

+ 24

# This is C++

JON KALB



**Cppcon**  
The C++ Conference

20  
24



This is





**WHAT'S  
YOUR  
SUPERPOWER?**



Uncompromising  
performance!

This is





This is



*“leave no room for a lower level language”*

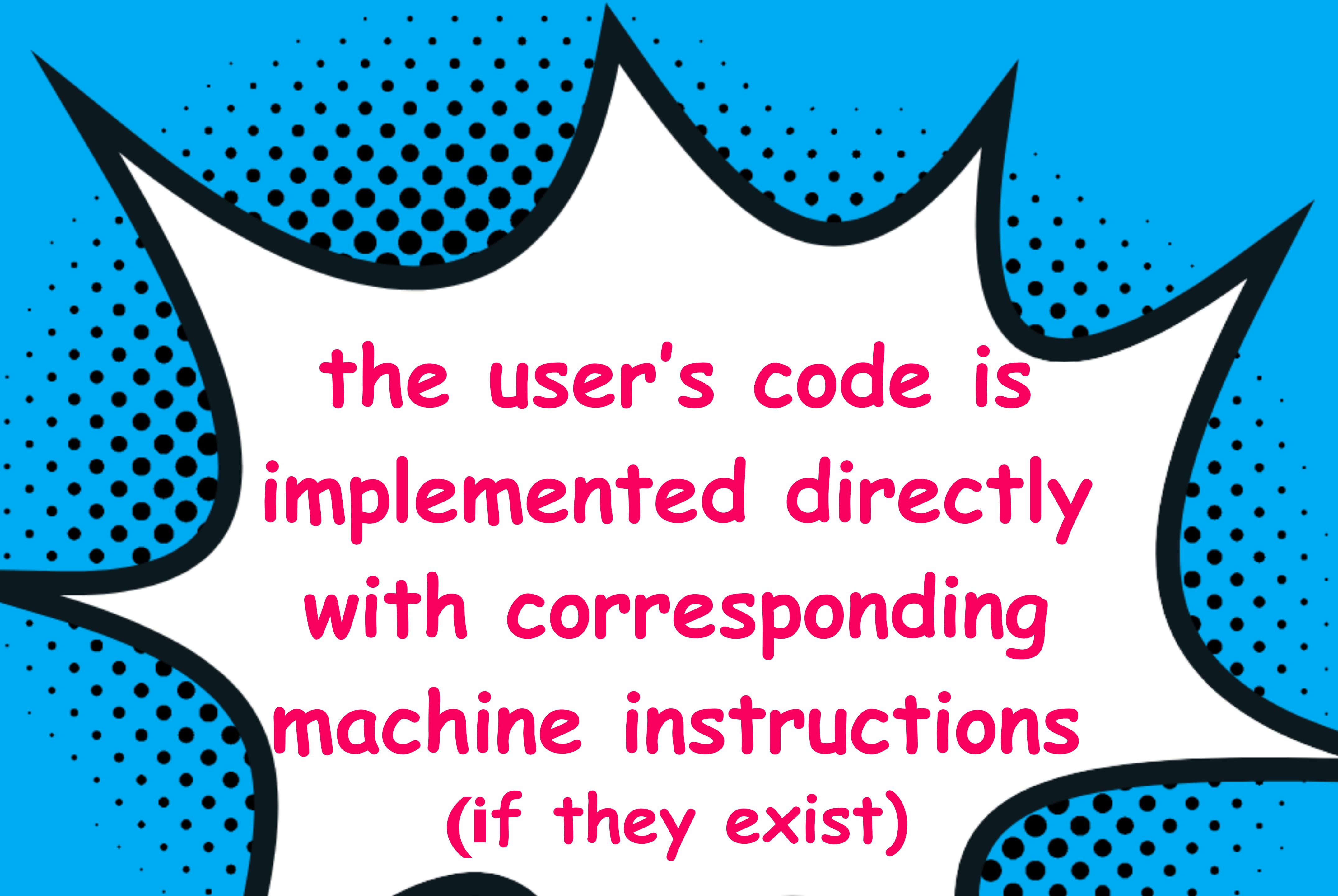
– P2137R0 (Originally Bjarne Stroustrup)

# | 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)



Add... More Templates ⓘ

Share Policies ⓘ Other

C++ source #1 ✎ ×

A - B + v 🔍 ⚡ C++ ▾

```
1 int add(int a, int b)
2 {
3     return a + b;
4 }
5
6 int mult(int a, int b)
7 {
8     return a * b;
9 }
10
11 int divide(int a, int b)
12 {
13     return a / b;
14 }
15
```

x86-64 gcc 12.2 (Editor #1) ✎ ×

x86-64 gcc 12.2 ▾ 🔍 ⓘ -std=c++11

A - B + v 🔍 ⚡ C++ ▾

```
1 add(int, int):
2     lea    eax, [rdi+rsi]
3     ret
4 mult(int, int):
5     mov    eax, edi
6     imul   eax, esi
7     ret
8 divide(int, int):
9     mov    eax, edi
10    cdq
11    idiv   esi
12    ret
```



Add... More Templates ⓘ

Share Policies ⓘ Other

C++ source #1 ⚪ X

A ⚪ + ⚪ v ⚪

C++

```
6 }
7
8 int mult(int a, int b)
9 {
10    return a * b;
11}
12
13 int divide(int a, int b)
14 {
15    return a / b;
16}
17
18 int div_safe(int a, int b)
19 {
20    if (b)
21        return a / b;
22    else
23        return std::numeric_limits<int>::max();
24}
25
```

x86-64 gcc 12.2 (Editor #1) ⚪ X

x86-64 gcc 12.2 ⚪ ⓘ -std=c++20 -Wall

A ⚪ ⚪ + ⚪

```
1 add(int, int):
2     lea    eax, [rdi+rsi]
3     ret
4 mult(int, int):
5     mov    eax, edi
6     imul   eax, esi
7     ret
```

```
8 divide(int, int):
9     mov    eax, edi
10    cdq
11    idiv   esi
12    ret
13 div_safe(int, int):
14    mov    eax, 2147483647
15    test   esi, esi
16    jne   .L9
17    ret
18 .L9:
19    mov    eax, edi
20    cdq
21    idiv   esi
22    ret
```

## | divide\_const

performance cost

- ❖ twice as many instructions
- ❖ includes a branch

note:

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

This is



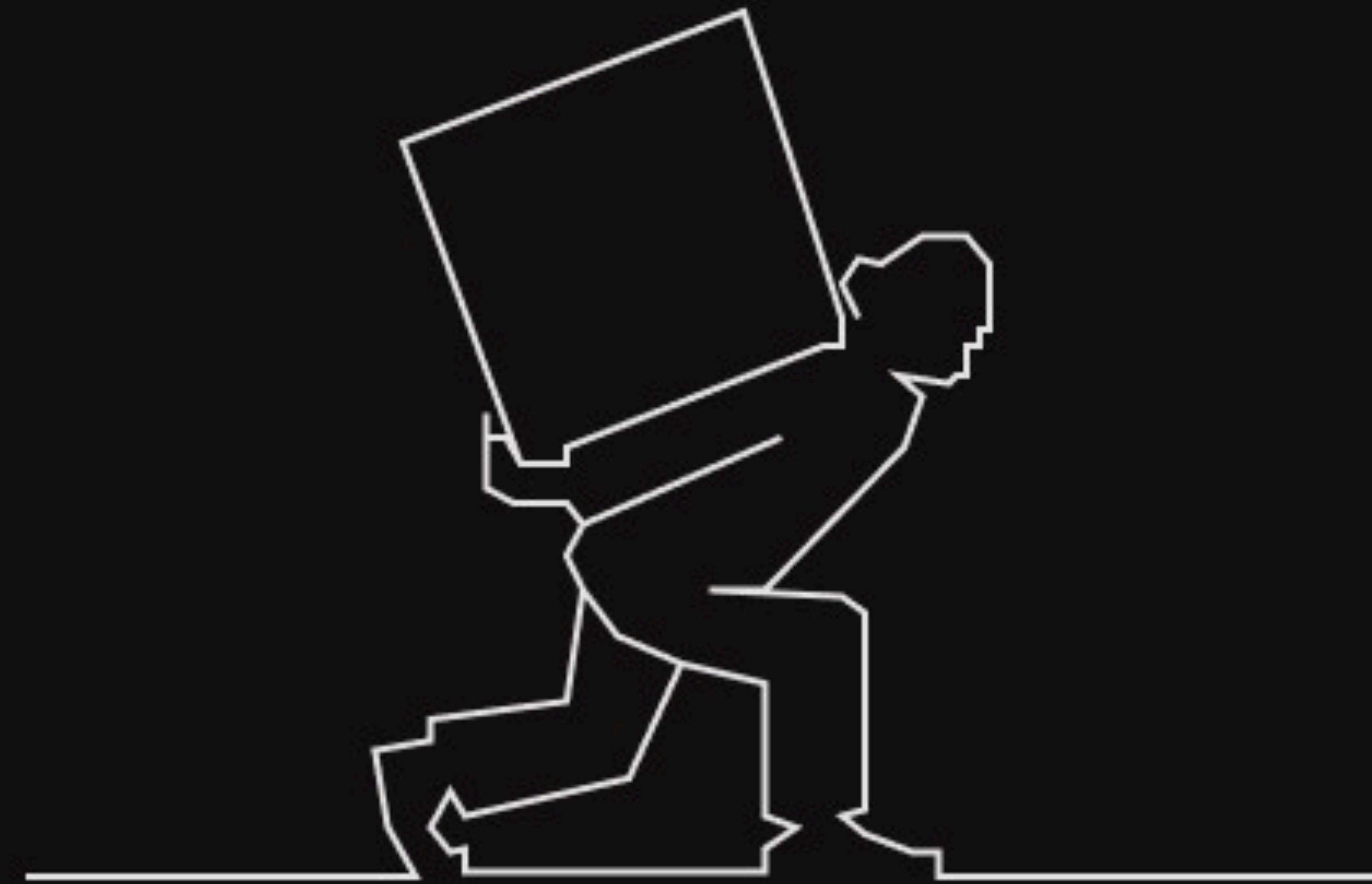
# | std::vector

both

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

This is





safe

fast



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

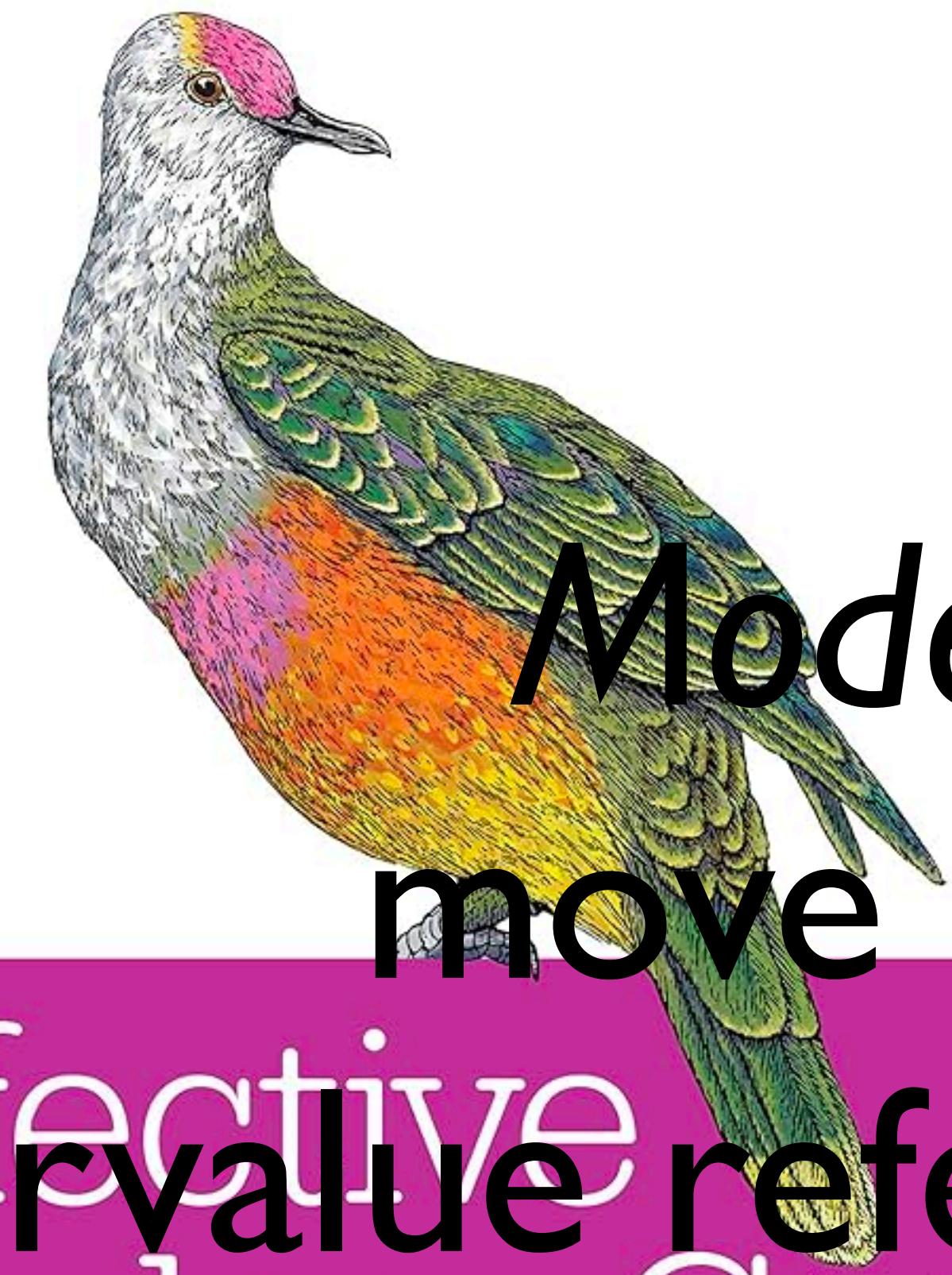




Undefined  
Behavior!

*Is C++ safe?*

O'REILLY®



# Modern C++: move semantics rvalue reference syntax

Effective  
Modern C++

42 SPECIFIC WAYS TO IMPROVE YOUR USE OF C++11 AND C++14

Scott Meyers

This is



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

Page Discussion

C++ Containers library std::vector

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

constexpr vector( vector&& other ) noexcept;

vector move is  
*noexcept*

(since C++20)

Page Discussion

C++ Containers library std::list

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

list( list&& other );

list move is  
*not noexcept*

(8) (since C++11)

# 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
- ◆ 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
<code>t = rv</code>	<code>T&amp;</code>	<code>t</code>	If <code>t</code> and <code>rv</code> do not refer to the same object, <code>t</code> is equivalent to the value of <code>rv</code> before the assignment
<code>rv</code> 's state is unspecified.			<p>[<i>Note 2</i>: <code>rv</code> must still meet the requirements of the library component that is using it, whether or not <code>t</code> and <code>rv</code> refer to the same object. The operations listed in those requirements must work as specified whether <code>rv</code> has been moved from or not.]</p> <p>— <i>end note</i></p>

```
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
<code>t = rv</code>	<code>T&amp;</code>	<code>t</code>	If <code>t</code> and <code>rv</code> do not refer to the same object, <code>t</code> is equivalent to the value of <code>rv</code> before the assignment

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

+ 21

# Warning: std::find() is broken!

SEAN PARENT



# Relaxing Requirements of Moved-From Objects

P2345 - Sean Parent

# *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  $\neq$  language safety

This is

