

C++ Club UK

Gleb Dolgich

2019-02-07

Bjarne Stroustrup at UC3M

Celebrating Honoris Causa Doctorate at UC3M by Bjarne Stroustrup

The continuing evolution of C++

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

- ▶ CppCon 2018 - Arthur O'Dwyer - RVO: Harder Than It Looks
- ▶ Trivially Relocatable FAQ

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

cppcon | 2018

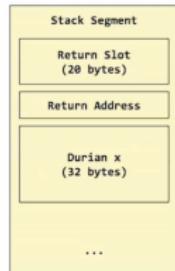
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

Slicing to base class

```
struct Durian : Fruit {  
    double smell;  
};  
  
Fruit slapchop()  
{  
    Durian x = ...;  
    return x;  
}
```



We can't elide this copy because,
while we do control x's physical
location, x is of the "wrong" type for
constructing into the return slot.



34



ARTHUR O'DWYER

Return Value
Optimization:
Harder Than It Looks

CppCon.org

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

Rules of thumb for RVO

Here's what I used to tell people when they asked about RVO:

- "Unnamed RVO" (URVO): Returning a temporary (a prvalue) will trigger copy-elision.

```
return Fruit{1,2,3,4,5};  
return my_helper_function();
```

- "Named RVO" (NRVO): Returning a local variable "by name" will trigger copy-elision, except in the corner cases we've covered.

```
return x;
```

- And even if copy-elision *doesn't* happen, *implicit move* happens...



ARTHUR O'DWYER

Return Value Optimization:
Harder Than It Looks

CppCon.org

A photograph of Arthur O'Dwyer, a man with glasses and a beard, speaking at a podium during a CppCon 2018 presentation. He is wearing a maroon shirt. The podium has a laptop and microphones. The background is dark. To the right of the photo is a slide with text and a title.

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

Moving into the return slot

```
struct Fruit {  
    int data[5];  
    Fruit(Fruit&&);  
};  
  
Fruit apples_and_oranges(bool condition)  
{  
    Fruit x = ...;  
    Fruit y = ...;  
    return std::move(condition ? x : y);  
}
```



Return Value
Optimization:
Harder Than It Looks

CppCon.org

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

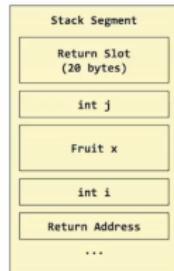
When can't we elide?

```
Fruit apples_to_apples(int i, Fruit x, int j)
{
    return x;
}
```

In this (slightly altered) example, the caller passes in `Fruit x` at one stack address, and the return slot at a different stack address.

We must get the data out of `x` and into the return slot somehow!

We can't elide the copy because we don't control `x`'s physical location.



Return Value
Optimization:
Harder Than It Looks

CppCon.org

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

Rules of thumb for RVO

Here's what I used to tell people when they asked about RVO:

- "Implicit move": When returning a local variable "by name" doesn't trigger copy-elision, the compiler's overload resolution will still automatically treat the name *x* as an rvalue!

```
std::string identity(std::string x) {  
    return x; // x will be implicitly moved-from, not copied!  
}
```

- Because of C++11's "implicit move," writing `return std::move(x)` is almost always a pessimization — it never helps, and it might hurt by disabling NRVO.



ARTHUR O'DWYER

Return Value Optimization:
Harder Than It Looks

CppCon.org

A photograph of Arthur O'Dwyer, a man with a beard and glasses, speaking at a podium during a CppCon 2018 presentation. He is wearing a maroon shirt with a yellow logo. The podium has a laptop and some papers on it. The background is dark, and the CppCon 2018 logo is visible at the top right. A text overlay on the right side of the slide identifies him as Arthur O'Dwyer and provides the title of his talk: "Return Value Optimization: Harder Than It Looks".

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

The actual rules of “implicit move”

Before 2013, the rules were slightly different — but even the C++11 rules were posthumously amended by Defect Report [CWG1579](#).

`[class.copy.elision] /3` If the *expression* in a return statement is a (possibly parenthesized) *id-expression* that names an object with automatic storage duration declared in the body or *parameter-declaration-clause* of the innermost enclosing function or *lambda-expression* [...] overload resolution to select the constructor for the copy is first performed as if the object were designated by an *rvalue*. If the first overload resolution fails or was not performed, or if the type of the first parameter of the selected constructor is not an *rvalue reference* to the object’s type (possibly *cv-qualified*), overload resolution is performed again, considering the object as an *lvalue*.

45

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

A photograph of Arthur O'Dwyer, a man with glasses and a beard, speaking at a podium. He is wearing a red shirt and has a microphone attached to his shirt. A laptop is on the podium in front of him.

ARTHUR O'DWYER

Return Value
Optimization:
Harder Than It Looks

CppCon.org

9 / 44

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

What went wrong here? Slicing.

```
Cexpr as_cexpr() const override {
    CexprList cfg;
    cfg.append(CexprInt(17));
    cfg.append(CexprInt(42));
    cfg.append(CexprString("hike"));
    return cfg;
}
```

The constructor that we want to call here
is Cexpr(Cexpr&).

But cfg's type is CexprList, not Cexpr!

"If the first overload resolution fails or was
not performed, or if the type of the first
parameter of the selected constructor is
not an rvalue reference to **the object's**
type (possibly cv-qualified), overload
resolution is performed again, considering
the object as an lvalue."

The slicing here is 100% intentional; but it
still silently disables the optimization we
thought the compiler was giving us.

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



ARTHUR O'DWYER

Return Value
Optimization:
Harder Than It Looks

CppCon.org

46

10 / 44

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

The image shows a presentation slide from CppCon 2018. At the top right is the conference logo: "cppcon | 2018" with "THE C++ CONFERENCE • BELLEVUE, WASHINGTON" underneath. On the left, a white rectangular box contains the title "Why are constructors so important?". Below it is a block of C++ code:

```
std::unique_ptr<ConfigManager> create() {  
    auto p = std::make_unique<ConfigManagerImpl>();  
    return p;  
}
```

A small number "53" is visible in the bottom right corner of the white box. To the right of the white box is a dark sidebar containing the speaker's name, title, and website.

ARTHUR O'DWYER

Return Value
Optimization:
Harder Than It Looks

CppCon.org

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

Example 1

```
void five() {
    Widget w;
    throw w;
}

Widget six(Widget w) {
    return w;
}

void seven(Widget w) {
    throw w;
}
```

P1155 proposes to
make seven an
implicit move.

	Copy elision?	Implicit move?	Plain old copy?
Technically permitted	✓		
	✓		
	Clang, MSVC, Intel	✓	

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



ARTHUR O'DWYER

Return Value
Optimization:
Harder Than It Looks

CppCon.org

72

12 / 44

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

Example 2

```
struct From {
    From(Widget const&);
    From(Widget&&);

};

struct To {
    operator Widget() const;
    operator Widget() &&;
};

From eight() {
    Widget w;
    return w;
}

Widget nine() {
    To t;
    return t;
}
```

Copy elision?	Implicit move?	Plain old copy?
	✓	
		✓

P1155 proposes to make both of these do implicit move.

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



ARTHUR O'DWYER

Return Value Optimization:
Harder Than It Looks

CppCon.org

74

13 / 44

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

Example 3

```
struct Fish {
    From(Widget const&);
    From(Widget&&);
};

struct Fowl {
    Fowl(Widget);
}

Fish ten() {
    Widget w;
    return w;
}

Fowl eleven() {
    Widget w;
    return w;
}
```

Copy elision?	Implicit move?	Plain old copy?
	✓	
	GCC	✓

P1155 proposes to make both of these do implicit move.

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

A photograph of Arthur O'Dwyer, a man with a beard and glasses, speaking at a podium. He is wearing a maroon shirt. A laptop is on the podium in front of him. The background is dark.

ARTHUR O'DWYER

Return Value Optimization:
Harder Than It Looks

76
CppCon.org

14 / 44

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

Example 4

```
struct Base {  
    Base(Base const&);  
    Base(Base&&);  
};  
struct Derived : Base {};  
  
unique_ptr<Base> twelve() {  
    unique_ptr<Derived> p;  
    return p;  
}  
  
Base thirteen() {  
    Derived d;  
    return d;  
}
```

Copy elision?	Implicit move?	Plain old copy?
	✓	
	GCC, Intel	✓

P1155 proposes to make both of these do implicit move.

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



ARTHUR O'DWYER

Return Value Optimization:
Harder Than It Looks

CppCon.org

78

15 / 44

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

Example 5

```
Widget fourteen(Widget&& w) {  
    return w;  
}  
  
Widget fifteen(Widget& w) {  
    Widget&& x = std::move(w);  
    return x;  
}
```

Copy elision?	Implicit move?	Plain old copy?
		✓
		✓

P0527 proposes to
make both of these
do implicit move,
instead of copy.

80



ARTHUR O'DWYER

Return Value
Optimization:
Harder Than It Looks

CppCon.org

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

Example 6

```
Widget sixteen(Widget w) {  
    w += 1;  
    return w;  
}  
  
Widget seventeen(Widget w) {  
    return w += 1;  
}
```

Copy elision?	Implicit move?	Plain old copy?
	✓	
		✓

There's no proposal
for this one yet,
as far as I know.

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



ARTHUR O'DWYER

Return Value
Optimization:
Harder Than It Looks

CppCon.org

82

17 / 44

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"



The image shows Arthur O'Dwyer, a man with a beard and glasses, speaking at a podium during a CppCon 2018 session. He is wearing a red shirt and is positioned behind a black podium with a silver laptop on it. A microphone is attached to the podium. The background is dark, and the CppCon 2018 logo is visible at the top right.

Moving into the return slot

```
struct Fruit {  
    int data[5];  
    Fruit(Fruit&&);  
};  
  
Fruit apples_and_oranges(bool condition)  
{  
    Fruit x = ...;  
    Fruit y = ...;  
    return std::move(condition ? x : y);  
}
```

25

ARTHUR O'DWYER

Return Value
Optimization:
Harder Than It Looks

CppCon.org

CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

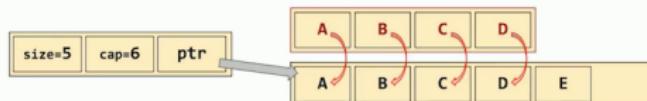
cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

Motivating “relocation”

Consider what happens when we resize a `std::vector<T>`.

```
std::vector<T> vec { A, B, C, D };
vec.push_back(E);
```



The “relocation” of objects A, B, C, D involves 4 calls to the move-constructor, followed by 4 calls to the destructor.

6



Trivially
Relocatable

CppCon.org

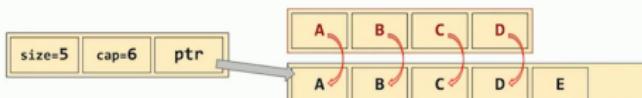
CppCon 2018: Arthur O'Dwyer "RVO: Harder Than It Looks"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON

Relocating non-trivial types

In principle, we *can* implement the "relocation" of objects A, B, C, D here with a simple `memcpy`. `shared_ptr`'s move constructor is non-trivial, and its destructor is also non-trivial, but if we always call them together, the *result* is tantamount to `memcpy`.



The operation of "calling the move-constructor and the destructor together in pairs" is known as *relocation*.

A type whose relocation operation is tantamount to `memcpy` is *trivially relocatable*.



Trivially
Relocatable

CppCon.org

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

CppCon 2018 - Jason Turner - Surprises in Object Lifetime

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



JASON TURNER

Surprises
in
Object Lifetime

CppCon.org

What's Returned From Main?

```
1 struct S {
2     const int& m;
3 };
4
5 int main() {
6     const S& s = S{1};
7     return s.m;
8 }
```

Copyright Jason Turner

@lefticus

6.5

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

The image shows a screenshot of a presentation slide from CppCon 2018. On the left, there is a video feed of Jason Turner speaking. To his right is a large text box containing the word "Surprise!" and a subtitle. Below the video feed, there is a title card with the author's name and the topic. At the bottom, there is footer information and a navigation bar.

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

JASON TURNER

**Surprises
in
Object Lifetime**

Lifetime extension rules apply recursively to member initializers

Copyright Jason Turner @lefticus 6.7

CppCon.org

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

The image is a screenshot of a presentation slide from CppCon 2018. On the left, there is a video feed of Jason Turner speaking. He is a man with glasses, wearing a light-colored button-down shirt, and is gesturing with his right hand while holding a white object in his left hand. To the right of the video feed is a large text box containing the word "Surprise!" in white. Below this, a gray box contains the text: "Complex rules allow for the lifetime extension of temporaries that are assigned to references (See: [class.temporary])". At the bottom of the slide, there is a footer bar with the text "JASON TURNER" on the left, "Surprises in Object Lifetime" in the center, "Copyright Jason Turner" and "@lefticus" on the bottom left, and "6.4" on the bottom right.

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

JASON TURNER

Surprises
in
Object Lifetime

Copyright Jason Turner @lefticus 6.4

CppCon.org

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

The slide is from CppCon 2018, featuring a photo of Jason Turner on the left and a presentation slide on the right.

Left Side:

- A photo of Jason Turner, a man with short hair wearing a light-colored striped shirt, standing and looking towards the right.
- A black rectangular overlay with the text "JASON TURNER" in white capital letters.
- A black rectangular overlay with the text "Surprises in Object Lifetime" in white capital letters.
- A yellow rectangular overlay at the bottom with the text "CppCon.org" in black.

Right Side:

- A dark rectangular box containing the following text:

```
std::initializer_list<>
1 // how many dynamic allocations are there?
2 std::vector<std::string> vec{"a", "b"};
```
- A text block below the code:

Almost certainly only 1. Every standard library implements a “Small String Optimization,” so only the `vector` needs an allocation.
- Small text at the bottom of the slide:

Copyright Jason Turner @lefticus 7.2

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

JASON TURNER

Surprises
in
Object Lifetime

```
std::initializer_list<>
1 // how many dynamic allocations are there?
2 std::vector<std::string> vec{"a long string of characters",
3                                "b long string of characters"};
```

5

Copyright Jason Turner @lefticus 7.4

CppCon.org

CppCon 2018: Jason Turner "Surprises in Object Lifetime"



cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

JASON TURNER

**Surprises
in
Object Lifetime**

CppCon.org

`std::initializer_list<>`

`initializer_list` is implemented by creating a hidden array for you, of the expected type, that is `const`.

So this is the approximate equivalence.

```
1 const std::string __data[]{"a long string of characters",
2                             "b long string of characters"};
3 std::vector<std::string> vec{initializer_list<std::string>{__data,
4                               __data + 2}};
```

Copyright Jason Turner

@lefticus

7.5

27 / 44

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



JASON TURNER

Surprises in Object Lifetime

CppCon.org

std::initializer_list<>

```
1 const std::string __data[]{"a long string of characters", // alloc 1
                           "b long string of characters"}; // alloc 2
2
3 // vector: alloc 3
4 // copy of str1: alloc 4
5 // copy of str2: alloc 5
6 std::vector<std::string> vec{std::string{__data,
                                         __data + 2}};
```

Copyright Jason Turner

@lefticus

7.6

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

The image shows a presentation slide from CppCon 2018. On the left, there is a video feed of Jason Turner speaking at a podium. Below the video, his name 'JASON TURNER' is displayed in a white box. To the right of the video, the title 'Surprises in Object Lifetime' is shown in a large white font inside a dark rectangular box. Below the title, a text box contains the sentence: `std::initializer_list<...>` invocations create hidden `const` arrays. At the bottom of the slide, there is copyright information: 'Copyright Jason Turner' and '@lefticus'. The slide number '7.7' is also present. At the very bottom, the URL 'CppCon.org' is visible.

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

JASON TURNER

Surprises
in
Object Lifetime

Copyright Jason Turner @lefticus 7.7

CppCon.org

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



JASON TURNER

Surprises in Object Lifetime

CppCon.org

std::array<>

```
1 // how many dynamic allocations are done?
2 // (C++17 class template type deduction)
3 std::array a{"a long string of characters", "b long string of characters"};
```

0 ... Why?

Type of `std::array` is deduced as `std::array<const char *, 2>`.

Copyright Jason Turner

@lefticus

7.8

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

The slide is from the CppCon 2018 conference, featuring a speaker named Jason Turner. The title of the talk is "Surprises in Object Lifetime". The slide content includes a code snippet demonstrating std::array, a question about dynamic allocations, and a poll asking for the number of allocations.

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

JASON TURNER

**Surprises
in
Object Lifetime**

std::array<>

```
1 // how many dynamic allocations are done?  
2 std::array<std::string, 2> a{"a long string of characters",  
3 "b long string of characters"};
```

Exactly 2.
What's the difference?

Copyright Jason Turner @lefticus 7.10

CppCon.org

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



JASON TURNER

Surprises in Object Lifetime

CppCon.org

std::array<>

std::array<> has no constructors, it is effectively something like:

```
1 template<typename T, std::size_t Size>
2 struct array
3 {
4     T _M_elems[Size];
5 };
```

So with std::array our “Initializer List” initialization does not use an initializer list, it directly initializes the internal data structure.

This is literally the most efficient thing possible!

Copyright Jason Turner

@lefticus

7.11

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



JASON TURNER

Surprises
in
Object Lifetime

CppCon.org

if-init

What warning might this give?

```
1 int get_val();
2 double get_other_val();
3
4 int main() {
5     if (const auto x = get_val(); x > 5) {
6         // do something with x
7     } else if (const auto x = get_other_val(); x < 5) {
8         // do something else with x
9     }
10 }
```

x shadows previous declaration of x (or similar)

Copyright Jason Turner

@lefticus

9.2

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

The image is a screenshot of a presentation slide from CppCon 2018. The slide features a dark background with a blue header bar. In the top left corner, there is a small video window showing Jason Turner speaking. To the right of the video, the text "Surprise!" is displayed in a large white font. Below "Surprise!", a grey rectangular box contains the text "If-init statements are visible for the `else` blocks as well". At the bottom of the slide, there is copyright information: "Copyright Jason Turner" and "@lefticus". On the far right, the number "9.6" is visible. The overall layout is clean and professional, typical of a technical conference presentation.

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

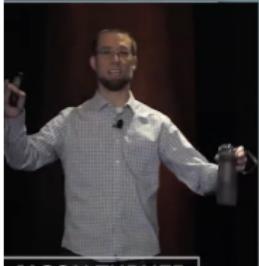
JASON TURNER

Surprises
in
Object Lifetime

Copyright Jason Turner @lefticus 9.6

CppCon.org

CppCon 2018: Jason Turner "Surprises in Object Lifetime"



The image shows Jason Turner, a man with glasses and a beard, wearing a light-colored button-down shirt, standing on a stage and speaking. He is holding a small device in his left hand and a remote or pointer in his right hand.

JASON TURNER

Surprises
in
Object Lifetime

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

Is The Destructor Called?

```
1 #include <cstdio>
2
3 struct S {
4     int i{};
5     S() = default;
6     S(int i) : i(i)
7     { throw 1; } //!
8     ~S() { puts("~S()"); }
9 };
10
11 int main() {
12     try {
13         S s(1);
14     } catch (...) {
15     }
16 }
```

Copyright Jason Turner @lefticus 13.3

CppCon.org

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



JASON TURNER

Surprises
in
Object Lifetime

CppCon.org

Is The Destructor Called?

```
1 #include <cstdio>
2
3 struct S {
4     int i{};
5     S() = default;
6     ~S(int i)
7         : S{} //!
8         { i = i; throw 1; }
9     ~S() { puts("~S()"); }
10 };
11
12 int main() {
13     try {
14         S s{1};
15     } catch (...) {
16     }
17 }
```

Yes!

Why?

Copyright Jason Turner

@lefticus

13.5

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



JASON TURNER

Surprises
in
Object Lifetime

CppCon.org

Is The Destructor Called?

Can be used in interesting ways (From Howard Hinnant)

```
1 struct S {
2     int *ptr=nullptr;
3     int *ptr2=nullptr;
4     S() = default;
5     S(int val1, int val2) : S{} /// make sure d'tor is called
6     {
7         ptr = new int(val1);
8         ptr2 = new int(val2);
9     }
10    ~S() { delete ptr; delete ptr2; } /// delete nullptr is well defined
11};
```

OF COURSE DON'T DO THIS, USE unique_ptr instead

Copyright Jason Turner

@lefticus

13.7

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

JASON TURNER

**Surprises
in
Object Lifetime**

Consider Requiring All Structured Bindings To Be &

```
1 auto get_sum()
2 {
3     // const & works well with lifetime extension rules
4     // and makes it clear we are actually playing with
5     // hidden references
6     const auto &[first, second] = get_pair();
7     return first + second;
8 }
```

Copyright Jason Turner @lefticus 16.3 CppCon.org

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON



JASON TURNER

Carefully Use `initializer_list<>`

- Understand the difference between an *Initializer List* and an `initializer_list<>` (note that [dcl.init.list] has 12 subclauses)
- Take advantage of direct initialization for type safety and performance
- Only use `initializer_list<>` constructors for `trivial` or `literal` types

Copyright Jason Turner @lefticus 17.4

Surprises
in
Object Lifetime

CppCon.org

CppCon 2018: Jason Turner "Surprises in Object Lifetime"

cppcon | 2018

THE C++ CONFERENCE • BELLEVUE, WASHINGTON



JASON TURNER

Surprises
in
Object Lifetime

CppCon.org

constexpr All The Things

What would this return?

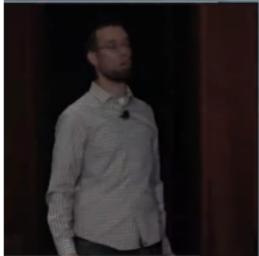
```
1 int & get_val() {  
2     int i;  
3     return i; // dandling reference  
4 }  
5  
6 int do_thing() {  
7     return ++get_val(); // invalid dereference  
8 }  
9  
10 int main() {  
11     auto val = do_thing();  
12     return val; // unknown  
13 }
```

Copyright Jason Turner

@lefticus

17.5

CppCon 2018: Jason Turner "Surprises in Object Lifetime"



cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

JASON TURNER

**Surprises
in
Object Lifetime**

constexpr All The Things

constexpr doesn't allow undefined behavior. Compiler enforcement varies.

```
1  constexpr int & get_val() {
2      int i{};
3      return i;
4  }
5
6  constexpr int do_thing() {
7      return ++get_val();
8  }
9
10 int main() {
11     constexpr auto val = do_thing();
12     return val;
13 }
```

Copyright Jason Turner @lefticus 17.6

CppCon.org

Arthur O'Dwyer: Feature that Always Works

- ▶ `for (auto&& elt : range)` Always Works
- ▶ `for (auto&& elt : range)` Still Always Works

```
1 | for (auto&& elt : range) {  
2 |     do_something_with(elt);  
3 | }
```



Scott Hanselman
@shanselman

2 Replies, 42 Quotes



If you're starting a sentence with "Why don't you just..." then it's very likely you don't understand the complexity of the problem.

26/09/2017, 18:25 (Yesterday)

Twitter Web Client

850 Likes

403 Retweets

Thread >

Twitter



Ahmed Khalaf @ahmedkhalaf_92

And "[WHYNOTJUST.md](#)" for justifying unintuitive solutions that have intuitive alternatives that don't work/impractical.



Joe Groff @jckarter

Ideas often get copied from well-known established projects into new projects with the background behind those decisions lost. Projects should have a "MISTAKES.md" to lay out regretful design choices that can't be reversed for time, compatibility, etc. reasons

13h • 14/01/2019 • 20:36

13h • 14/01/2019 • 20:46