Can You RVO?

Using Return Value Optimization for Performance in Bloomberg C++ Codebases

MICHELLE FAE D'SOUZA

Cppcon
The C++ Conference

20
24

September 15 - 20

Can You RVO?

How many people here have heard about “Return Value Optimization”?

How many people here are **experts** on “Return Value Optimization”?

TechAtBloomberg.com

Michelle Fae D’Souza

Software Engineer @ Bloomberg

Builds code for the solution that helps traders buy & sell securities on the exchange

Computer Science @ UC Berkeley

Member of C++ Guild @ Bloomberg

Speak 5+ languages

TechAtBloomberg.com

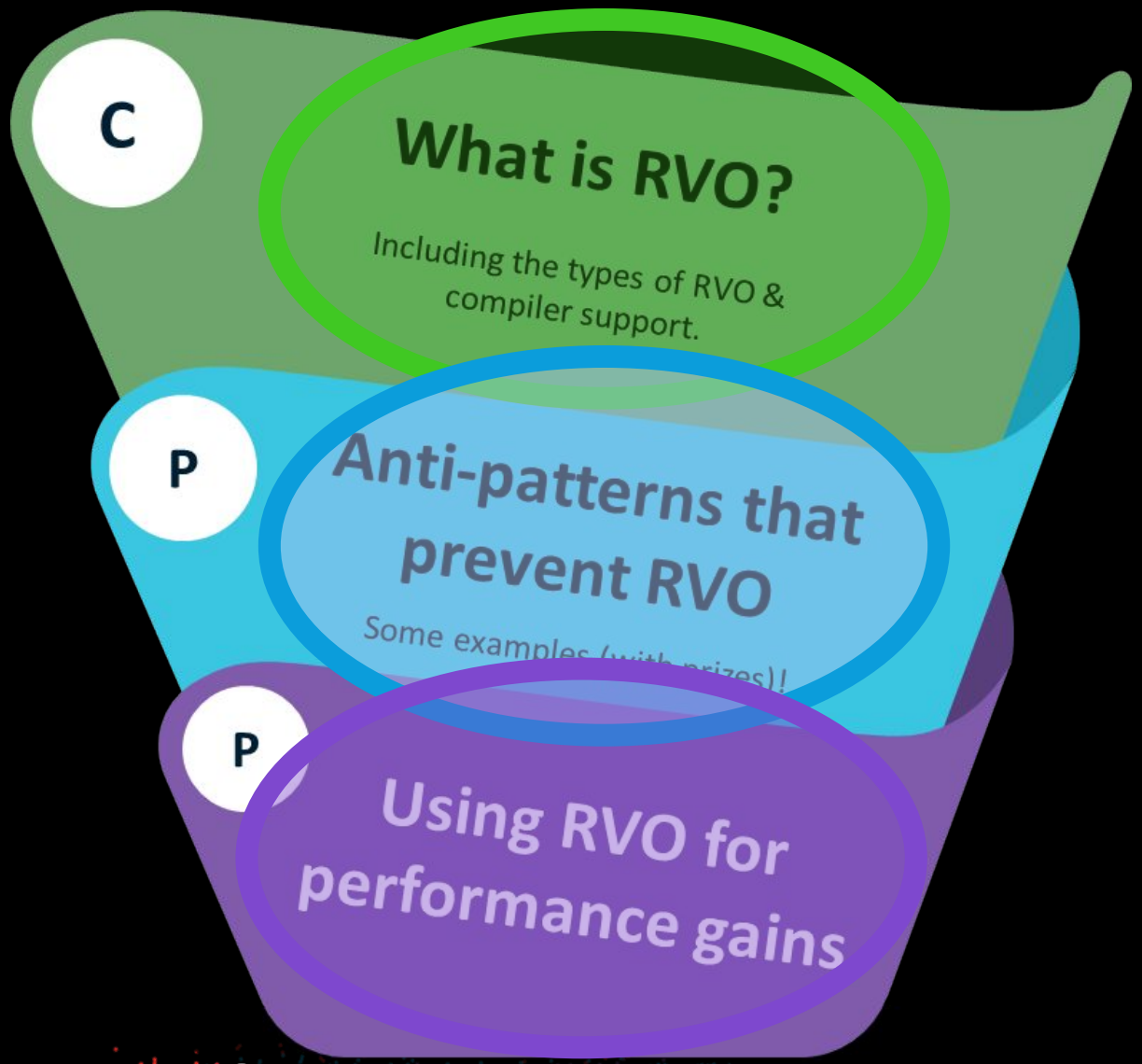
Bloomberg

Engineering

NO ONE CARES

Return Value Optimization (RVO)

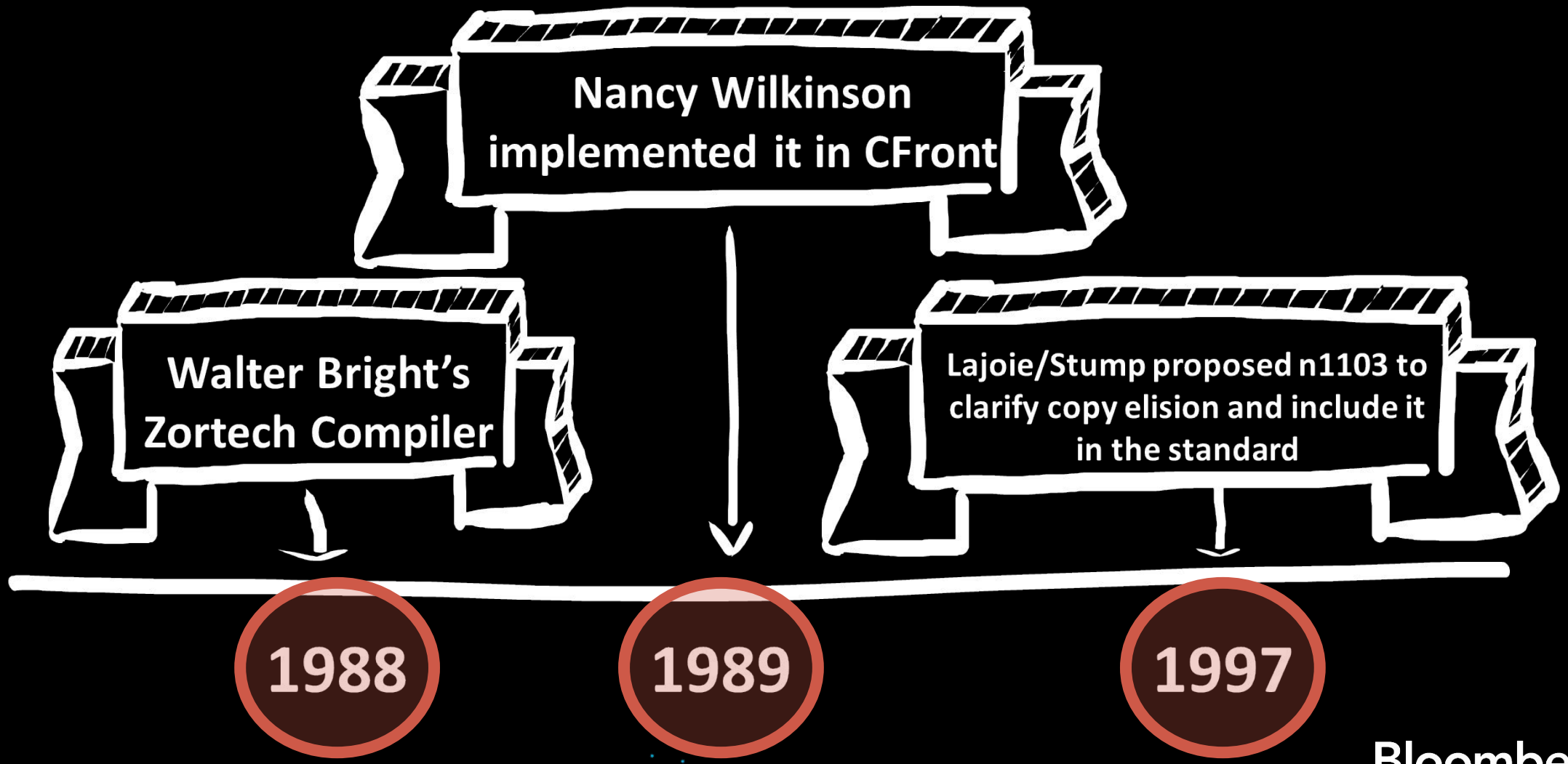
Agenda



Source: PresentationGO (www.presentationgo.com)

Questions at the end

The History of RVO



Source: PresentationGO (www.presentationgo.com)

Bloomberg

Engineering

RVO at Bloomberg

aim-ote/[redacted] 8015

[ENG1AIOTPE-259] Prefetch afmt bits

Probably better make request a value instead of a reference. **RVO** will make it a free copy anyway.

[redacted] opened on Feb 1 16 comments aim-ote:main aj6:afmt_prefetch

encodingutil in the response side

... corresponding run-lengths of each column value. Return 0 on success and a non-zero value otherwise. However, we agreed to split the output so that you can use **RVO** to assign the salient value (the column) to ...

[redacted] opened 14 days ago 31 comments BQL:main ← [redacted]:array

PORT/[redacted] #3638

Invert dependency to Context in PortfolioDummy

Both are valid options. If you use const& your temporary object will have its life extended while if you use const you copy (I think we could even get **RVO** or NRVO) so in both cases the net effect is the same.

[redacted] opened 11 days ago 48 comments PORT:master ←

atw/[redacted] #71

Clean Repeating Code for Getting Latest External Status

> strings are expensive to pass back by value. Shouldn't we try to pass by reference and make the string const? I was hoping this would just leverage Return Value Optimization (**RVO**)

[redacted] opened 26 days ago 21 comments atw:main ← [redacted]cancel_not_sent_vcon_check

bqnt/[redacted] #73

ENG2BQNTMM-1880 : Check bars present

I have a different suggestion. Allocate the vector in the function. That way there is no code duplication. **RVO** should take care of avoiding vector copy.

[redacted] opened on Jan 31 110 comments main [redacted]unt

...rning a string this way prevents return-value-optimization (**RVO**) i...lable to return the idea of a possible violation.

RVO in the C++ World

microsoft/cpp_client_telemetry

Fix for build errors when building with gcc 13

Seems **return value optimization** does apply here so explicit move is not required.

geodynamics/aspect

support different compositions with particles

This should not allocate (**return value optimization**, as we **return** a copy from a locally constructed temporary in that function).

work in progress

[redacted] 21 · Opened 24 days ago · #5963

So what if I forget RVO?!?!

Added functions for generating a blinded id for version check auth

... then **returning** that **value**, C++ **return value optimization** kicks in and will avoid having a that **return** variable in this function at all: the compiler will write the returned value directly into wherever the calling function is going to store it.

[redacted] 25 · Opened on Jun 17 · #87

Bloomberg

Engineering

What is Return Value Optimization?

What is Return Value Optimization?

11.9.6 Copy/move elision [class.copy.elision]

1 When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the constructor selected for the copy/move operation and/or the destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object. If the first parameter of the selected constructor is an rvalue reference to the object's type, the destruction of that object occurs when the target would have been destroyed; otherwise, the destruction occurs at the later of the times when the two objects would have been destroyed without the optimization.[104] This elision of copy/move operations, called *copy elision*, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

.1) — in a `return` statement in a function with a class return type, when the *expression* is the name of a non-volatile object with automatic storage duration (other than a function parameter or a variable introduced by the *exception-declaration* of a *handler* ([except.handle])) with the same type (ignoring cv-qualification) as the function return type, the copy/move operation can be omitted by constructing the object directly into the function call's return object

.2) — in a *throw-expression* ([expr.throw]), when the operand is the name of a non-volatile object with automatic storage duration (other than a function or catch-clause parameter) that belongs to a scope that does not contain the innermost enclosing *compound-statement* associated with a *try-block* (if there is one), the copy/move operation can be omitted by constructing the object directly into the exception object

.3) — in a coroutine, a copy of a coroutine parameter can be omitted and references to that copy replaced with references to the corresponding parameter if the meaning of the program will be unchanged except for the execution of a constructor and destructor for the parameter copy object

.4) — when the *exception-declaration* of a *handler* ([except.handle]) declares an object of the same type (except for cv-qualification) as the exception object ([except.throw]), the copy operation can be omitted by treating the *exception-declaration* as an alias for the exception object if the meaning of the program will be unchanged except for the execution of constructors and destructors for the object declared by the *exception-declaration*.

[*Note 1*: There cannot be a move from the exception object because it is always an lvalue. — *end note*]

Copy elision is not permitted where an expression is evaluated in a context requiring a constant expression ([expr.const]) and in constant initialization ([basic.start.static]).

[*Note 2*: It is possible that copy elision is performed if the same expression is evaluated in another context. — *end note*]

Copy Elision

Return Value Optimization conditions

Source: cppreference.com
https://en.cppreference.com/w/cpp/language/copy_elision

Bloomberg
Engineering

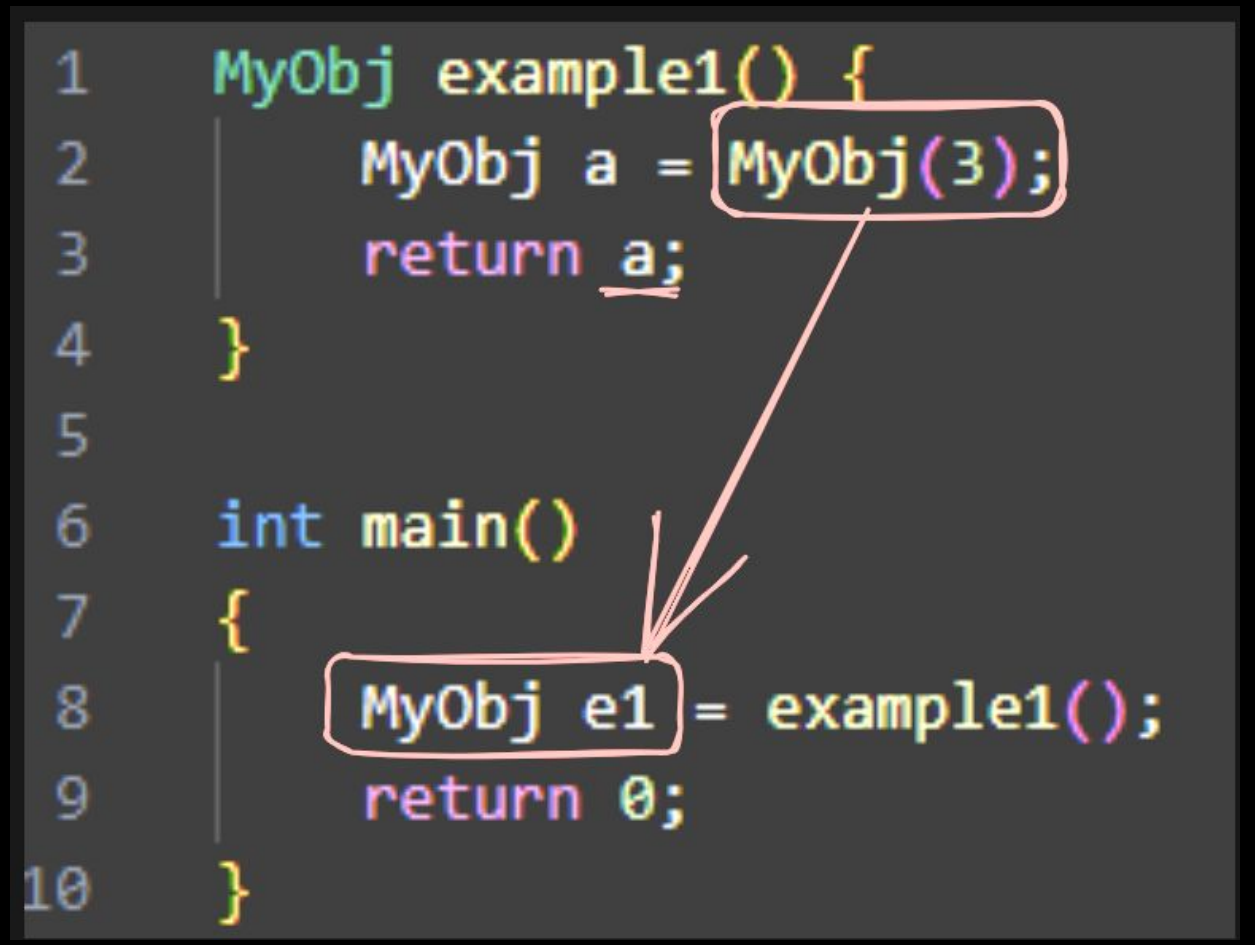
What is Return Value Optimization?

Michelle's Best Friend ...

Bloomberg

Engineering

What is Return Value Optimization?



Bloomberg

Engineering

No RVO vs. RVO: An Assembly Perspective

```
1  example1(): #RVO OFF
2          push    rbp
3          mov     rbp, rsp
4          sub     rsp, 32
5          mov     QWORD PTR [rbp-24], rdi
6          lea     rax, [rbp-8]
7          mov     esi, 3
8          mov     rdi, rax
9          call    MyObj::MyObj(int) [complete
   object constructor]
10         lea     rdx, [rbp-8]
11         mov     rax, QWORD PTR [rbp-24]
12         mov     rsi, rdx
13         mov     rdi, rax
14         call    MyObj::MyObj(MyObj&&) [comp
   lete object constructor]
15         mov     rax, QWORD PTR [rbp-24]
16         leave
17         ret
18
```

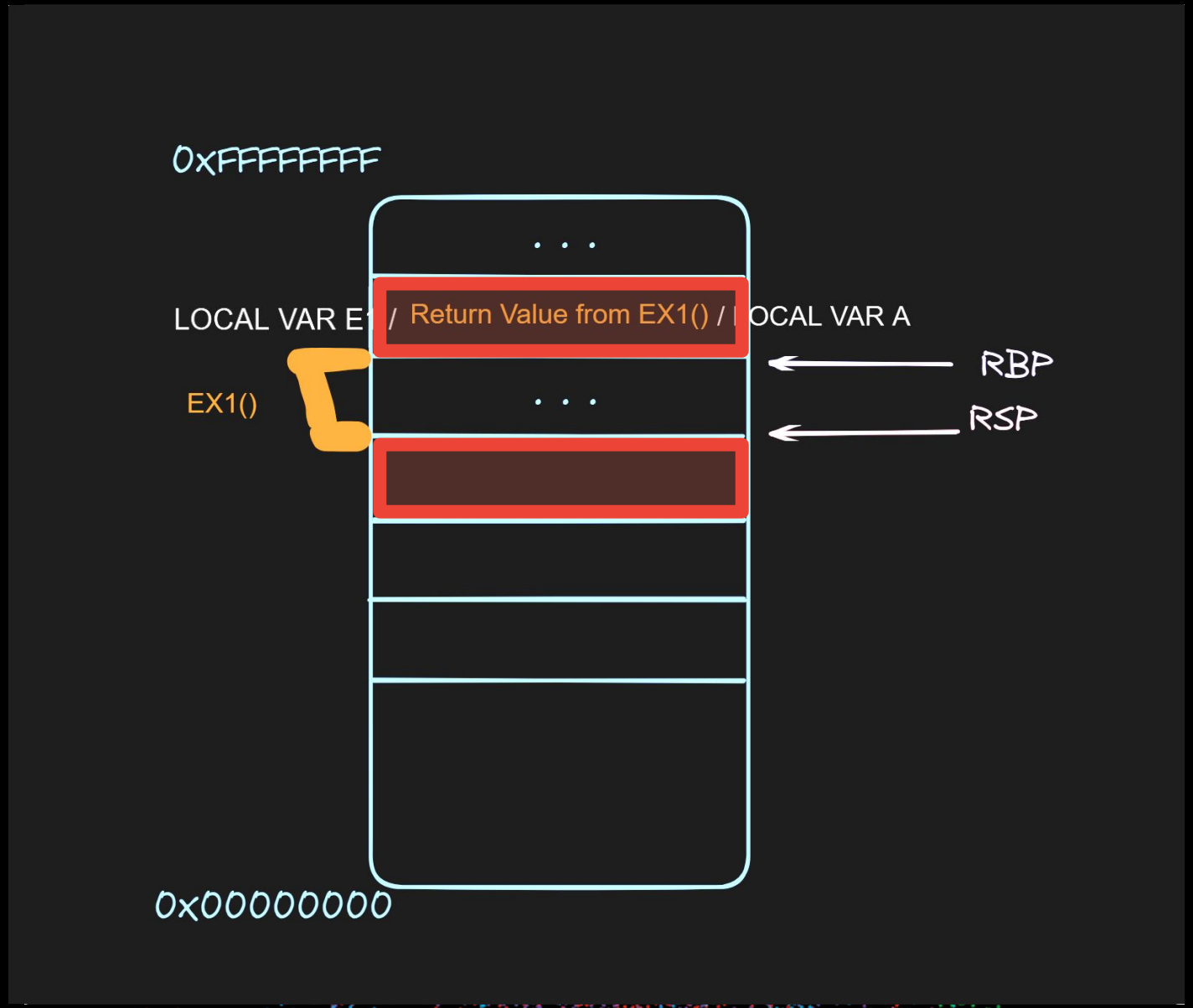
```
1  example1(): #RVO ON
2          push    rbp
3          mov     rbp, rsp
4          sub     rsp, 16
5          mov     QWORD PTR [rbp-8], rdi
6          mov     rax, QWORD PTR [rbp-8]
7          mov     esi, 3
8          mov     rdi, rax
9          call    MyObj::MyObj(int) [complete
   object constructor]
10         nop
11         mov     rax, QWORD PTR [rbp-8]
12         leave
13         ret
14
```

x86 - 64 bit

Bloomberg

Engineering

No RVO vs. RVO: A Stack Frame Perspective

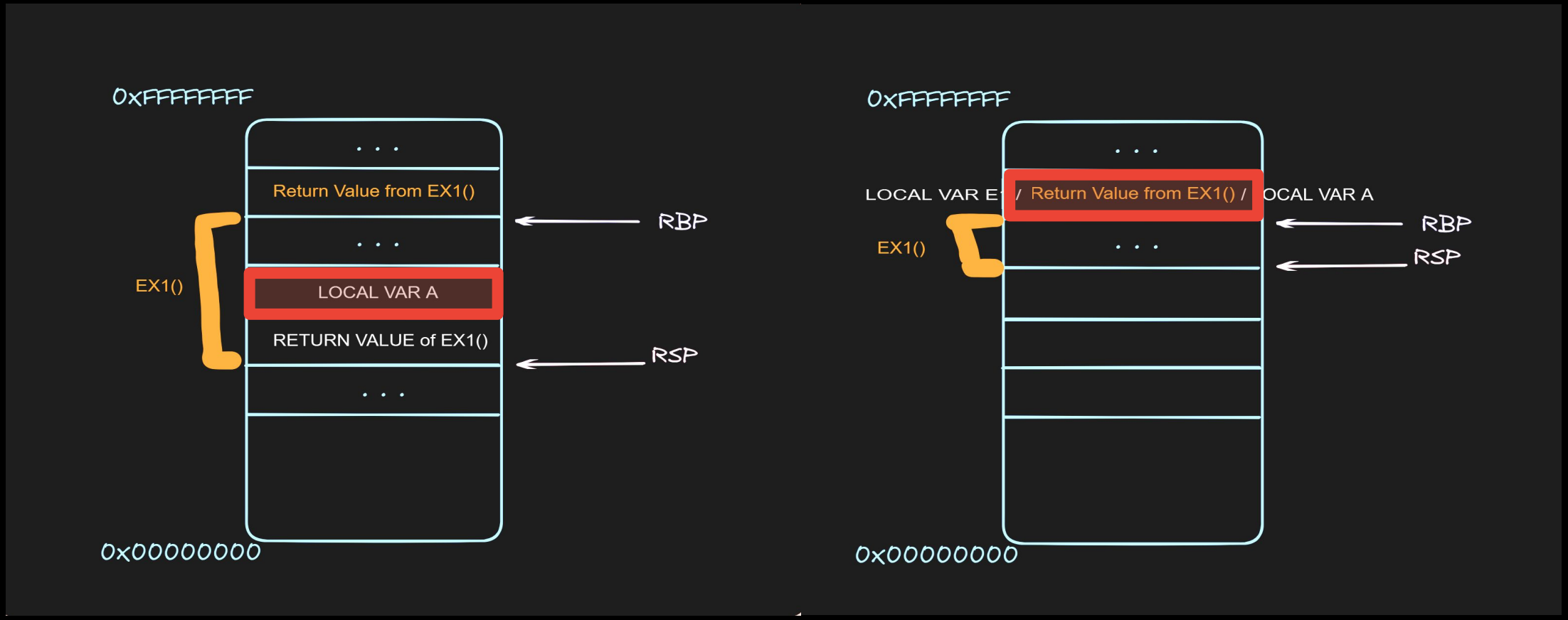


x86 - 64 bit

Bloomberg

Engineering

No RVO vs. RVO: A Stack Frame Perspective



x86 - 64 bit

Bloomberg

Engineering

```
1   MyObj example1() {
2       // NRVO
3       MyObj a = MyObj(3);
4       return a;
5   }
```

Why do I need to know the difference?

Unnamed Return Value Optimization

```
15      MyObj e2 = example2();
16      return 0;
17  }
```

Does my compiler have to support RVO?

- Compilers are allowed to perform URVO since C++98
- Compilers are **required** to provide URVO support since C++17
- NRVO support is optional, but recommended

TechAtBloomberg.com

So … we can turn off (N)RVO 😈…

(Doesn’t work for C++ 17 (and onwards) *U*RVO)

-fno-elide-constructors

… for gcc & clang

```
-fno-elide-constructors
    The C++ standard allows an implementation to omit creating a
    temporary which is only used to initialize another object of the
    same type.  Specifying this option disables that optimization, and
    forces G++ to call the copy constructor in all cases.
```

What about MSVC?

MSVC disables NRVO by default

… can be enabled using /Zc:nrvo, which is turned on automatically with the /O2 optimization, the /permissive- option, or /std:c++20 or later.

Playing around with an example!

Assume the copy assignment operator + destructor is also defined (due to the rule of three)

```
#include <iostream>
using namespace std;
class MyObj {
    public:
        int a;
        MyObj(int x) {
            cout << "calling og constructor \n";
            a=x;
        }
        MyObj(const MyObj &obj) { // copy constructor
            cout << "calling copy constructor \n";
            a = obj.a;
        }
};
```

x86-64 gcc 13.2 -std=c++17 -O0 -w -fno-elide-constructors

“OG” ???

Original

```
cout << "calling og constructor \n";
```

Playing around with an example!

Assume the copy assignment operator + destructor is also defined (due to the rule of three)

```
#include <iostream>
using namespace std;
class MyObj {
    public:
        int a;
        MyObj(int x) {
            cout << "calling og constructor \n";
            a=x;
        }
        MyObj(const MyObj &obj) { // copy constructor
            cout << "calling copy constructor \n";
            a = obj.a;
        }
};
```

x86-64 gcc 13.2 -std=c++17 -O0 -w -fno-elide-constructors

C++23 produces the same results for these examples

```cpp
// 0 and 1 do not work the same way ... why
MyObj example0(){
    return MyObj(3);
}

MyObj example1(){
    // no copy constructor called for line 8. Equivalent to a(MyObj(3))
    MyObj a = MyObj(3);
    return a;
}

int main()
{
    cout<< "___ calling ex0 \n";
    MyObj e0 = example0();
    cout<< "___ calling ex1 \n";
    MyObj e1 = example1();
    return 0;
}
```

```
___ calling ex0
calling og constructor
```

-std=c++17 -O0 -w -fno-elide-constructors

```
___ calling ex0
calling og constructor
calling copy constructor
calling copy constructor
___ calling ex1
calling og constructor
calling copy constructor
calling copy constructor
calling copy constructor
```

-std=c++11 -O0 -w -fno-elide-construct

Depending on the compiler & C++ version - there might be a different number of copies

```
1   // 0 and 1 do not work the same way ... why
2   MyObj example0(){
3       return MyObj(3);
4   }
5
6   MyObj example1(){
7       // no copy constructor called for line 8. Equivalent to a(MyObj(3))
8       MyObj a = MyObj(3);
9       return a;
10  }
11
12  int main()
13  {
14      cout<< "___ calling ex0 \n";
15      MyObj e0 = example0();
16      cout<< "___ calling ex1 \n";
17      MyObj e1 = example1();
18      return 0;
19  }
```

Copy initialize the return value from a

All the possible copies for example1()...

```
___ calling ex1
calling og constructor
calling copy constructor
calling copy constructor
calling copy constructor
```

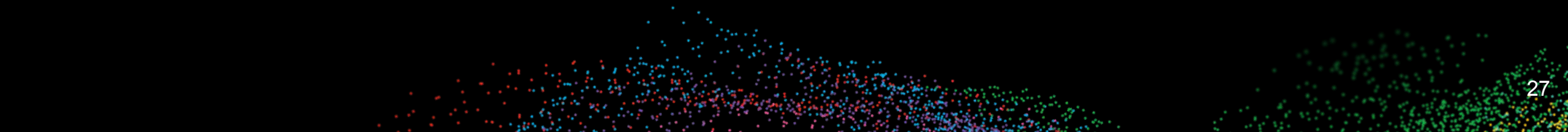
-std=c++11 -O0 -w -fno-elide-constructo

```
1   // 0 and 1 do not work the same way ... why
2   MyObj example0(){
3       return MyObj(3);
4   }
5
6   MyObj example1(){
7       // no copy constructor called for line 8. Equivalent to a(MyObj(3))
8       MyObj a = MyObj(3);
9       return a;
10  }
11
12  int main()
13  {
14      cout<< "___ calling ex0 \n";
15      MyObj e0 = example0();
16      cout<< "___ calling ex1 \n";
17      MyObj e1 = example1();
18      return 0;
19  }
```

And what if we removed the flag?

```
___ calling ex0
calling og constructor
___ calling ex1
calling og constructor
```

… But we've forgotten something …



```cpp
#include <iostream>
using namespace std;
class MyObj {
    public:
        int a;
        MyObj(int x) {
            cout << "calling og constructor \n";
            a=x;
        }
        MyObj(const MyObj &obj) { // copy constructor
            cout << "calling copy constructor \n";
            a = obj.a;
        }
};
```

Add a move constructor … since there is no default one

```cpp
#include <iostream>
using namespace std;
class MyObj {
   public:
        int *a;  // raw pointer
        MyObj(int x) {
            cout << "calling og constructor \n";
            a = new int;
            *a = x;
        }
        MyObj(const MyObj &obj): MyObj{ *obj.a } { // copy constructor
            cout << "calling copy constructor \n" ;
            // deep copy
        }
        MyObj(MyObj &&obj): a{ obj.a } { // move constructor
            cout << "calling move constructor \n";
            obj.a = nullptr;
        }
};
```

```cpp
// 0 and 1 do not work the same way ... why
MyObj example0(){
    return MyObj(3);
}

MyObj example1(){
    // no copy constructor called for line 8 - Equivalent to a(MyObj(3))
}

int main()
{
    cout<< "___ calling ex0 \n";
    MyObj e0 = example0();
    cout<< "___ calling ex1 \n";
    MyObj e1 = example1();
    return 0;
}
```

```
___ calling ex0
calling og constructor
___ calling ex1
calling og constructor
calling move constructor
```

-std=c++17 -O0 -w -fno-elide-constructors

```
___ calling ex0
calling og constructor
calling move constructor
calling move constructor
___ calling ex1
calling og constructor
calling move constructor
calling move constructor
calling move constructor
```

-std=c++11 -O0 -w -fno-elide-constructo

It's the same thing - but the move constructor gets called!

Depending on the compiler & C++ version - there might be a different number of copies

So, does RVO even really matter if we have move?

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

class MyObj {
   public:
        int a;
        int* b;
        MyObj(int x) {
            cout << "calling og constructor\n";
            a = x;
            b = new int[2];
        }
};

MyObj example1(){
    MyObj a = MyObj(42);
    return a;
}

int main()
{
    for (int i=0; i <100000; ++i)
    {
        MyObj a = example1();
    }
}
```

```
g++ -std=c++17 -O0 -fno-elide-constructors nrvo.cpp -o nrvoOFF
g++ -std=c++17 -O0  nrvo.cpp -o nrvoON
```

Bloomberg

Engineering

So, does RVO even really matter if we have move?

No RVO

```
real    0m7.846s
user    0m0.043s
sys     0m0.217s
```

```
real    0m6.022s
user    0m0.035s
sys     0m0.182s
```

```
real    0m5.921s
user    0m0.073s
sys     0m0.215s
```

26.24% Speedup

YES!

```
int main()
{
    for (int i=0; i <100000; ++i)
    {
        MyObj a = example1();
    }
}
```

RVO

```
real    0m5.857s
user    0m0.017s
sys     0m0.213s
```

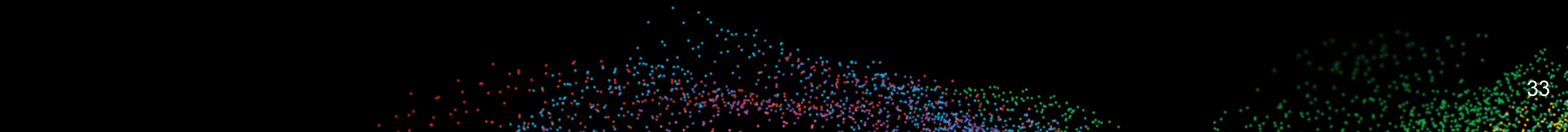
```
real    0m5.956s
user    0m0.026s
sys     0m0.172s
```

```
real    0m5.985s
user    0m0.030s
sys     0m0.155s
```

Bloomberg

Engineering

So when does (U/N)RVO not work?



When certain optimizations are disabled

So … we can turn off (N)RVO 😈…

(Doesn't work for C++ 17 (and onwards) *U*RVO)

-fno-elide-constructors

… for gcc & clang

```
-fno-elide-constructors
    The C++ standard allows an implementation to omit creating a
    temporary which is only used to initialize another object of the
    same type.  Specifying this option disables that optimization, and
    forces G++ to call the copy constructor in all cases.
```

When the object construction happens outside the scope of the current function

```
MyObj a = MyObj(3);
MyObj example1(){
    /* NO RVO!
       if returning a param,
       a global var, etc.
    */
    return a;
}
```

When the return type is not the same as what's being returned

```
class MyObjChild:public  MyObj {
    public:
        MyObjChild(){…
        }
};

MyObj inheritanceFailsRVO(){
    return MyObjChild();
}
```

When there are multiple return statements returning different objects (for NRVO only)

```
MyObj example2(){
    /* 2 possible return values
       of the same type */
    MyObj x1 = MyObj(3);
    MyObj x2 = MyObj(3);
    int a = rand() % 100;
    if (a > 5) {
        return x1;
    }
    return x2;
}
```

When you're returning a complex expression (for NRVO only)

```
MyObj example4_contd(){
    MyObj x1 = MyObj(3);
    MyObj x2 = MyObj(3);
    int a =  rand() % 100;
    return std::move(a > 50? x1 : x2);
}
```

For not guaranteed RVO (e.g., NRVO): either move or copy is sufficient

For guaranteed RVO (e.g., URVO): no move or copy constructor is required

```
class MyObj {
    public:
        MyObj(int x) {
            cout << "calling og constructor \n";
        }
        MyObj(const MyObj &obj) = delete;
        MyObj(MyObj &&obj) = delete;
};
```


```
MyObj example1(){
MyObj example0(){
    return MyObj(3);
}
```

Is RVO always applied?

Let’s go through some examples!

Setup for each example:

```
int main()
{
    MyObj e0 = example0();
    return 0;
}
```

… with the move+copy constructors we had previously…

Example 1

```
MyObj example1(){
    MyObj a = MyObj(3);
    // we change a
    *a.a = rand() % 100;
    return a;
}
```

```
calling og constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling move constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 1.2

```
MyObj example1(){
    auto a = MyObj(3);
    // we change a
    *a.a = rand() % 100;
    return a;
}
```

```
calling og constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling move constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 1.3

```
MyObj example1(){
    MyObj a = MyObj(3);
    return std::move(a);
}
```

```
calling og constructor
calling move constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling move constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 2

```
MyObj example2(){
    /* 2 possible return values
       of the same type */
    MyObj x1 = MyObj(3);
    MyObj x2 = MyObj(3);
    int a = rand() % 100;
    if (a > 5) {
        return x1;
    }
    return x2;
}
```

```
calling og constructor
calling og constructor
calling move constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling og constructor
calling move constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 3

```
const MyObj example3(){
    MyObj x1 = MyObj(3);
    return x1;
}
```

```
calling og constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling move constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 4

```
MyObj example4(){
    MyObj x1 = MyObj(3);
    MyObj x2 = MyObj(3);
    int a =  rand() % 100;
    return (a > 50? x1 : x2);
}
```

```
calling og constructor
calling og constructor
calling og constructor
calling copy constructor
```

With -fno-elide-constructors

```
calling og constructor
calling og constructor
calling og constructor
calling copy constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

Example 4 (continued)

```
MyObj example4(){
    MyObj x1 = MyObj(3);
    MyObj x2 = MyObj(3);
    int a =  rand() % 100;
    /* write the lvalue expression
       outside the return */
    (a > 50? x1 : x2);
    return x1;
}
```

```
calling og constructor
calling og constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling og constructor
calling move constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

YOU SHOULD HAVE USED A MOVE HERE!!!

Example 4 (continued)

```
MyObj example4_contd(){
    MyObj x1 = MyObj(3);
    MyObj x2 = MyObj(3);
    int a =  rand() % 100;
    return std::move(a > 50? x1 : x2);
}
```

```
calling og constructor
calling og constructor
calling move constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling og constructor
calling move constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

OR USE URVO!!!

Example 4 (continued)

```
MyObj exampleWowSoJustUVRO(){
    int a =  rand() % 100;
    return (a > 50? MyObj(3) : MyObj(5));
}
```

```
calling og constructor
```

With `-fno-elide-constructors`

```
calling og constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

If you hate ternaries . . .

Example 4 (continued)

```
MyObj example5_(){
    int a = rand() % 100;
    if (a < 5) {
        return MyObj(3);
    } else{
        return MyObj(8);
    }
}
```

```
calling og constructor
```

With `-fno-elide-constructors`

```
calling og constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 5

```
MyObj example5(){
    // return the same object
    MyObj x1 = MyObj(3);
    int a = rand() % 100;
    if (a > 5) {
        *(x1.a) = a;
        return x1;
    } else {
        *(x1.a) = 7;
        return x1;
    }
}
```


```
calling og constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling move constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

Example 5.2 - inside the if branch

```
MyObj example5(){
    int a = rand() % 100;
    if (a > 5) {
        MyObj x1 = MyObj(3);
        return x1;
    } else {
        MyObj x1 = MyObj(3);
        return x1;
    }
}
```

```
calling og constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling move constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

Example 5.4

```
MyObj example5(){
    int a = rand() % 100;
    if (a > 5) {
        MyObj x1 = MyObj(3);
        return x1;
    } else {
        MyObj x2 = MyObj(3);
        return x2;
    }
}
```

```
calling og constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling move constructor
```

x86-64 clang 18.1.0 -std=c++17 -O0

Example 6

Slightly out of scope because it's a "throw" and not a "return," but it is good to be aware

```
MyObj example6(){
    MyObj a = MyObj(3);
    throw a;
}
```

```
calling og constructor
calling move constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling move constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 6.5

```
MyObj example6_5(){
    try {
        throw MyObj(3);
    } catch (MyObj& e) {
        cout << "caug
        return e;
    } catch (...) {
        return MyObj(3);
    }
}
```

also fyi: removing this reference still means no RVO


```
calling og constructor
caught object
calling og constructor
calling copy constructor
```

With -fno-elide-constructors

```
calling og constructor
caught object
calling og constructor
calling copy constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

Example 6.5 - without &

```
MyObj example6_2(){
    try {
        throw MyObj(3);
    } catch (MyObj e) {
        cout << "caught object\n";
        return e;
    } catch (...) {
        return MyObj(3);
    }
}
```


```
___ calling ex6_2
calling og constructor
calling og constructor
calling copy constructor
caught object
calling move constructor
```

With -fno-elide-constructors

```
___ calling ex6_2
calling og constructor
calling og constructor
calling copy constructor
caught object
calling move constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

Example 7

```
int main()
{
    MyObj e7 = example7(MyObj(3));
    return 0;
}
```

```
MyObj example7(MyObj a){
    return a;
}
```


```
calling og constructor
calling move constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling move constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 8

```
class MyObj2 {
    public:
        MyObj2() {
            cout << "Calling MyObj2 og constructor" << endl;
        }
        MyObj2(const MyObj2 &obj) { // copy constructor
            cout << "calling MyObj2 copy constructor" << endl;
        }
        MyObj2(MyObj2 &&obj) { // move constructor
            cout << "calling myObj2 move constructor" << endl;
        }
        // assume rule of 5 holds (there are other functions here)
};
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 8

```
class MyObj {
    public:
        MyObj2 b;
        // and same constuctors as previous examples
}
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 8

```
MyObj2 example8(){
    MyObj x1 = MyObj(3);
    return x1.b;
}
```

```
Calling MyObj2 og constructor
calling og constructor
calling MyObj2 copy constructor
```

With `-fno-elide-constructors`

```
Calling MyObj2 og constructor
calling og constructor
calling MyObj2 copy constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

Example 8

```
class MyObj {
   public:
        int *a;  // raw pointer

        MyObj(int x) { …
        }
        MyObj(const MyObj &obj): MyObj{ *obj.a } { …
        }
        MyObj(MyObj &&obj): a{ obj.a } {  …
        }

        MyObj2 getB() {
            MyObj2 b;
            return b;
        }
};
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 8

```
MyObj2 example8(){
    MyObj x1 = MyObj(3);
    return x1.getB();
}
```

```
calling og constructor
Calling MyObj2 og constructor
```

With `-fno-elide-constructors`

```
calling og constructor
Calling MyObj2 og constructor
calling myObj2 move constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

Example 9

```
MyObj example9(){
    MyObj a = MyObj(3);
    return a+=1;
}
```

```
calling og constructor
calling og constructor
calling og constructor
calling copy constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling og constructor
calling og constructor
calling copy constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 10

`volatile` tells the compiler that the value of the variable may change at any time, without any action being taken by the code the compiler finds nearby

```
class  MyObj {
    public:
        int *a;  // raw pointer
        MyObj(int x) {
            cout << "calling og constructor\n";
            a = new int;
            *a = x;
        }

        MyObj(MyObj volatile &obj): MyObj{ *obj.a } { // copy constructor
            cout << "calling copy constructor\n";
        }
        MyObj(MyObj volatile &&obj): a{ obj.a } { // move constructor
            cout << "calling move constructor\n";
            obj.a = nullptr;
        }
}
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 10

```
MyObj example10(){
    MyObj volatile a = MyObj(3);
    return a;
}
```

```
___ calling ex10
calling og constructor
calling og constructor
calling copy constructor
```

With `-fno-elide-constructors`

```
___ calling ex10
calling og constructor
calling og constructor
calling copy constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

Example 10 (continued)

```
MyObj example10(){
    MyObj           a = MyObj(3);
    return a;
}
```

```
___ calling ex10
calling og constructor
```

With `-fno-elide-constructors`

```
___ calling ex10
calling og constructor
calling move constructor
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 11 - structured bindings

```
std::pair<MyObj, MyObj> return2MyObjPair() {
    auto myPair = std::pair(MyObj(3), MyObj(3));
    return myPair;
}

MyObj example11() {
    auto[a, b] = return2MyObjPair();
    std::cout << "after obj a asignment\n";
    return a;
}
```

x86-64 gcc 13.2 -std=c++17 -O0

Example 11 - structured bindings

Instead, you could make two separate functions return *a* and *b* respectively

```
MyObj example11() {
    auto[a, b] = return2MyObjPair();
    std::cout << "after obj a asignment\n";
    return a;
}
```


```
after obj a assignment
calling og constructor
calling copy constructor
```

With `-fno-elide-constructors`

```
after obj a assignment
calling og constructor
calling copy constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

Example 12 - BONUS

```
// example 12
MyObj example12() {
    MyObj a = MyObj(5);
    return (a);
}
```

```
calling og constructor
```

With `-fno-elide-constructors`

```
calling og constructor
calling move constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

WARNING!

If you are writing a copy / move constructor,

never make the constructor do anything else other than a copy/move,

because it can get elided!

Going back to the Standard!

11.9.6 Copy/move elision [class.copy.elision]

1 When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the constructor selected for the copy/move operation and/or the destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object. If the first parameter of the selected constructor is an rvalue reference to the object's type, the destruction of that object occurs when the target would have been destroyed; otherwise, the destruction occurs at the later of the times when the two objects would have been destroyed without the optimization.[104] This elision of copy/move operations, called *copy elision*, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

Copy elision

1.1) — in a `return` statement in a function with a class return type, when the *expression* is the name of a non-volatile object with automatic storage duration (other than a function parameter or a variable introduced by the *exception-declaration* of a *handler* ([except.handle])) with the same type (ignoring cv-qualification) as the function return type, the copy/move operation can be omitted by constructing the object directly into the function call's return object

Return value optimization conditions

1.2) — in a *throw-expression* ([expr.throw]), when the operand is the name of a non-volatile object with automatic storage duration (other than a function or catch-clause parameter) that belongs to a scope that does not contain the innermost enclosing *compound-statement* associated with a *try-block* (if there is one), the copy/move operation can be omitted by constructing the object directly into the exception object

1.3) — in a coroutine, a copy of a coroutine parameter can be omitted and references to that copy replaced with references to the corresponding parameter if the meaning of the program will be unchanged except for the execution of a constructor and destructor for the parameter copy object

1.4) — when the *exception-declaration* of a *handler* ([except.handle]) declares an object of the same type (except for cv-qualification) as the exception object ([except.throw]), the copy operation can be omitted by treating the *exception-declaration* as an alias for the exception object if the meaning of the program will be unchanged except for the execution of constructors and destructors for the object declared by the *exception-declaration*.

[*Note 1*: There cannot be a move from the exception object because it is always an lvalue. — *end note*]

Copy elision is not permitted where an expression is evaluated in a context requiring a constant expression ([expr.const]) and in constant initialization ([basic.start.static]).

[*Note 2*: It is possible that copy elision is performed if the same expression is evaluated in another context. — *end note*]

Source: cppreference.com
https://en.cppreference.com/w/cpp/language/copy_elision

Going back to the Standard!

ways of referring to the same object. If the first parameter of the selected constructor is an rvalue reference to the object's type, the destruction of that object occurs when the target would have been destroyed; otherwise, the destruction occurs at the later of the times when the two objects would have been destroyed without the optimization.[104] This elision of copy/move operations, called *copy elision*, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

— in a `return` statement in a function with a class return type, when the *expression* is the name of a non-volatile object with automatic storage duration (other than a function parameter or a variable introduced by the *exception-declaration* of a *handler* (except handle)) with the same type (ignoring cv-qualification) as the function return type, t
turn object

Storage dura

All objects in a pr

- ***automati***
  extern i
  function pa
- ***static*** stor
  the progra
  global nam
  and static

- ***thread*** st
  deallocate
  declared t
  or extern

When the retu
type is not th
same as wha
being returne

Example 6.5

```
MyObj example6_5(){
    try {
        throw MyObj(3);
    } catch (MyObj& e) {
        cout << "caught object\n";
        return e;
    } catch (...) {
        return MyObj(3);
    }
}
```

No NRVO!

```
calling og constructor
caught object
calling og constructor
calling copy constructor
```

With -fno-elide-constructors

```
calling og constructor
caught object
calling og constructor
calling copy constructor
```

x86-64 gcc 13.2

-std=c++17 -O0

WHEN YOU FINALLY UNDERSTAND

THE C++ STANDARD

Source: Secret Lab Institute (CC-0 license)
https://secretlab.institute/2021/02/15/cc-0-licensed-galaxy-brain-images/

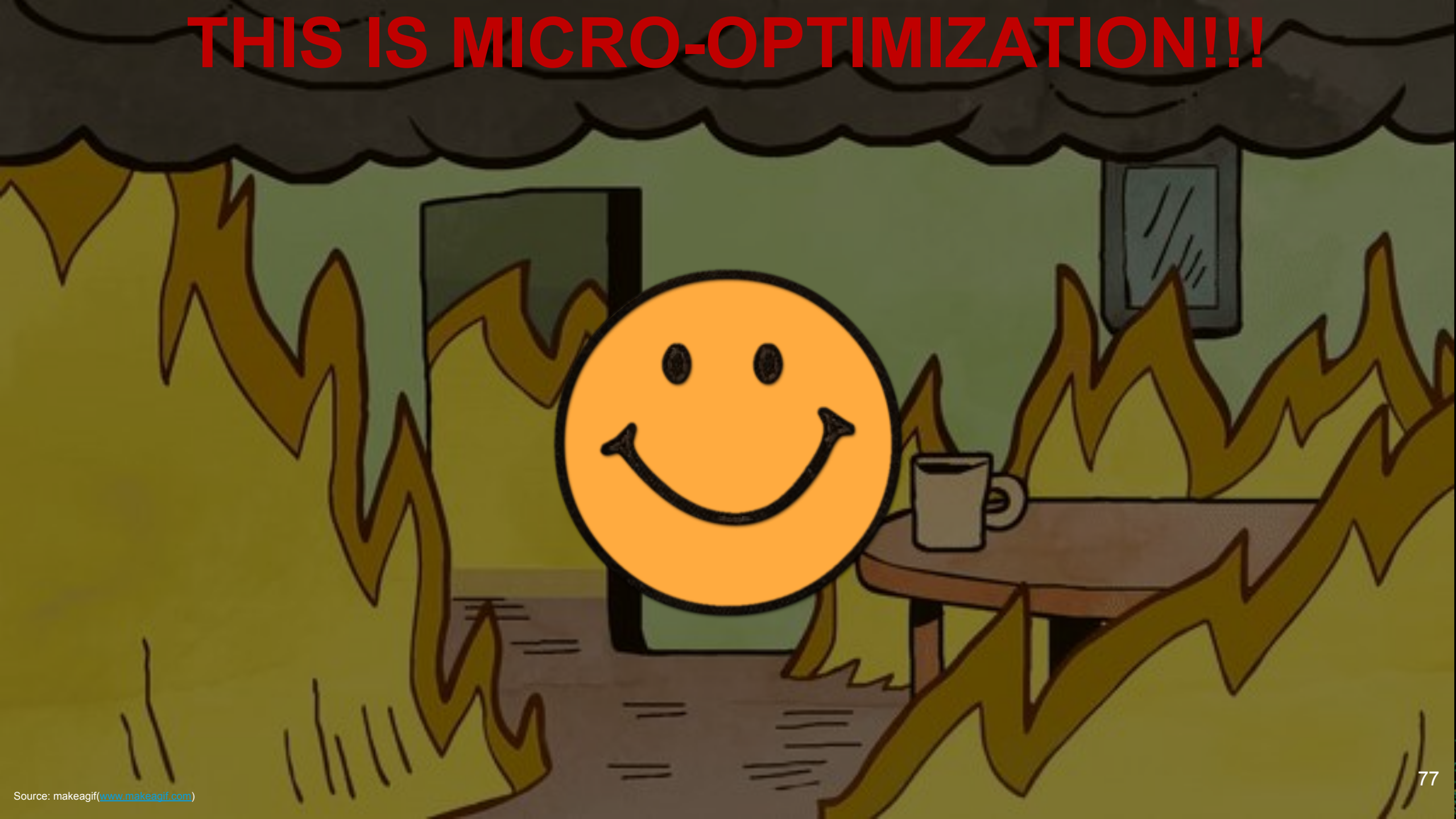
THIS IS A GREAT IDEA!!!

Now - time to go into my codebase and change everything so that return value optimization is supported!

THIS IS A GREAT IDEA!!!

Source: makeagif(www.makeagif.com)

THIS IS MICRO-OPTIMIZATION!!!



Source: makeagif(www.makeagif.com)

THIS IS MICRO-OPTIMIZATION!!!

Source: makeagif(www.makeagif.com)

THIS IS MICRO-OPTIMIZATION!!!

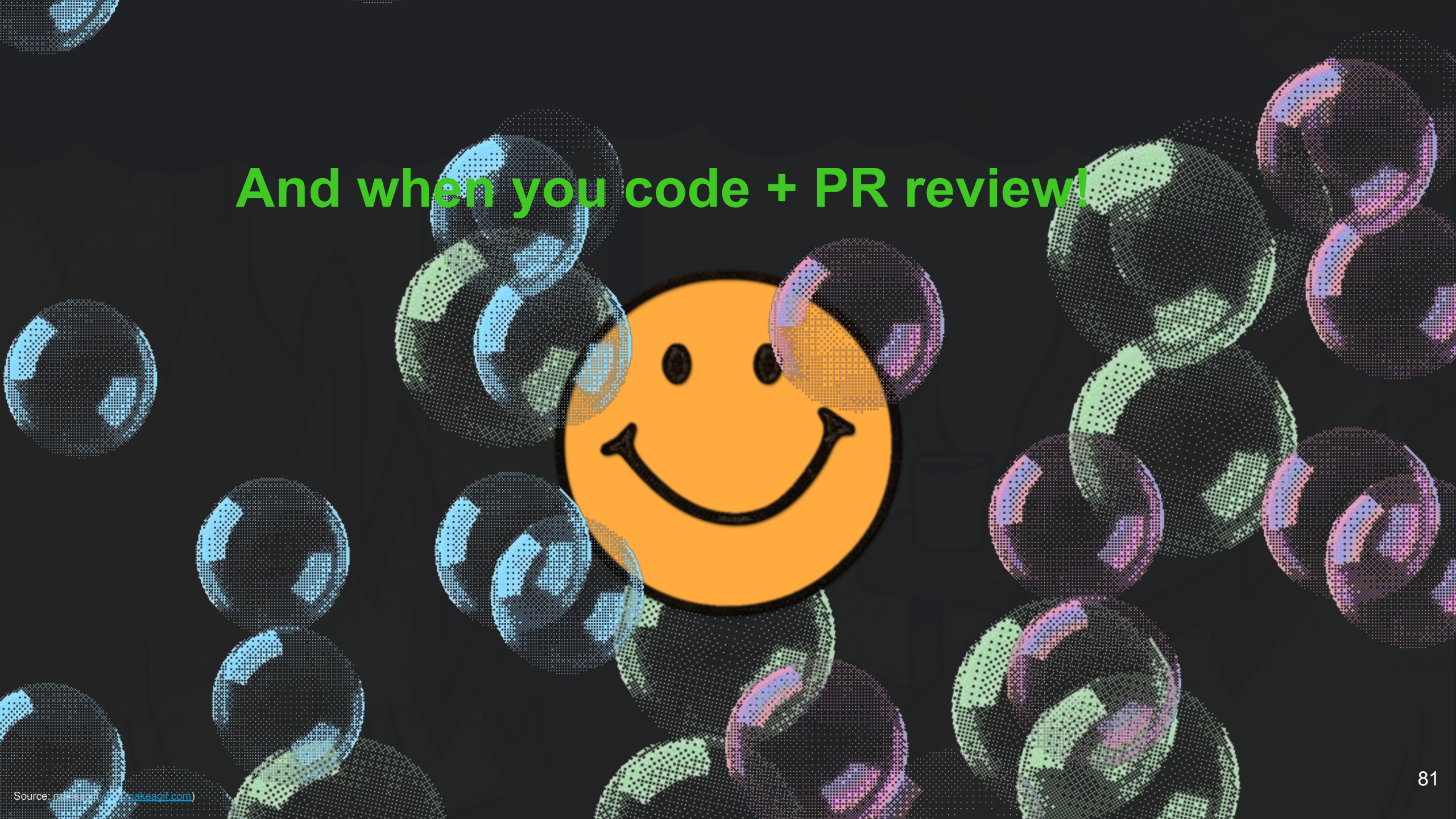
But do it when you see a bottleneck!

Source: makeagif(www.makeagif.com)

Use Profilers!

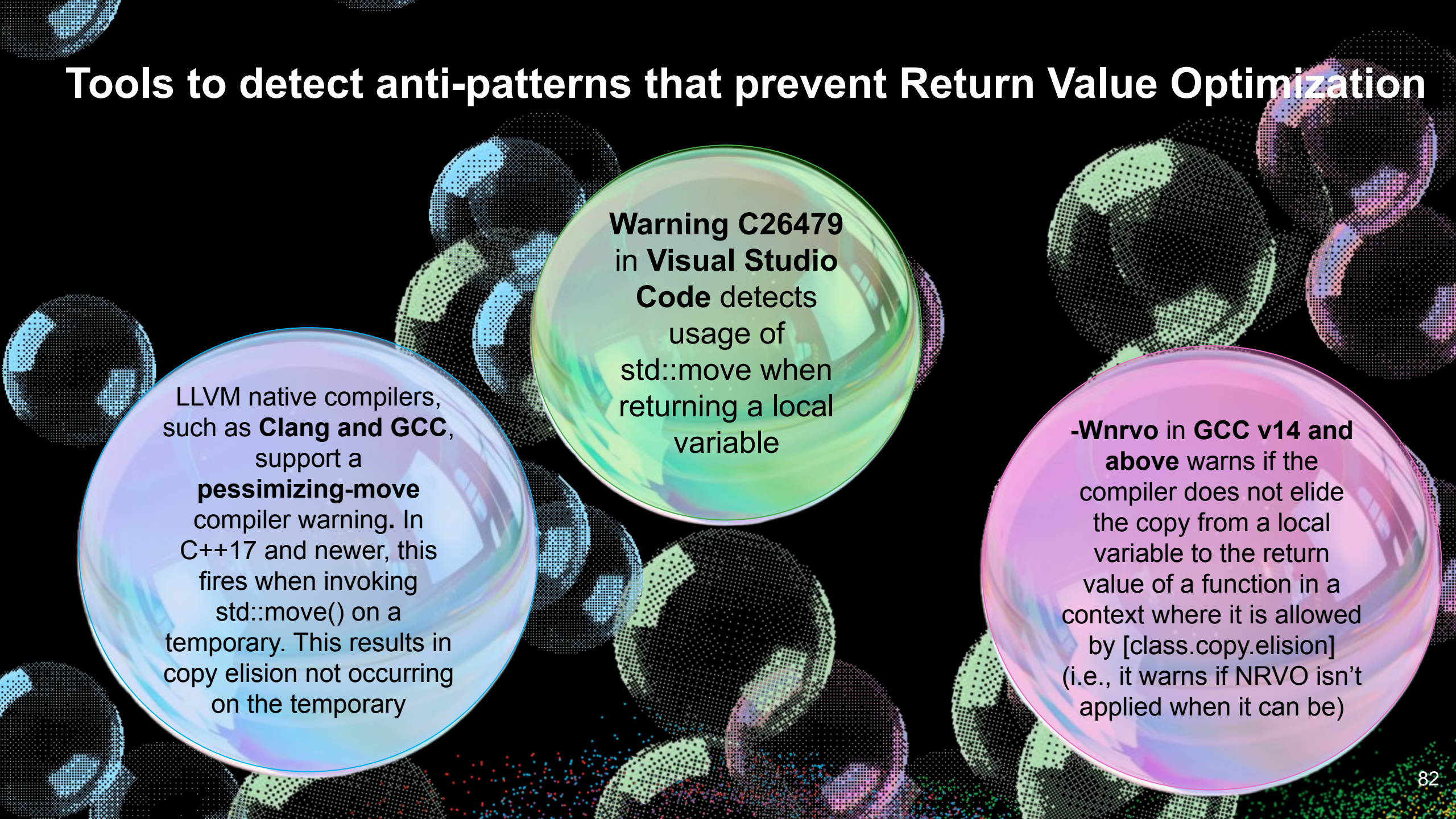
Source: makeagif(www.makeagif.com)

And when you code + PR review!



Source: [redacted] (makeagif.com)

Tools to detect anti-patterns that prevent Return Value Optimization



Conclusion

- An optimization that skips the invocation of a copy constructor when creating an object returned from a function

WHAT IS RVO?

- URVO (guaranteed) & NRVO (not guaranteed)
- Important to know to understand what the compiler guarantees

TYPES OF RVO

- Object constructed outside the current function's scope
- Return type doesn't match function signature
- Multiple return statements (NRVO only)
- Returning complex expressions that are not vanilla lvalue & prvalues
- Using Throw, Volatile, Static, etc.

ANTI-PATTERNS PREVENTING RVO

- Though it depends on the use case, proper usage of RVO can boost performance
- Profile, find the bottlenecks, and use RVO techniques, if applicable
- RVO during everyday coding & PR reviews: Try to mitigate the anti-patterns

HOW TO USE RVO FOR PERFORMANCE

Thank You!

linkedin.com/in/michellefae/

michellefae.github.io/

Playground!

https://tinyurl.com/ICanRVO123