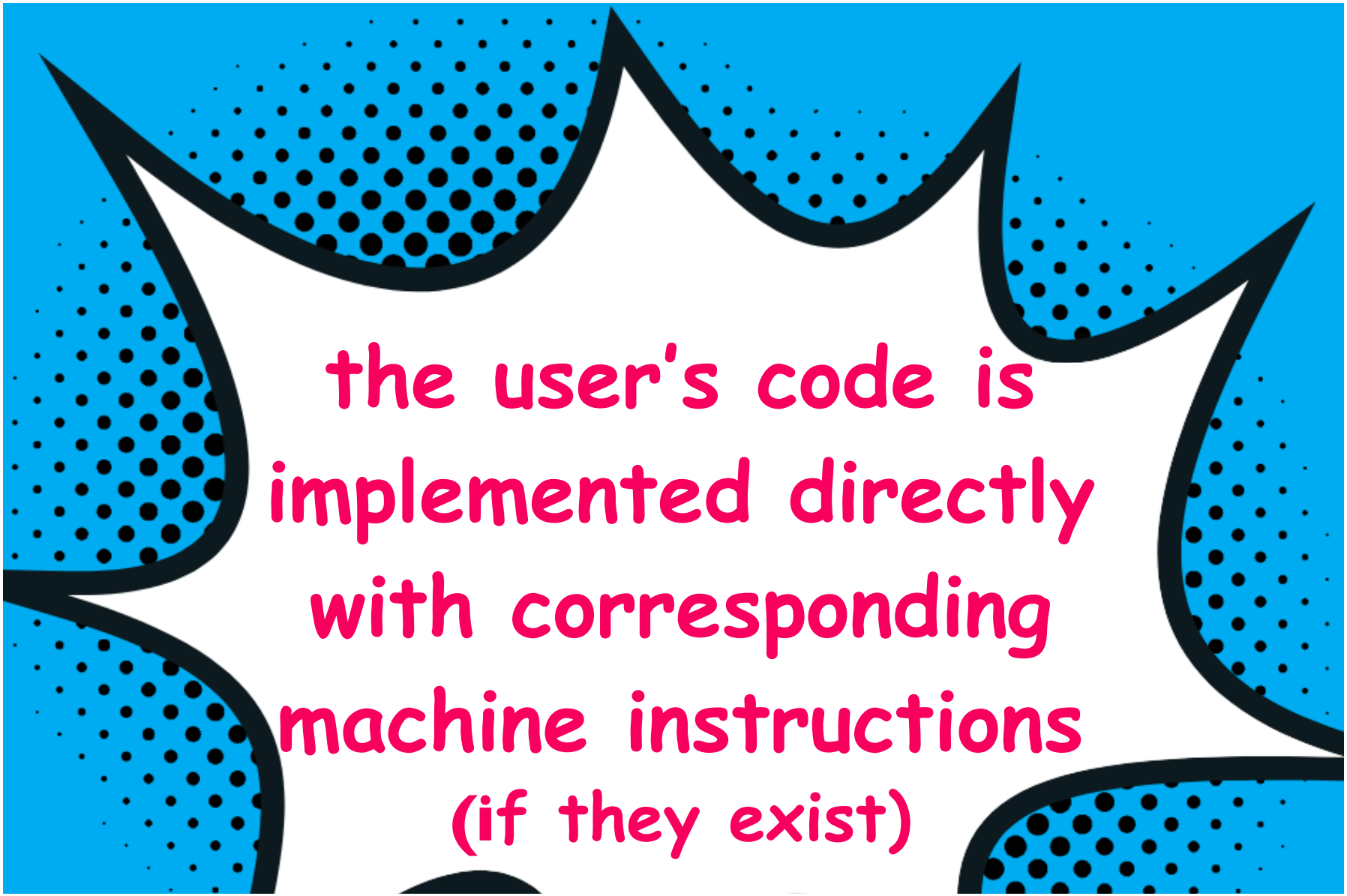
"*leave no room for a lower level language*"

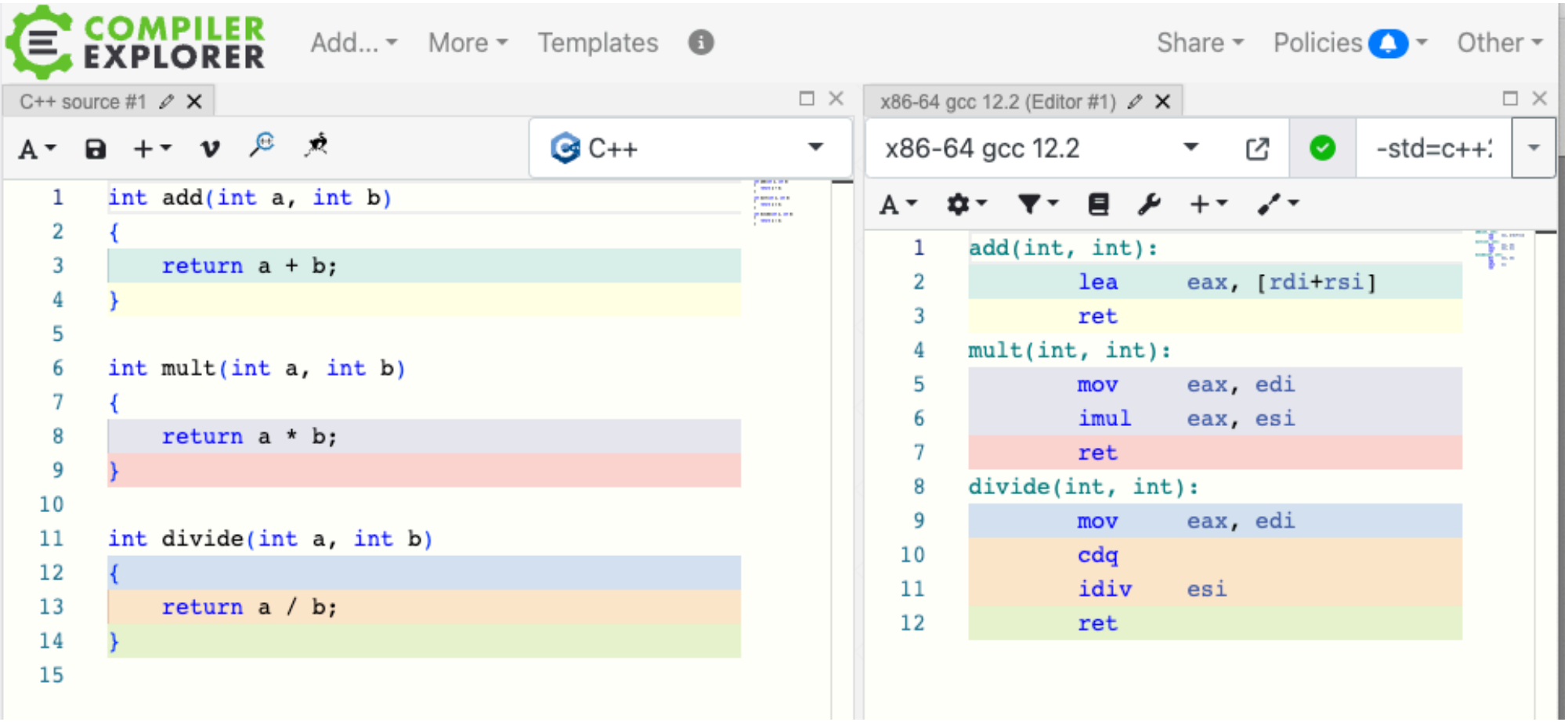
— P2137R0 (Originally Bjarne Stroustrup)

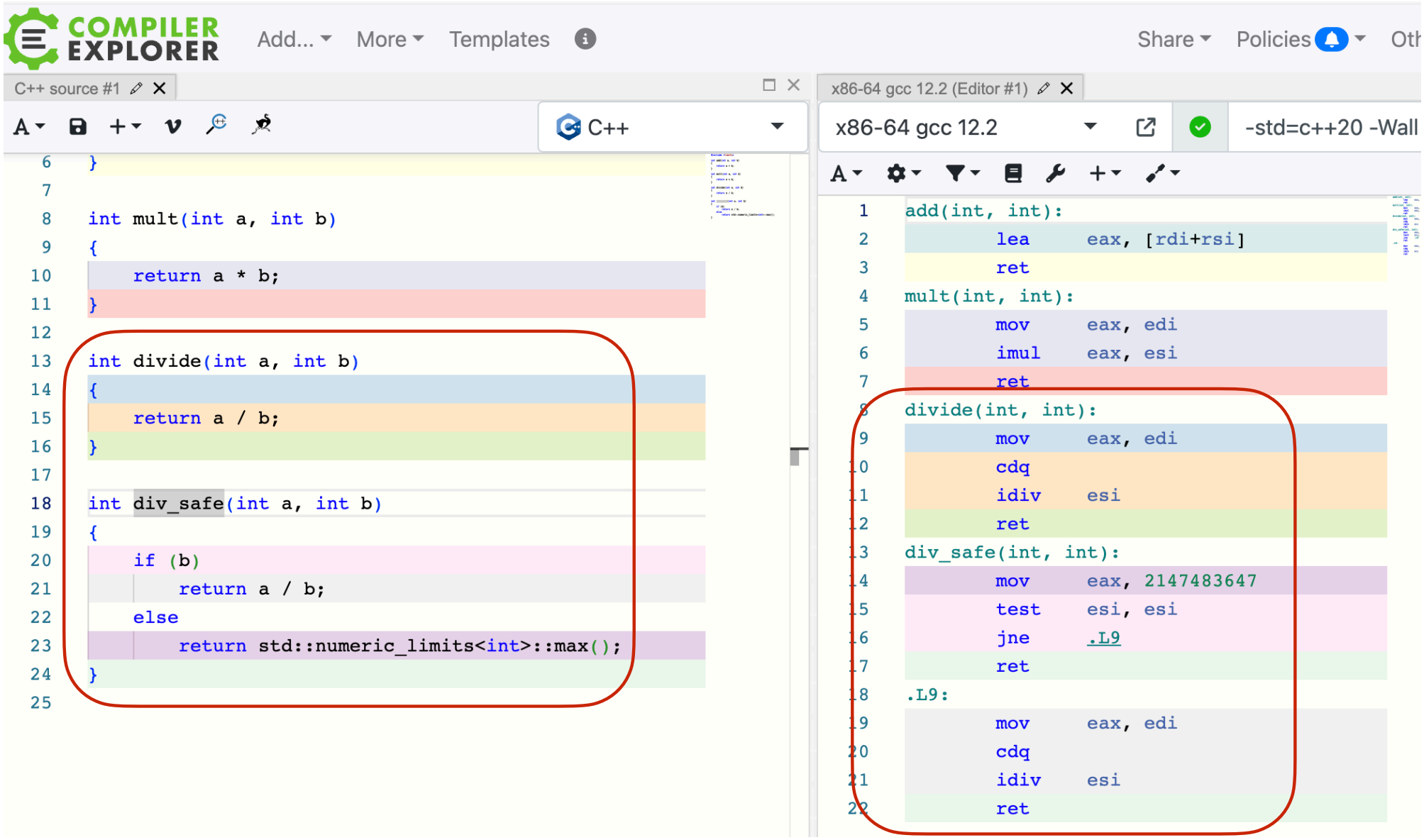
Zero Overhead Rule

- if a feature is not used: no cost
- if a feature is used: no greater cost


How do we
do it?


the user's code is
implemented directly
with corresponding
machine instructions
(if they exist)


COMPILER EXPLORER
Add...
More
Templates
Share
Policies
Other
C++ source #1
C++
x86-64 gcc 12.2 (Editor #1)
x86-64 gcc 12.2
-std=c++
int add(int a, int b)
{
return a + b;
}
int mult(int a, int b)
{
return a * b;
}
int divide(int a, int b)
{
return a / b;
}
add(int, int):
lea eax, [rdi+rsi]
ret
mult(int, int):
mov eax, edi
imul eax, esi
ret
divide(int, int):
mov eax, edi
cdq
idiv esi
ret


COMPILER EXPLORER
Add...
More
Templates
Share
Policies
C++ source #1
C++
x86-64 gcc 12.2 (Editor #1)
x86-64 gcc 12.2
-std=c++20 -Wall
}
int mult(int a, int b)
{
return a * b;
}
int divide(int a, int b)
{
return a / b;
}
int div_safe(int a, int b)
{
if (b)
return a / b;
else
return std::numeric_limits<int>::max();
}
add(int, int):
lea eax, [rdi+rsi]
ret
mult(int, int):
mov eax, edi
imul eax, esi
ret
divide(int, int):
mov eax, edi
cdq
idiv esi
ret
div_safe(int, int):
mov eax, 2147483647
test esi, esi
jne .L9
ret
.L9:
mov eax, edi
cdq
idiv esi
ret

divide_const

performance cost

- twice as many instructions
- includes a branch

note:

- simplest action on *divide by zero* (return a constant)

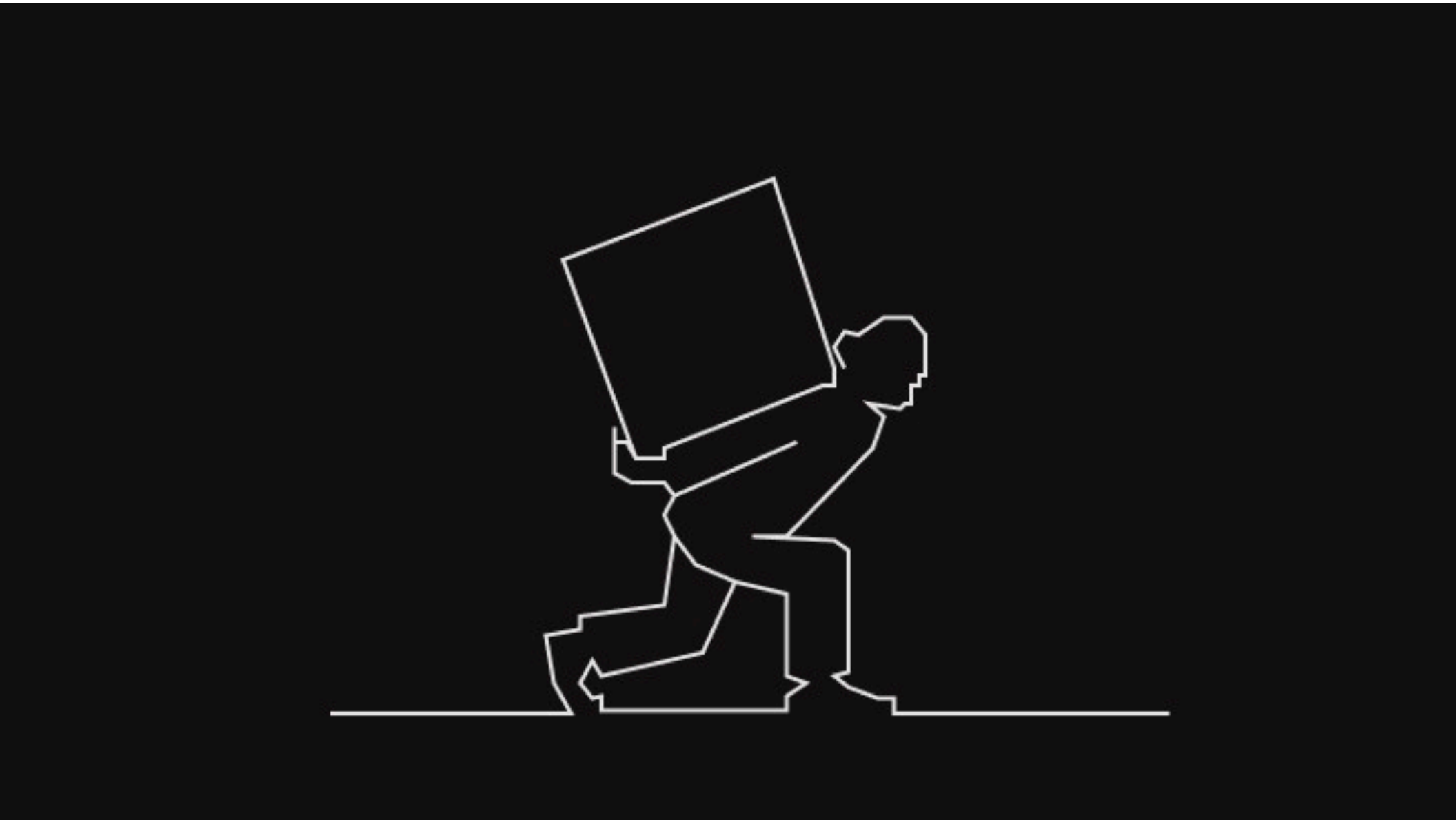
This is

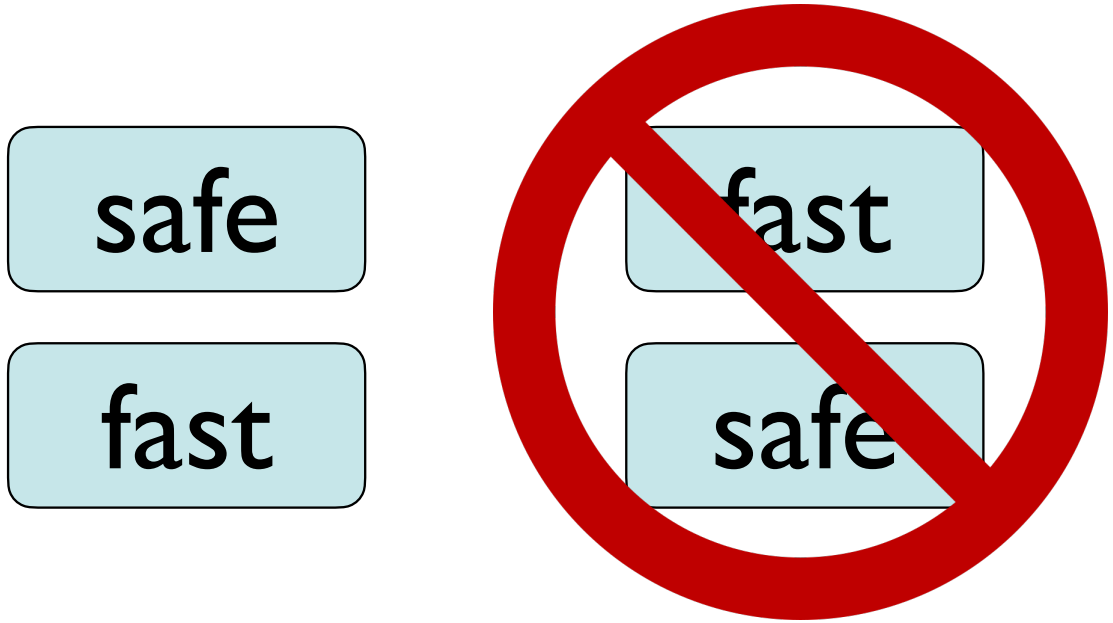
std::vector

both

- operator[]() <== efficient (not checked)
- at() <== safe (range checked)

This is




safe
fast
fast
safe

This is


Undefined
Behavior!

This is

Erroneous behavior for uninitialized reads

P2795R5 (C++26) - Thomas Köppe

```
int a; // not zero initialized
```

This is

```
int a{void}; // hypothetical syntax
```

```
int a{void}; // hypothetical syntax

int a = a;   // legal today
```

```
int a; // hypothetical future syntax
       // for zero initialization
```

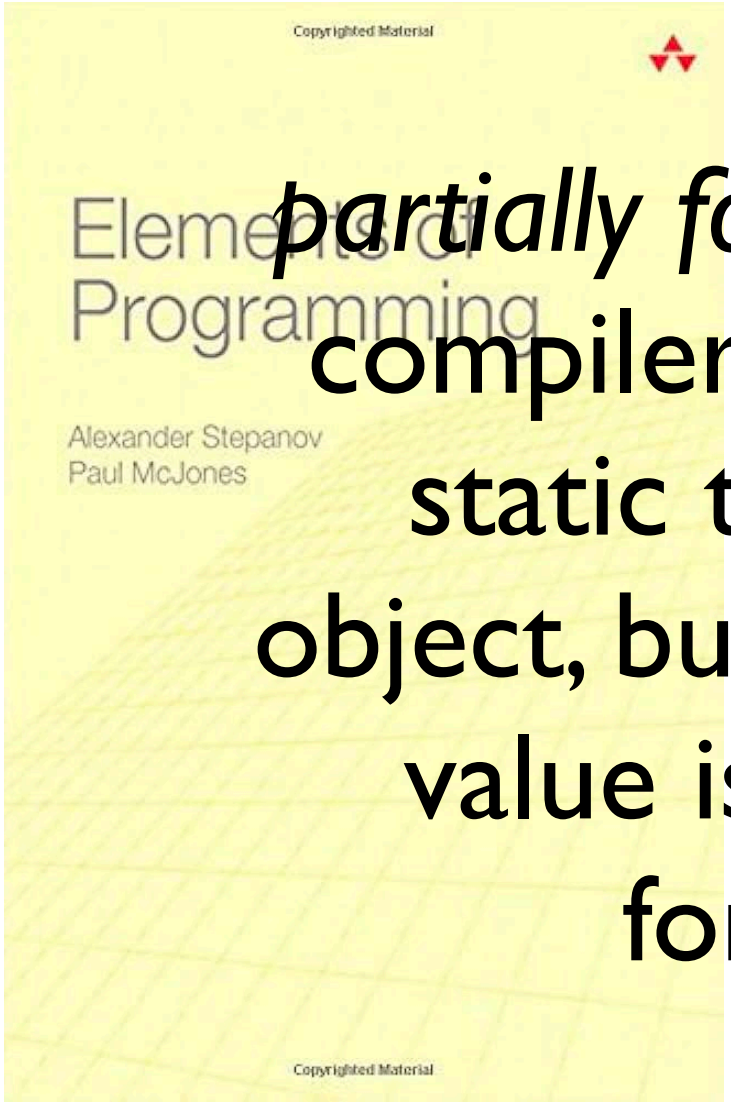
This is

```
int a; // not initialized
```

```
int a;     // not initialized
int b{a}; // ??
```

```
int a;     // not initialized
int b{a}; // undefined behavior
```

```
auto double_in_place(int& v)
{
    v *= 2;
}
```



partially formed object: compiler knows the static type of an object, but its runtime value is not fully formed

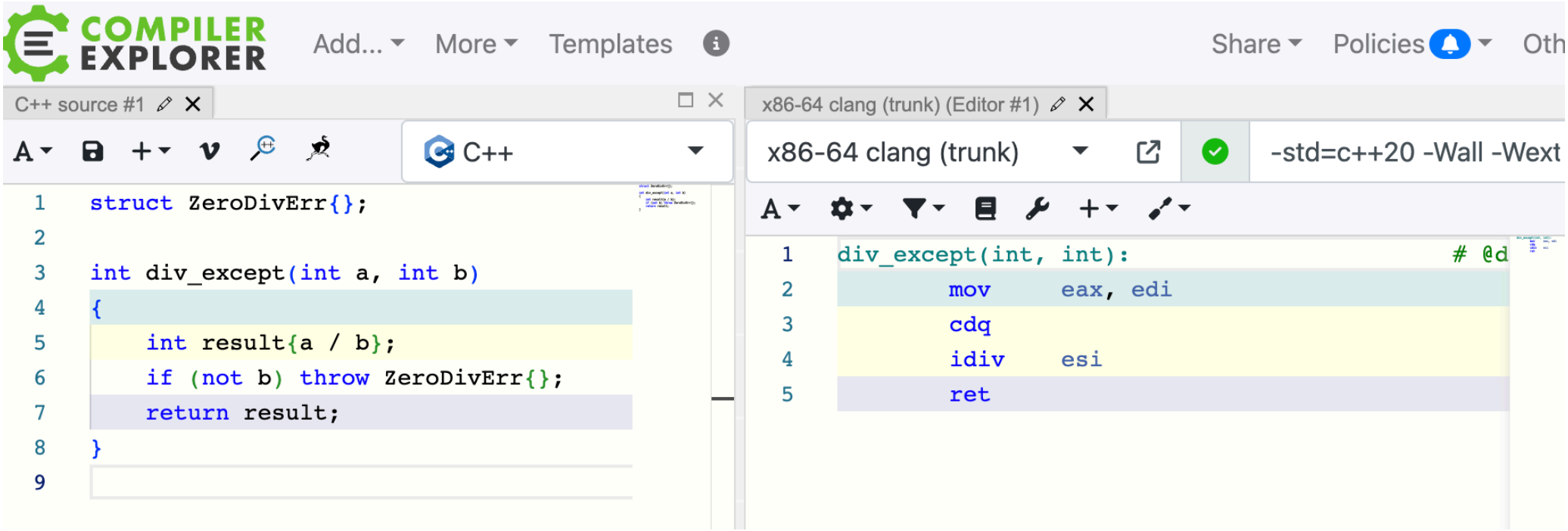
This is

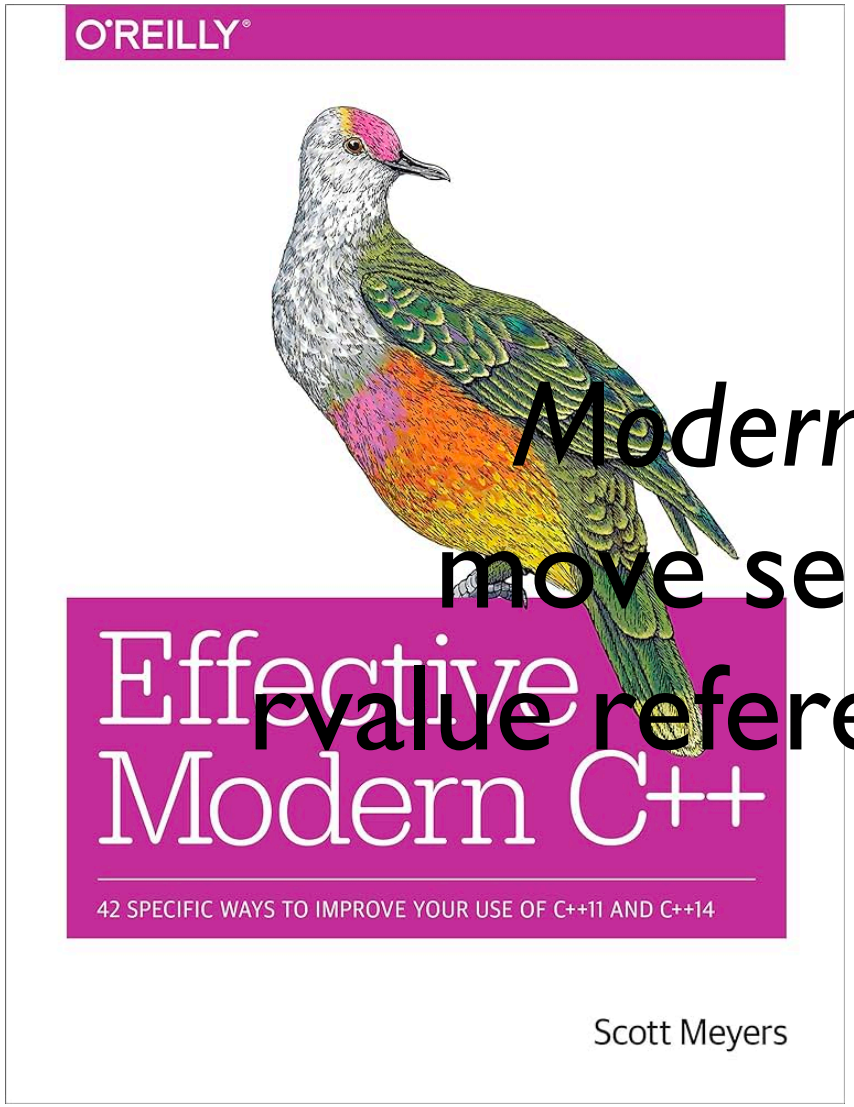
This is


Undefined
Behavior!

```
int div_except(int a, int b)
{
    int result{a / b};
    if (not b) throw ZeroDiv{};
    return result;
}
```

Will this ever be thrown?


COMPILER EXPLORER
Add...
More
Templates
Share
Policies
Oth
C++ source #1
C++
x86-64 clang (trunk) (Editor #1)
x86-64 clang (trunk)
-std=c++20 -Wall -Wext
1 struct ZeroDivErr{};
2
3 int div_except(int a, int b)
4 {
5 int result{a / b};
6 if (not b) throw ZeroDivErr{};
7 return result;
8 }
9
1 div_except(int, int): # @d
2 mov eax, edi
3 cdq
4 idiv esi
5 ret



Modern C++:

move semantics

rvalue reference syntax

This is

```
Widget::Widget(
      std::list<Gadget> && inGL
                        ):
      mGadgets{std::move(inGL)}
{}
```

cppreference.com

Page | Discussion

C++ / Containers library / std::vector

std::vector<T,Allocator>::vector

```
constexpr vector( vector&& other ) noexcept;
```
(since C++20)

vector move is *noexcept*

cppreference.com

Page | Discussion

C++ / Containers library / std::list

std::list<T,Allocator>::list

```
list( list&& other );
```
(8) (since C++11)

list move is **not** *noexcept*

move enabling

- primitive type ==> shallow copy
- (owning) pointer ==> shallow copy, zero the source
- compound type ==> (create and) use move operations that recursively follow these rules

move enabling (cont)

- primitive type ==> shallow copy
- (owning) pointer ==> shallow copy, zero the source
- compound type ==> (create and) use move operations that recursively follow these rules
- destructor: must be safe on moved from objects
- assignment operator: must be safe to assign to moved from objects

```
allocation: <some address>
size:       10
capacity:   16

allocation: <nil>
size:       10
capacity:   16
```

“…a sentinel node is a specifically designated node used with linked lists and trees as a traversal path terminator. This type of node does not hold or reference any data managed by the data structure.”

— Wikipedia

What do we call invariants that don't always hold?

They are called ***not invariants****.*

Sutter's Mill

Herb Sutter on software development

Move, simply

Herb Sutter 2020-02-17 9 Minutes

Table 33: *Cpp17MoveAssignable* requirements [tab:cpp17.moveassignable]

Expression	Return type	Return value	Post-condition
`t = rv`	`T&`	`t`	If `t` and `rv` do not refer to the same object, `t` is equivalent to the value of `rv` before the assignment

`rv`'s state is unspecified.

[*Note 2*: `rv` must still meet the requirements of the library component that is using it, whether or not `t` and `rv` refer to the same object. The operations listed in those requirements must work as specified whether `rv` has been moved from or not. — *end note*]

```
unique_ptr<int> a[]
      {
          make_unique<int>(1),
          make_unique<int>(0)
      };

sort(begin(a),
       end(a),
       [](const auto& a, const auto& b)
       { return *a < *b; });
```

UB if moved from

Table 33: *Cpp17MoveAssignable* requirements [tab:cpp17.moveassignable]

Expression	Return type	Return value	Post-condition
`t = rv`	`T&`	`t`	If `t` and `rv` do not refer to the same object, `t` is equivalent to the value of `rv` before the assignment

rv is assignable-to and destructible. [Note: rv's state is otherwise unspecified.]

What is the meaning of these function signatures?

```
auto foo(widget& w);

auto foo(widget&& w);
```

The blog's suggestion: these two mean the same.

```
ViewPortID CreateViewPort(/**/);
Void FreeViewPt(ViewPortID vpID);
int GetBitDepth(ViewPortID vpID);
int GetXDim(ViewPortID vpID);
…
```

```
struct ViewPort
{
    ViewPort(/**/):
        vpID_(CreateViewPt(/**/))
       {/*RAII*/}

    ~ViewPort(){if (vpID_)
                   FreeViewPt(vpID_);}
    ~~~~

private:
     ViewPortID vpID_;
};
```

How do we implement move operations?

```
struct ViewPort
{
    ~~~~
    ViewPort(ViewPort&& vp):
        vpID_(vp.vpID_)
       {vp.vpID_ = 0;}
    ~~~~
};
```

```
struct ViewPort
{
    ~~~~
    ViewPort(ViewPort&& vp):
        vpID_(vp.vpID_)
       {vp.vpID_ = 0;
       /* Call OS to make a
          new ViewPort */}
    ~~~~
};
```

```
struct ViewPort
{
    ~~~~
    ViewPort(ViewPort&& vp):
        vpID_(vp.vpID_)
       {vp.vpID_ = 0;}
   /* modify all member functions
      to “fake” something when moved
      From */
   ~~~~
};
```

Summary

- What is C++?
  - Uncompromised performance.
- Achieved by: living with UB,
  accepting a safety trade-off

Summary (continued)

- Committee: Standard types: moved from == fully formed
  - This ship has sailed
  - IMHO: a mistake because:
    - Minor/major performance hits
    - Leads to logic errors

Summary (continued)

- Committee direction: user code moved from == fully formed
- Minor reduction in cognitive load, but encourages logic errors.
- Some use cases, unacceptable performance, defeating move semantics.

Summary (continued)

- Correct:

  moved from == partially formed

This is

Summary (continued)

- Application safety != language safety

This is