

Demystifying Value Categories in C++

iCSC 2020

Nis Meinert

Rostock University

Disclaimer

- This talk is mainly about hounding (unnecessary) copy ctors
- In case you don't care:

“If you're not at all interested in performance, shouldn't you be in the Python room down the hall?” (Scott Meyer)

PART I

- Understanding References
- Value Categories
- Perfect Forwarding
- Reading Assembly for Fun and Profit
- Implicit Costs of `const&`

PART II

- Dangling References
- `std::move` in the wild
- What Happens on `return`?
- RVO in Depth
- Perfect Backwarding

PART I

Understanding References

Q: What is the output of the programs?

```
1  #!/usr/bin/env python3
2
3  class S:
4      def __init__(self, x):
5          self.x = x
6
7  def swap(a, b):
8      b, a = a, b
9
10 if __name__ == '__main__':
11     a, b = S(1), S(2)
12     swap(a, b)
13     print(f'{a.x}{b.x}')
```

```
1  #include <iostream>
2
3  struct S {
4      int x;
5  };
6
7  void swap(S& a, S& b) {
8      S& tmp = a;
9      a = b;
10     b = tmp;
11 }
12
13 int main() {
14     S a{1}; S b{2};
15     swap(a, b);
16     std::cout << a.x << b.x;
17 }
```

godbolt.org/z/L6sr2B

Q: What is the output of the programs?

A: 12

```
1 #!/usr/bin/env python3
2
3 class S:
4     def __init__(self, x):
5         self.x = x
6
7     def swap(a, b):
8         b, a = a, b
9
10 if __name__ == '__main__':
11     a, b = S(1), S(2)
12     swap(a, b)
13     print(f'{a.x}{b.x}')
```

A: 22

```
1 #include <iostream>
2
3 struct S {
4     int x;
5 };
6
7 void swap(S& a, S& b) {
8     S& tmp = a;
9     a = b;
10    b = tmp;
11 }
12
13 int main() {
14     S a{1}; S b{2};
15     swap(a, b);
16     std::cout << a.x << b.x;
17 }
```

godbolt.org/z/L6sr2B

Q: What is the output of the program?

```
1 #include <iostream>
2
3 struct S {
4     int x;
5 };
6
7 void swap(S& a, S& b) {
8     S tmp = a;
9     a = b;
10    b = tmp;
11 }
12
13 int main() {
14     S a{1}; S b{2};
15     swap(a, b);
16     std::cout << a.x << b.x;
17 }
```

godbolt.org/z/P6t_eR


```
1 #include <iostream>
2
3 struct S {
4     int x;
5 };
6
7 void swap(S& a, S& b) {
8     S tmp = a;
9     a = b;
10    b = tmp;
11 }
12
13 int main() {
14     S a{1}; S b{2};
15     swap(a, b);
16     std::cout << a.x << b.x;
17 }
```

godbolt.org/z/P6t_eR

Q: What is the output of the program?

```
1  #include <iostream>
2
3  struct S {
4      int x;
5      S(int x): x(x) { std::cout << 'a'; }
6      S(const S& other): x(other.x) { std::cout << 'b'; }
7      S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8  };
9
10 void swap(S& a, S& b) {
11     S& tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2};
18     swap(a, b);
19     std::cout << a.x << b.x;
20 }
```

godbolt.org/z/HV5Y4B

A: aacc22

```
1 #include <iostream>
2
3 struct S {
4     int x;
5     S(int x): x(x) { std::cout << 'a'; }
6     S(const S& other): x(other.x) { std::cout << 'b'; }
7     S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8 };
9
10 void swap(S& a, S& b) {
11     S& tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2};
18     swap(a, b);
19     std::cout << a.x << b.x;
20 }
```

godbolt.org/z/HV5Y4B

Q: What is the output of the program?

```
1 #include <iostream>
2
3 struct S {
4     int x;
5     S(int x): x(x) { std::cout << 'a'; }
6     S(const S& other): x(other.x) { std::cout << 'b'; }
7     S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8 };
9
10 void swap(S& a, S& b) {
11     S tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2};
18     swap(a, b);
19     std::cout << a.x << b.x;
20 }
```

godbolt.org/z/BySvgV

A: aabcc21

```
1 #include <iostream>
2
3 struct S {
4     int x;
5     S(int x): x(x) { std::cout << 'a'; }
6     S(const S& other): x(other.x) { std::cout << 'b'; }
7     S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8 };
9
10 void swap(S& a, S& b) {
11     S tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2};
18     swap(a, b);
19     std::cout << a.x << b.x;
20 }
```

godbolt.org/z/BySvgV

Q: What is the output of the program?

```
1 #include <iostream>
2
3 struct S {
4     int x;
5     S(int x): x(x) { std::cout << 'a'; }
6     S(const S& other): x(other.x) { std::cout << 'b'; }
7     S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8 };
9
10 void swap(S* a, S* b) {
11     S* tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2};
18     swap(&a, &b);
19     std::cout << a.x << b.x;
20 }
```

godbolt.org/z/ynizkH

```
1 #include <iostream>
2
3 struct S {
4     int x;
5     S(int x): x(x) { std::cout << 'a'; }
6     S(const S& other): x(other.x) { std::cout << 'b'; }
7     S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8 };
9
10 void swap(S* a, S* b) {
11     S* tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2};
18     swap(&a, &b);
19     std::cout << a.x << b.x;
20 }
```

godbolt.org/z/ynizkH

Q: What is the output of the program?

```
1  #include <iostream>
2
3  struct S {
4      int x;
5      S(int x): x(x) { std::cout << 'a'; }
6      S(const S& other): x(other.x) { std::cout << 'b'; }
7      S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8  };
9
10 void swap(S* a, S* b) {
11     S* tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2}; S* a_ptr = &a; S* b_ptr = &b;
18     swap(a_ptr, b_ptr);
19     std::cout << a_ptr->x << b_ptr->x;
20 }
```

godbolt.org/z/ssPRn7


```
1 #include <iostream>
2
3 struct S {
4     int x;
5     S(int x): x(x) { std::cout << 'a'; }
6     S(const S& other): x(other.x) { std::cout << 'b'; }
7     S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8 };
9
10 void swap(S* a, S* b) {
11     S* tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2}; S* a_ptr = &a; S* b_ptr = &b;
18     swap(a_ptr, b_ptr);
19     std::cout << a_ptr->x << b_ptr->x;
20 }
```

godbolt.org/z/ssPRn7

Q: What is the output of the program?

```
1 #include <iostream>
2
3 struct S {
4     int x;
5     S(int x): x(x) { std::cout << 'a'; }
6     S(const S& other): x(other.x) { std::cout << 'b'; }
7     S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8 };
9
10 void swap(S*& a, S*& b) {
11     S* tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2}; S* a_ptr = &a; S* b_ptr = &b;
18     swap(a_ptr, b_ptr);
19     std::cout << a_ptr->x << b_ptr->x;
20 }
```

godbolt.org/z/yUnrbf

```
1 #include <iostream>
2
3 struct S {
4     int x;
5     S(int x): x(x) { std::cout << 'a'; }
6     S(const S& other): x(other.x) { std::cout << 'b'; }
7     S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8 };
9
10 void swap(S*& a, S*& b) {
11     S* tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2}; S* a_ptr = &a; S* b_ptr = &b;
18     swap(a_ptr, b_ptr);
19     std::cout << a_ptr->x << b_ptr->x;
20 }
```

godbolt.org/z/yUnrbf

Q: What is the output of the program?

```
1  #include <iostream>
2
3  struct S {
4      int x;
5      S(int x): x(x) { std::cout << 'a'; }
6      S(const S& other): x(other.x) { std::cout << 'b'; }
7      S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8  };
9
10 void swap(S*& a, S*& b) {
11     S* tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2};
18     swap(&a, &b);
19     std::cout << a.x << b.x;
20 }
```

godbolt.org/z/kbRGx7

Q: What is the output of the program?

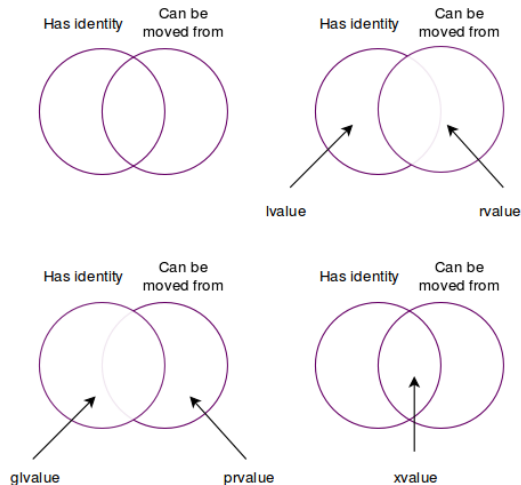
error: cannot bind non-const lvalue reference of type “S&” to an rvalue of type “S”

```
1 #include <iostream>
2
3 struct S {
4     int x;
5     S(int x): x(x) { std::cout << 'a'; }
6     S(const S& other): x(other.x) { std::cout << 'b'; }
7     S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
8 };
9
10 void swap(S*& a, S*& b) {
11     S* tmp = a;
12     a = b;
13     b = tmp;
14 }
15
16 int main() {
17     S a{1}; S b{2};
18     swap(&a, &b);
19     std::cout << a.x << b.x;
20 }
```

godbolt.org/z/kbRGx7

Value Categories

Value categories with Venn diagrams

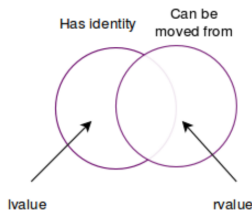


(diagrams shamelessly stolen from bajamircea.github.io/coding/cpp/2016/04/07/move-forward.html)

Value categories with Venn diagrams

(diagrams shamelessly stolen from bajamircea.github.io/coding/cpp/2016/04/07/move-forward.html)

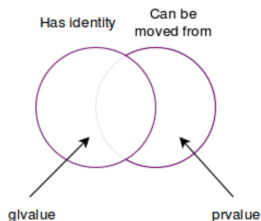
```
1 struct S{ int x; };
2
3 S make_S(int x) {
4     S s{.x = x};
5     return s; // has no name after returning
6 }
7
8 int main() {
9     S a = make_S(42); // `a` is an lvalue
10                     // initialized with a prvalue
11
12     S b = std::move(a); // prepare to die, `a`!
13                       // now `a` became an xvalue
14
15     auto x = a.x; // ERROR: `a` is in an undefined state
16     a = make_S(13);
17     x = a.x; // fine!
18 }
```



Value categories with Venn diagrams

(diagrams shamelessly stolen from bajamircea.github.io/coding/cpp/2016/04/07/move-forward.html)

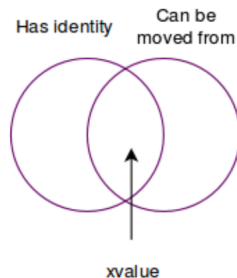
```
1 struct S{ int x; };
2
3 S make_S(int x) {
4     S s{.x = x};
5     return s; // has no name after returning
6 }
7
8 int main() {
9     S a = make_S(42); // 'a' is an lvalue
10                    // initialized with a prvalue
11
12     S b = std::move(a); // prepare to die, 'a'!
13                      // now 'a' became an xvalue
14
15     auto x = a.x; // ERROR: 'a' is in an undefined state
16     a = make_S(13);
17     x = a.x; // fine!
18 }
```



Value categories with Venn diagrams

(diagrams shamelessly stolen from bajamircea.github.io/coding/cpp/2016/04/07/move-forward.html)

```
1 struct S{ int x; };
2
3 S make_S(int x) {
4     S s{.x = x};
5     return s; // has no name after returning
6 }
7
8 int main() {
9     S a = make_S(42); // `a` is an lvalue
10                    // initialized with a prvalue
11
12     S b = std::move(a); // prepare to die, `a`!
13                    // now `a` became an xvalue
14
15     auto x = a.x; // ERROR: `a` is in an undefined state
16     a = make_S(13);
17     x = a.x; // fine!
18 }
```



Binding references to temporaries

error: cannot bind non-const lvalue reference of type “S*&” to an rvalue of type “S”

```
1  template <typename T>
2  void swap(T& a, T&b) { ... }
3
4  int main() {
5      S a{1};
6      S b{2};
7      swap(&a, &b);
8  }
```

- Memory addresses are always rvalues!
- One cannot refer to something that doesn't have a name...
- ...except it is a const reference (lifetime extension)

`std::move`

std::move

```
1  #include <iostream>
2  #include <utility>
3
4  struct S{};
5
6  void f(const S&) { std::cout << 'a'; }
7  void f(S&&) { std::cout << 'b'; }
8
9  int main() {
10     S s;
11     f(s);           // prints 'a'
12     f(std::move(s)); // prints 'b'
13 }
```

godbolt.org/z/tK34H9

→ std::move creates xvalues

→ Syntax:

→ lvalue ref.: S&

→ rvalue ref.: S&&

Q: What is the output of the program?

```
1 #include <iostream>
2 #include <utility>
3
4 struct S{
5     S() { std::cout << 'a'; }
6     S(const S&) { std::cout << 'b'; }
7     S(S&&) { std::cout << 'c'; }
8 };
9
10 int main() {
11     S s1;
12     S s2(s1);
13     S s3(S{});
14     S s4(std::move(s1));
15 }
```

godbolt.org/z/xi4DbB

A: abac

```
1  #include <iostream>
2  #include <utility>
3
4  struct S{
5      S() { std::cout << 'a'; }
6      S(const S&) { std::cout << 'b'; }
7      S(S&&) { std::cout << 'c'; }
8  };
9
10 int main() {
11     S s1;
12     S s2(s1);
13     S s3(S{});
14     S s4(std::move(s1));
15 }
```

godbolt.org/z/xi4DbB

- S s1: no surprise
- S s2(s1): no surprise
- S s3(S{}): *mandatory* copy elision (initializer is prvalue of the same class type)
- S s4(std::move(s1)): forced move construction

Q: What is the output of the program?

```
1 #include <iostream>
2 #include <utility>
3
4 struct S {
5     S() { std::cout << 'a'; }
6     S(const S&) { std::cout << 'b'; }
7     S(S&&) { std::cout << 'c'; }
8 };
9
10 void f(const S&) { std::cout << '1'; }
11 void f(S&) { std::cout << '2'; }
12 void f(S&&) { std::cout << '3'; }
13
14 int main() {
15     S s1;
16     f(s1);
17     f(S{});
18     f(std::move(s1));
19 }
```

godbolt.org/z/yB5zTx

A: a2a33

```
1 #include <iostream>
2 #include <utility>
3
4 struct S {
5     S() { std::cout << 'a'; }
6     S(const S&) { std::cout << 'b'; }
7     S(S&&) { std::cout << 'c'; }
8 };
9
10 void f(const S&) { std::cout << '1'; }
11 void f(S&) { std::cout << '2'; }
12 void f(S&&) { std::cout << '3'; }
13
14 int main() {
15     S s1;
16     f(s1);
17     f(S{});
18     f(std::move(s1));
19 }
```

godbolt.org/z/yB5zTx

Q: What is the output of the program?

```
1 #include <iostream>
2 #include <utility>
3
4 struct S {
5     S() { std::cout << 'a'; }
6     S(const S&) { std::cout << 'b'; }
7     S(S&&) { std::cout << 'c'; }
8 };
9
10 void f(const S&) { std::cout << '1'; }
11 void f(S) { std::cout << '2'; }
12 void f(S&&) { std::cout << '3'; }
13
14 int main() {
15     S s1;
16     f(s1);
17     f(S{});
18     f(std::move(s1));
19 }
```

godbolt.org/z/MN9mQM

Q: What is the output of the program?

```
1  #include <iostream>
2  #include <utility>
3
4  struct S {
5      S() { std::cout << 'a'; }
6      S(const S&) { std::cout << 'b'; }
7      S(S&&) { std::cout << 'c'; }
8  };
9
10 void f(const S&) { std::cout << '1'; }
11 void f(S) { std::cout << '2'; }
12 void f(S&&) { std::cout << '3'; }
13
14 int main() {
15     S s1;
16     f(s1);
17     f(S{});
18     f(std::move(s1));
19 }
```

godbolt.org/z/MN9mQM

Compile-time error (in all three cases)

→ `f(s1)`: ambiguity between 2 and 1

→ `f(S{})`: ambiguity between 2 and 3

→ `f(std::move(s1))`: same as `f(S)`

↪ compiler cannot differentiate between copy and reference overloads! (neither lvalue, nor rvalue)

Q: What is the output of the program?

```
1  #include <iostream>
2  #include <utility>
3
4  struct S {
5      ~S() { std::cout << 'a'; }
6  };
7
8  void f(const S&) { std::cout << '1'; }
9  void f(S&) { std::cout << '2'; }
10 void f(S&&) { std::cout << '3'; }
11
12 int main() {
13     S&& r1 = S{};
14     f(r1);
15
16     S&& r2 = S{};
17     f(std::move(r2));
18 }
```

godbolt.org/z/ZsfzmD

```

1  #include <iostream>
2  #include <utility>
3
4  struct S {
5      ~S() { std::cout << 'a'; }
6  };
7
8  void f(const S&) { std::cout << '1'; }
9  void f(S&) { std::cout << '2'; }
10 void f(S&&) { std::cout << '3'; }
11
12 int main() {
13     S&& r1 = S{};
14     f(r1);
15
16     S&& r2 = S{};
17     f(std::move(r2));
18 }

```

godbolt.org/z/ZsfzmD

- S&&: object that nobody cares about anymore and which will die soon (cf. lifetime extension!)
- std::move does not actually kill, but makes the object look like a dying object



An rvalue has no name

NB: an rvalue ref behaves like an lvalue ref except that it can bind to a temporary (an rvalue), whereas one cannot bind a (non const) lvalue ref to an rvalue.

std::move

```
1 #include <type_traits>
2
3 template <typename T>
4 decltype(auto) move(T&& t) {
5     using R = std::remove_reference_t<T>&&;
6     return static_cast<R>(t);
7 }
```

godbolt.org/z/yp8Rpc

So what does std::move?

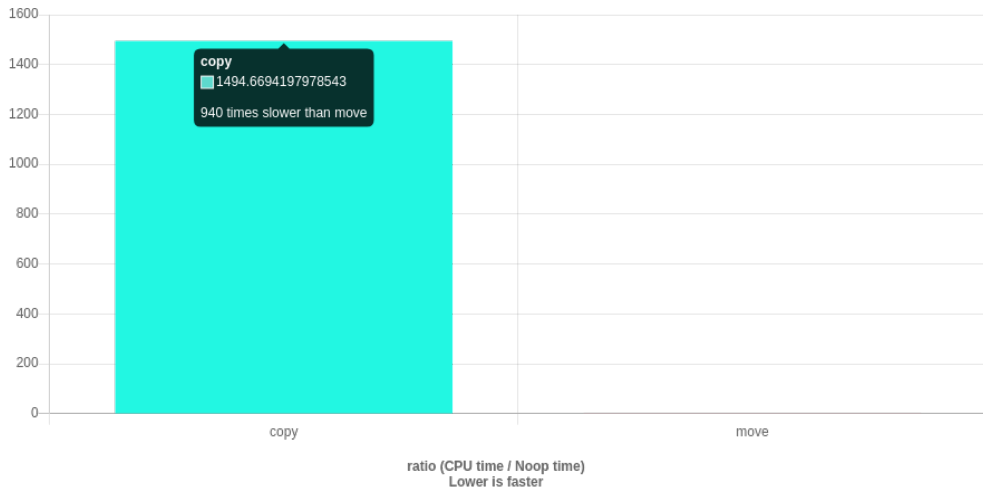
- does not *move*
- does not destroy
- does nothing at all during runtime
- **unconditionally casts** its argument to an rvalue

Quick Bench: tinyurl.com/y67sg7to

```
1 std::vector<int> x(1000, 42);
2 std::vector<int> y(1000, 42);
3 for (auto _ : state) {
4     auto tmp = x;
5     x = y;
6     y = tmp;
7     benchmark::DoNotOptimize(x[345] + y[678]);
8 }
```

```
1 std::vector<int> x(1000, 42);
2 std::vector<int> y(1000, 42);
3 for (auto _ : state) {
4     auto tmp = std::move(x);
5     x = std::move(y);
6     y = std::move(tmp);
7     benchmark::DoNotOptimize(x[345] + y[678]);
8 }
```

Quick Bench: tinyurl.com/y67sg7to



Universal References

Rvalue ref. or no rvalue ref.?

Rvalue refs are declared using “&&”: reasonable to assume that the presence of “&&” in a type declaration indicates an rvalue reference?

```
struct S{};

S&& s = S{};           // (1)

auto&& s2 = s;          // (2)

void f(S&& s);          // (3)

template <typename T>
void f(T&& t);          // (4)

template <typename T>
void f(const T&& t);     // (5)

template <typename T>
void f(std::vector<T&& v); // (6)
```

Does “&&” mean rvalue reference?

→ (1): ???

→ (2): ???

→ (3): ???

→ (4): ???

→ (5): ???

→ (6): ???

Rvalue ref. or no rvalue ref.?

Rvalue refs are declared using “&&”: reasonable to assume that the presence of “&&” in a type declaration indicates an rvalue reference? **No!**

```
struct S{};

S&& s = S{}; // (1)

auto&& s2 = s; // (2)

void f(S&& s); // (3)

template <typename T>
void f(T&& t); // (4)

template <typename T>
void f(const T&& t); // (5)

template <typename T>
void f(std::vector<T&& v); // (6)
```

Does “&&” mean rvalue reference?

- (1): yes
- (2): no
- (3): yes
- (4): no
- (5): yes
- (6): yes

Universal references

Universal references*

- Syntax (x is a universal reference):
 - `auto&& x`
 - `template <typename T>`
`f(T&& x...`
- Rule of thumb: substitute fully qualified type into `auto` or `T` and reduce:
 - $\&\& \mapsto \&\&$
 - $\&\&\& \mapsto \&$
 - $\&\&\&\& \mapsto \&\&$

```
std::vector<S> v;  
auto&& s = v[0]; // S&&& -> S&  
  
auto&& s2 = S{}; // S&&&& -> S&&  
auto&& s3 = s2; // S&&& -> S&  
  
// S&&&& -> S&&  
auto&& s3 = std::move(s2);
```

* *Universal reference*: term introduced by Scott Meyers

Q: What is the output of the program?

```
1  #include <iostream>
2  #include <type_traits>
3
4  struct S {
5      S() { std::cout << 'a'; }
6      S(const S&) { std::cout << 'b'; }
7      S(S&&) { std::cout << 'c'; }
8  };
9
10 template <typename T>
11 S f(T&& t) { return t; }
12
13 int main() {
14     S s{};
15     f(s);
16     f(std::move(s));
17 }
```

godbolt.org/z/rYE5ny

A: abb

```
1 #include <iostream>
2 #include <type_traits>
3
4 struct S {
5     S() { std::cout << 'a'; }
6     S(const S&) { std::cout << 'b'; }
7     S(S&&) { std::cout << 'c'; }
8 };
9
10 template <typename T>
11 S f(T&& t) { return t; }
12
13 int main() {
14     S s{};
15     f(s);
16     f(std::move(s));
17 }
```

godbolt.org/z/rYE5ny

...how to preserve the value category?

Q: What is the output of the program?

```
1 #include <iostream>
2 #include <type_traits>
3
4 struct S{};
5
6 void f(S&) { std::cout << 'a'; }
7 void f(S&&) { std::cout << 'b'; }
8
9 int main() {
10     auto&& r1 = S{};
11     static_assert(std::is_same_v<decltype(r1), S&&>);
12     f(r1);
13     f(static_cast<S&&>(r1));
14
15     S s;
16     auto&& r2 = s;
17     static_assert(std::is_same_v<decltype(r2), S&>);
18     f(r2);
19 }
```

godbolt.org/z/Lat-6T

A: aba

```
1 #include <iostream>
2 #include <type_traits>
3
4 struct S{};
5
6 void f(S&) { std::cout << 'a'; }
7 void f(S&&) { std::cout << 'b'; }
8
9 int main() {
10     auto&& r1 = S{};
11     static_assert(std::is_same_v<decltype(r1), S&&>);
12     f(r1);
13     f(static_cast<S&&>(r1));
14
15     S s;
16     auto&& r2 = s;
17     static_assert(std::is_same_v<decltype(r2), S&>);
18     f(r2);
19 }
```

godbolt.org/z/Lat-6T

Q: What is the output of the program?

```
1  #include <iostream>
2
3  struct S{
4      void f() & { std::cout << 'a'; }
5      void f() && { std::cout << 'b'; }
6  };
7
8  int main() {
9      auto&& r1 = S{};
10     r1.f();
11     static_cast<decltype(r1)>(r1).f();
12
13     auto&& r2 = r1;
14     r2.f();
15     static_cast<decltype(r2)>(r2).f();
16 }
```

godbolt.org/z/ry4Ybx

```
1 #include <iostream>
2
3 struct S{
4     void f() & { std::cout << 'a'; }
5     void f() && { std::cout << 'b'; }
6 };
7
8 int main() {
9     auto&& r1 = S{};
10    r1.f();
11    static_cast<decltype(r1)>(r1).f();
12
13    auto&& r2 = r1;
14    r2.f();
15    static_cast<decltype(r2)>(r2).f();
16 }
```

godbolt.org/z/ry4Ybx

How do we fuse these implementations?

```
1 // if `t` is an lvalue of type `S`  
2 S& forward(S& t) {  
3     return t;  
4 }  
5  
6 // if `t` is an rvalue of type `S`  
7 S&& forward(S& t) { // not `S&&!`  
8     return std::move(t); // static_cast<S&&>(t)  
9 }
```

```
1 #include <type_traits>  
2  
3 template <typename T>  
4 T&& forward(std::remove_reference_t<T>& t) {  
5     return static_cast<T&&>(t);  
6 }
```

godbolt.org/z/k997Zh

Q: What is the output of the program?

```
1  #include <iostream>
2  #include <type_traits>
3
4  struct S {
5      S() { std::cout << 'a'; }
6      S(const S&) { std::cout << 'b'; }
7      S(S&&) { std::cout << 'c'; }
8  };
9
10 template <typename T>
11 S f(T&& t) { return std::forward<T>(t); }
12
13 int main() {
14     S s{};
15     f(s);
16     f(std::move(s));
17 }
```

godbolt.org/z/WvVBwL

```
1 #include <iostream>
2 #include <type_traits>
3
4 struct S {
5     S() { std::cout << 'a'; }
6     S(const S&) { std::cout << 'b'; }
7     S(S&&) { std::cout << 'c'; }
8 };
9
10 template <typename T>
11 S f(T&& t) { return std::forward<T>(t); }
12
13 int main() {
14     S s{};
15     f(s);
16     f(std::move(s));
17 }
```

godbolt.org/z/WvVBwL

Rule of thumb: Use `std::move` for rvalues and `std::forward` for universal references

Q: Why can't we use perfect forwarding here?

```
1 #include <functional>
2
3 template <typename Iter, typename Callable, typename... Args>
4 void foreach (Iter current, Iter end, Callable op, const Args&... args) {
5     while (current != end) {
6         std::invoke(op, args..., *current);
7         ++current;
8     }
9 }
```

godbolt.org/z/PZ-tqV

Q: Why can't we use perfect forwarding here?

```
1 #include <functional>
2
3 template <typename Iter, typename Callable, typename... Args>
4 void foreach (Iter current, Iter end, Callable op, const Args&... args) {
5     while (current != end) {
6         std::invoke(op, args..., *current);
7         ++current;
8     }
9 }
```

godbolt.org/z/PZ-tqV

A: The first call in the loop might *steal* the values, leading to unexpected behavior calling `op` in subsequent iterations.

Reading x86-64 Assembly

...for fun and profit

Function Prologue & Epilogue

- Few lines of code at the beginning (*prologue*) and end (*epilogue*) of a function, which **prepare**
 - the **stack** and
 - **registers**
- Not part of assembly: **convention** (defined & interpreted differently by different OS and compilers)

Prologue

```
1 push rbp
2 mov rbp, rsp
3 sub rsp, N
```

alternatively

```
1 enter N, 0
```

(reserve N bytes on stack for local use)

Epilogue

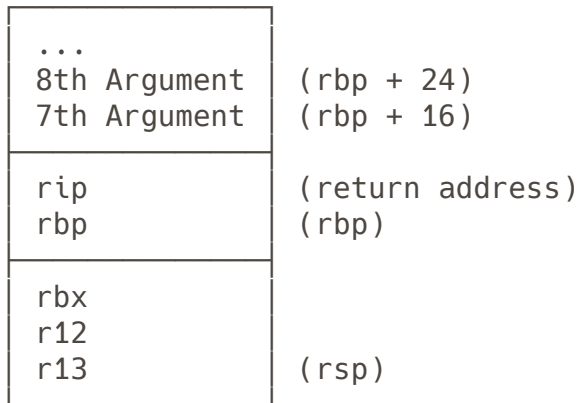
```
1 mov rsp, rbp
2 pop rbp
3 ret
```

alternatively

```
1 leave
2 ret
```

Stack frame for function call

- **CALL** = **PUSH** *address of next instruction* + **JMP** *target*
- **RET** pops return address and transfers control there
- pass arguments 1 ...6 in registers (*rsi*, *rdx*, ...)



(stack frame for function call with 8 arguments and local registers *rbx*, *r12* and *r13*)

lea vs. mov

- `lea`: load effective address
- puts memory address from `src` into the destination `dest`
- Example: `lea eax, [ebx+8]`
 - put `[ebx+8]` into `eax`
 - value of `eax` after instruction: `0x00403A48`
- ...whereas: `mov eax, [ebx+8]`
 - value of `eax` after instruction: `0x0012C140`

`0x00403A40`
`0x00403A44`
`0x00403A48`
`0x00403A4C`

| Registers |
|------------------|
| EAX = 0x00000000 |
| EBX = 0x00403A40 |

| Memory |
|------------|
| 0x7C81776F |
| 0x7C911000 |
| 0x0012C140 |
| 0x7FFDB000 |

Reading assembly for fun and profit

```
1 int f(int x, int y, int z) {  
2     int sum = x + y + z;  
3     return sum;  
4 }
```

godbolt.org/z/GEVny6

```
# g92 -O0  
| f(int, int, int):  
1| push rbp  
1| mov rbp, rsp  
1| mov DWORD PTR [rbp-20], edi  
1| mov DWORD PTR [rbp-24], esi  
1| mov DWORD PTR [rbp-28], edx  
2| mov edx, DWORD PTR [rbp-20]  
2| mov eax, DWORD PTR [rbp-24]  
2| add edx, eax  
2| mov eax, DWORD PTR [rbp-28]  
2| add eax, edx  
2| mov DWORD PTR [rbp-4], eax  
3| mov eax, DWORD PTR [rbp-4]  
4| pop rbp  
4| ret
```

godbolt.org/z/GEVny6

Reading assembly for fun and profit

```
1 int f(int x, int y, int z) {  
2     int sum = x + y + z;  
3     return sum;  
4 }
```

godbolt.org/z/GEVny6

```
# g92 -01  
| f(int, int, int):  
2|     add edi, esi  
2|     lea eax, [rdi+rdx]  
4|     ret
```

godbolt.org/z/RVLjZE

```
# g92 -00  
| f(int, int, int):  
1|     push rbp  
1|     mov rbp, rsp  
1|     mov DWORD PTR [rbp-20], edi  
1|     mov DWORD PTR [rbp-24], esi  
1|     mov DWORD PTR [rbp-28], edx  
2|     mov edx, DWORD PTR [rbp-20]  
2|     mov eax, DWORD PTR [rbp-24]  
2|     add edx, eax  
2|     mov eax, DWORD PTR [rbp-28]  
2|     add eax, edx  
2|     mov DWORD PTR [rbp-4], eax  
3|     mov eax, DWORD PTR [rbp-4]  
4|     pop rbp  
4|     ret
```

godbolt.org/z/GEVny6

Reading assembly for fun and profit

```
1 int f(int x) {  
2     return x + 1;  
3 }  
4  
5 int g(int x) {  
6     return f(x + 2);  
7 }
```

godbolt.org/z/ox78bF

```
# g92 -O0  
| f(int):  
1| push rbp  
1| mov rbp, rsp  
1| mov DWORD PTR [rbp-4], edi  
2| mov eax, DWORD PTR [rbp-4]  
2| add eax, 1  
3| pop rbp  
3| ret  
| g(int):  
5| push rbp  
5| mov rbp, rsp  
5| sub rsp, 8  
5| mov DWORD PTR [rbp-4], edi  
6| mov eax, DWORD PTR [rbp-4]  
6| add eax, 2  
6| mov edi, eax  
6| call f(int)  
7| leave  
7| ret
```

godbolt.org/z/ox78bF

Reading assembly for fun and profit

```
1  int f(int x) {  
2      return x + 1;  
3  }  
4  
5  int g(int x) {  
6      return f(x + 2);  
7  }
```

godbolt.org/z/ox78bF

```
# g92 -01  
| f(int):  
2|  lea eax, [rdi+1]  
3|  ret  
| g(int):  
2|  lea eax, [rdi+3]  
7|  ret
```

godbolt.org/z/Lp0c8r

```
# g92 -00  
| f(int):  
1|  push rbp  
1|  mov rbp, rsp  
1|  mov DWORD PTR [rbp-4], edi  
2|  mov eax, DWORD PTR [rbp-4]  
2|  add eax, 1  
3|  pop rbp  
3|  ret  
| g(int):  
5|  push rbp  
5|  mov rbp, rsp  
5|  sub rsp, 8  
5|  mov DWORD PTR [rbp-4], edi  
6|  mov eax, DWORD PTR [rbp-4]  
6|  add eax, 2  
6|  mov edi, eax  
6|  call f(int)  
7|  leave  
7|  ret
```

godbolt.org/z/ox78bF

Reading assembly for fun and profit

```
1 void side_effect();  
2  
3 int f(int x) {  
4     auto a = x;  
5     side_effect();  
6     return a - x;  
7 }
```

godbolt.org/z/HvigZu

```
# g92 -O0  
| f(int):  
3| push rbp  
3| mov rbp, rsp  
3| sub rsp, 32  
3| mov DWORD PTR [rbp-20], edi  
4| mov eax, DWORD PTR [rbp-20]  
4| mov DWORD PTR [rbp-4], eax  
5| call side_effect()  
6| mov eax, DWORD PTR [rbp-4]  
6| sub eax, DWORD PTR [rbp-20]  
7| leave  
7| ret
```

godbolt.org/z/HvigZu

Implicit Costs of using const&

```
1 void side_effect();  
2  
3 int f(int x) {  
4     auto a = x;  
5     side_effect();  
6     return a - x;  
7 }
```

godbolt.org/z/HvigZu

```
1 void side_effect();  
2  
3 int f(const int& x) {  
4     auto a = x;  
5     side_effect();  
6     return a - x;  
7 }
```

godbolt.org/z/m26mJX

Implicit Costs of using const&

```
# g92 -O0
| f(int):
3|   push rbp
3|   mov rbp, rsp
3|   sub rsp, 32
3|   mov DWORD PTR [rbp-20], edi
4|   mov eax, DWORD PTR [rbp-20]
4|   mov DWORD PTR [rbp-4], eax
5|   call side_effect()
6|   mov eax, DWORD PTR [rbp-4]
6|   sub eax, DWORD PTR [rbp-20]
7|   leave
7|   ret
```

godbolt.org/z/HvigZu

```
# g92 -O0
| f(int const&):
3|   push rbp
3|   mov rbp, rsp
3|   sub rsp, 32
3|   mov QWORD PTR [rbp-24], rdi
4|   mov rax, QWORD PTR [rbp-24]
4|   mov eax, DWORD PTR [rax]
4|   mov DWORD PTR [rbp-4], eax
5|   call side_effect()
6|   mov rax, QWORD PTR [rbp-24]
6|   mov eax, DWORD PTR [rax]
6|   mov edx, DWORD PTR [rbp-4]
6|   sub edx, eax
6|   mov eax, edx
7|   leave
7|   ret
```

godbolt.org/z/m26mJX

Implicit Costs of using const&

```
# g92 -O3
| f(int):
3|   sub rsp, 8
5|   call side_effect()
7|   xor eax, eax
7|   add rsp, 8
7|   ret
```

godbolt.org/z/Ntkj2y

```
# g92 -O3
| f(int const&):
3|   push rbp
3|   push rbx
3|   mov rbx, rdi
3|   sub rsp, 8
4|   mov ebp, DWORD PTR [rdi]
5|   call side_effect()
6|   mov eax, ebp
6|   sub eax, DWORD PTR [rbx]
7|   add rsp, 8
7|   pop rbx
7|   pop rbp
7|   ret
```

godbolt.org/z/gJqDbA

Implicit Costs of using const&

```
# g92 -O3
| f(int):
3|   sub rsp, 8
5|   call side_effect()
7|   xor eax, eax
7|   add rsp, 8
7|   ret
```

godbolt.org/z/Ntkj2y

NB #1: adjusting `rsp` in function prologue necessary when function is not a leaf function since callee have to know where to start saving variables on stack. (Adjusting `rsp` can be omitted in leaf functions.)

```
# g92 -O3
| f(int const&):
3|   push rbp
3|   push rbx
3|   mov rbx, rdi
3|   sub rsp, 8
4|   mov ebp, DWORD PTR [rdi]
5|   call side_effect()
6|   mov eax, ebp
6|   sub eax, DWORD PTR [rbx]
7|   add rsp, 8
7|   pop rbx
7|   pop rbp
7|   ret
```

godbolt.org/z/gJqDbA

Implicit Costs of using const&

```
# g92 -O3
| f(int):
3|   sub rsp, 8
5|   call side_effect()
7|   xor eax, eax
7|   add rsp, 8
7|   ret
```

godbolt.org/z/Ntkj2y

NB #2: Offset x in `sub rsp, x`, x is objective of optimizations such as alignment: ABI requires stack to be aligned to 16 bytes.

```
# g92 -O3
| f(int const&):
3|   push rbp
3|   push rbx
3|   mov rbx, rdi
3|   sub rsp, 8
4|   mov ebp, DWORD PTR [rdi]
5|   call side_effect()
6|   mov eax, ebp
6|   sub eax, DWORD PTR [rbx]
7|   add rsp, 8
7|   pop rbx
7|   pop rbp
7|   ret
```

godbolt.org/z/gJqDbA

Will it compile?

```
1 #include <array>
2 #include <span>
3
4 int main() {
5     constexpr std::array x{
6         4, 8, 15, 16, 23, 42
7     };
8     constexpr std::span x_view{x};
9 }
```

godbolt.org/z/3CIQW_

```
1 template <std::size_t N>
2 constexpr span(const std::array<value_type, N>& arr) noexcept;
```

cppreference.com/w/cpp/container/span/span

Will it compile?

```
1 #include <array>
2 #include <span>
3
4 int main() {
5     constexpr std::array x{
6         4, 8, 15, 16, 23, 42
7     };
8     constexpr std::span x_view{x};
9 }
```

godbolt.org/z/3CIQW_

No!

- Constructor takes by reference
- References to automatic storage objects are not constant expressions!
- Solutions?

```
1 template <std::size_t N>
2 constexpr span(const std::array<value_type, N>& arr) noexcept;
```

cppreference.com/w/cpp/container/span/span

Will it compile?

```
1 #include <array>
2 #include <span>
3
4 int main() {
5     constexpr static std::array x{
6         4, 8, 15, 16, 23, 42
7     };
8     constexpr std::span x_view{x};
9 }
```

godbolt.org/z/MXgBx4

this will work though, since reference / pointer does not *escape* constant expression ...

```
1 #include <array>
2 #include <span>
3
4 constexpr auto f() {
5     std::array x{4, 8, 15, 16, 23, 42};
6     std::span x_view{x};
7     return 0;
8 }
9
10 int main() {
11     static_assert(f() == 0);
12 }
```

godbolt.org/z/CHeY7r

PART II

Dangling References

Will it compile?

```
1 struct S {  
2     int x;  
3 };  
4  
5 auto f() {  
6     S s{.x = 42};  
7     return s;  
8 }  
9  
10 int main() {  
11     S& s = f();  
12     return s.x;  
13 }
```

godbolt.org/z/pEgJ6W

Will it compile?

```
1 struct S {  
2     int x;  
3 };  
4  
5 auto f() {  
6     S s{.x = 42};  
7     return s;  
8 }  
9  
10 int main() {  
11     S& s = f();  
12     return s.x;  
13 }
```

godbolt.org/z/pEgJ6W

error: cannot bind non-const lvalue reference of type “S&” to an rvalue of type “S”

Will it invoke undefined behavior?

```
1 struct S {  
2     int x;  
3 };  
4  
5 auto f() {  
6     S s{.x = 42};  
7     return s;  
8 }  
9  
10 int main() {  
11     const S& s = f();  
12     return s.x;  
13 }
```

godbolt.org/z/85aznD

...binding a reference to a temporary???

Temporary object lifetime extension

```
1 struct S {  
2     int x;  
3 };  
4  
5 auto f() {  
6     S s{.x = 42};  
7     return s;  
8 }  
9  
10 int main() {  
11     const S& s = f();  
12     return s.x;  
13 }
```

godbolt.org/z/85aznD

cppreference.com: “The lifetime of a temporary object may be extended by binding to a const lvalue reference or to an rvalue reference (since C++11).”

Q: What is the output of the program?

```
1  #include <iostream>
2
3  template <char id> struct Log {
4      Log() { std::cout << id << 1; }
5      virtual ~Log() { std::cout << id << 2; }
6  };
7  struct A: Log<'a'> {
8      int x;
9      A(int x): x(x) {};
10 };
11 struct B: Log<'b'> {
12     const A& a;
13     B(const A& a): a(a) {}
14 };
15
16 int main() {
17     const B& b = B{A{42}};
18     std::cout << 'x';
19     return b.a.x;
20 }
```

godbolt.org/z/_iGynP

A: a1b1a2xb2

```
1 #include <iostream>
2
3 template <char id> struct Log {
4     Log() { std::cout << id << 1; }
5     virtual ~Log() { std::cout << id << 2; }
6 };
7 struct A: Log<'a'> {
8     int x;
9     A(int x): x(x) {};
10 };
11 struct B: Log<'b'> {
12     const A& a;
13     B(const A& a): a(a) {}
14 };
15
16 int main() {
17     const B& b = B{A{42}};
18     std::cout << 'x';
19     return b.a.x;
20 }
```

godbolt.org/z/_iGynP

Dangling reference!!!

- lifetime extension only for result of the temporary expression, **not any sub-expression**
- use address sanitizer!

contrived?

Reference lifetime extension

(derived from `abseil.io`: Tip of the Week #107: “Reference Lifetime Extension”)

```
1 std::vector<std::string_view> explode(const std::string& s);  
2  
3 for (std::string_view s: explode(str_cat("oo", "ps"))) { // WRONG!  
4     [...]
```

Q: What is the output of the program?

```
1 #include <vector>
2
3 int main() {
4     std::vector<int> v;
5     v.push_back(1);
6     auto& x = v[0];
7     v.push_back(2);
8     return x;
9 }
```

godbolt.org/z/vx9cTq

Q: What is the output of the program?

```
1 #include <vector>
2
3 int main() {
4     std::vector<int> v;
5     v.push_back(1);
6     auto& x = v[0];
7     v.push_back(2);
8     return x;
9 }
```

godbolt.org/z/vx9cTq

Dangling reference!!!

- `std::vector` needs to reallocate all the space the second time an element is pushed
- use address sanitizer!

std::move in the wild

Moving `std::string`

(derived from CppCon 2019: *Ben Deane* “Everyday Efficiency: In-Place Construction (Back to Basics?)”)

```
1 static void cp_small_str(benchmark::State& state) {  
2     for (auto _ : state) {  
3         std::string original("small");  
4         benchmark::DoNotOptimize(original);  
5         std::string copied = original;  
6         benchmark::DoNotOptimize(copied);  
7     }  
8 }  
9 BENCHMARK(cp_small_str);
```

```
1 static void mv_small_str(benchmark::State& state) {  
2     for (auto _ : state) {  
3         std::string original("small");  
4         benchmark::DoNotOptimize(original);  
5         std::string moved = std::move(original);  
6         benchmark::DoNotOptimize(moved);  
7     }  
8 }  
9 BENCHMARK(mv_small_str);
```

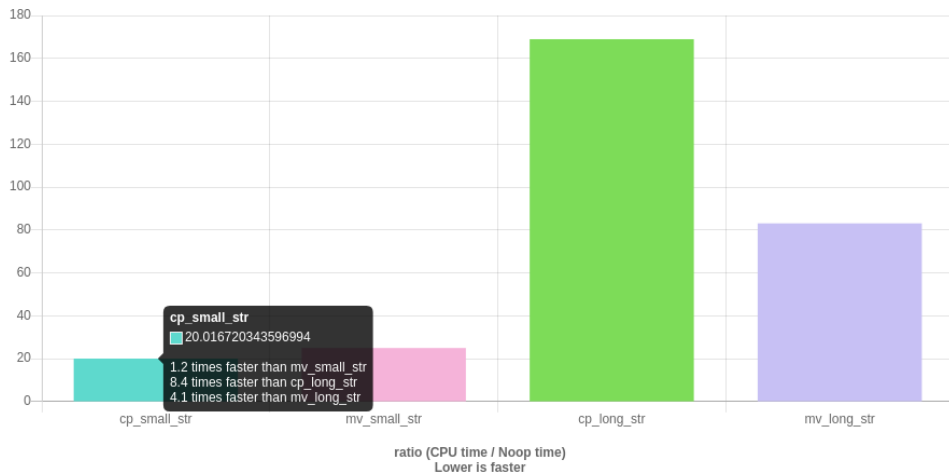
Moving `std::string`

(derived from CppCon 2019: *Ben Deane* “Everyday Efficiency: In-Place Construction (Back to Basics?)”)

```
1 static void cp_long_str(benchmark::State& state) {  
2     for (auto _ : state) {  
3         std::string original("this is too long for short string optimization");  
4         benchmark::DoNotOptimize(original);  
5         std::string copied = original;  
6         benchmark::DoNotOptimize(copied);  
7     }  
8 }  
9 BENCHMARK(cp_long_str);
```

```
1 static void mv_long_str(benchmark::State& state) {  
2     for (auto _ : state) {  
3         std::string original("this is too long for short string optimization");  
4         benchmark::DoNotOptimize(original);  
5         std::string moved = std::move(original);  
6         benchmark::DoNotOptimize(moved);  
7     }  
8 }  
9 BENCHMARK(mv_long_str);
```


Quick Bench result



Quick Bench: tinyurl.com/yybmdngv

Moving `std::string`

Copy small `std::string`

1. copy stack allocated data

Move small `std::string`

1. copy stack allocated data
2. set string length of moved string to zero

↪ moving is not necessarily better than copying!

Did they forget to mark the move ctor noexcept?

```
// since C++11  
std::map(const std::map&&)  
  
// until C++17  
std::map& operator(std::map&&)  
  
// since C++17  
std::map& operator(std::map&&) noexcept
```

Moving `std::map`

Did they forget to mark the move ctor `noexcept`? **No!**

```
// since C++11
std::map(const std::map&&)

// until C++17
std::map& operator=(std::map&&)

// since C++17
std::map& operator=(std::map&&) noexcept
```

- Move ctor needs to allocate new sentinel node, because moved from container must still be a valid container (albeit in an unspecified state)
- Move assignment can swap, thus no need to allocate

↪ move ctor of `std::map` allocates heap space!

(Billy O'Neal: twitter.com/MalwareMinigun/status/1165310509022736384)

Moving `std::map`

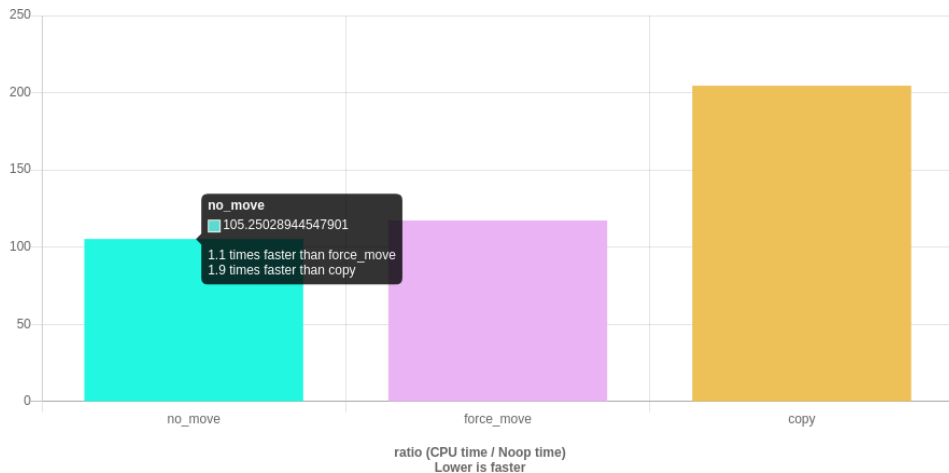
```
1 static void rvo(benchmark::State& state) {  
2     for (auto _ : state) {  
3         auto m = []() -> std::map<int, int> {  
4             std::map<int, int> m{{0, 42}};  
5             return m;  
6         }();  
7         benchmark::DoNotOptimize(m);  
8     }  
9 }  
10 BENCHMARK(rvo);
```

```
1 static void fmove(benchmark::State& state) {  
2     for (auto _ : state) {  
3         auto m = []() -> std::map<int, int> {  
4             std::map<int, int> m{{0, 42}};  
5             return std::move(m);  
6         }();  
7         benchmark::DoNotOptimize(m);  
8     }  
9 }  
10 BENCHMARK(fmove);
```

Moving `std::map`

```
1 static void copy(benchmark::State& state) {  
2     for (auto _ : state) {  
3         std::map<int, int> m{{0, 42}};  
4         benchmark::DoNotOptimize(m);  
5         auto m2 = m;  
6         benchmark::DoNotOptimize(m2);  
7     }  
8 }  
9 BENCHMARK(copy);
```

Quick Bench result



Quick Bench: tinyurl.com/y57egvjp

Why?

Does this code bother anyone?

```
1 #include <cstddef>
2 #include <type_traits>
3 #include <utility>
4
5 template <typename T>
6 struct Data final {
7     T *data;
8     explicit Data(const std::size_t size): data(new T[size]) {}
9     ~Data() { delete [] data; }
10 };
11
12 auto init() {
13     Data<int> d(3);
14     d.data[0] = 1; d.data[1] = 2; d.data[2] = 3;
15     return d;
16 }
17
18 int main() { return init().data[2]; }
```

godbolt.org/z/NR2t9D

Interlude

What happens on return?

Will it compile?

```
1 struct A {  
2     int x;  
3     A(int x, int y = 0) : x(x + y) {}  
4 };  
5  
6 struct B {  
7     int x;  
8     explicit B(int x, int y = 0) : x(x + y) {}  
9 };  
10  
11 template <typename T>  
12 T init() { return 42; }  
13  
14 int main() {  
15     auto x = init<A>().x;  
16     auto y = init<B>().x;  
17 }
```

godbolt.org/z/yfxxRN

Implicit conversion to the function return type

```
1 struct A {  
2     int x;  
3     A(int x, int y = 0) : x(x + y) {}  
4 };  
5  
6 struct B {  
7     int x;  
8     explicit B(int x, int y = 0) : x(x + y) {}  
9 };  
10  
11 template <typename T>  
12 T init() { return 42; }  
13  
14 int main() {  
15     auto x = init<A>().x;  
16     auto y = init<B>().x;  
17 }
```

godbolt.org/z/yfxxRN

error: could not convert “42” from “int” to “B”

Implicit conversion to the function return type

Implicit conversion

- ...if ctor is **not** marked explicit
- Examples:
 - `std::optional(T&&)`
 - `std::string(const char*)`

```
1  #include <optional>
2  #include <string>
3
4  std::optional<int> f() {
5      return 42;
6  }
7
8  std::string g() {
9      return "foo";
10 }
```

godbolt.org/z/Q7h_RW

Q: What is the output of the program?

Compiler flags: `-std=c++14 -fno-elide-constructors`

```
1 #include <iostream>
2
3 struct S {
4     S() { std::cout << 'a'; }
5     S(const S&) { std::cout << 'b'; }
6     S(const S&&) { std::cout << 'c'; }
7     S &operator=(const S&) { std::cout << 'd'; return *this; }
8     S &operator=(const S&&) { std::cout << 'e'; return *this; }
9 };
10
11 S f() { return S{}; }
12
13 int main() {
14     S s(S{}); std::cout << ", ";
15     auto s1 = f(); std::cout << ", ";
16     auto s2{f()};
17 }
```

godbolt.org/z/uoUypK

Compiler flags: `-std=c++14 -fno-elide-constructors`

```
1 #include <iostream>
2
3 struct S {
4     S() { std::cout << 'a'; }
5     S(const S&) { std::cout << 'b'; }
6     S(const S&&) { std::cout << 'c'; }
7     S &operator=(const S&) { std::cout << 'd'; return *this; }
8     S &operator=(const S&&) { std::cout << 'e'; return *this; }
9 };
10
11 S f() { return S{}; }
12
13 int main() {
14     S s(S{}); std::cout << ", ";
15     auto s1 = f(); std::cout << ", ";
16     auto s2{f()};
17 }
```

godbolt.org/z/uoUypK

Q: What is the output of the program?

Compiler flags: `-std=c++17`

```
1  #include <iostream>
2
3  struct S {
4      S() { std::cout << 'a'; }
5      S(const S&) { std::cout << 'b'; }
6      S(const S&&) { std::cout << 'c'; }
7      S &operator=(const S&) { std::cout << 'd'; return *this; }
8      S &operator=(const S&&) { std::cout << 'e'; return *this; }
9  };
10
11 S f() { return S{}; }
12
13 int main() {
14     S s(S{}); std::cout << ", ";
15     auto s1 = f(); std::cout << ", ";
16     auto s2{f()};
17 }
```

godbolt.org/z/ycZulQ

A: a, a, a

Compiler flags: -std=c++17

```
1  #include <iostream>
2
3  struct S {
4      S() { std::cout << 'a'; }
5      S(const S&) { std::cout << 'b'; }
6      S(const S&&) { std::cout << 'c'; }
7      S &operator=(const S&) { std::cout << 'd'; return *this; }
8      S &operator=(const S&&) { std::cout << 'e'; return *this; }
9  };
10
11 S f() { return S{ }; }
12
13 int main() {
14     S s(S{ }); std::cout << ", ";
15     auto s1 = f(); std::cout << ", ";
16     auto s2{f( )};
17 }
```

godbolt.org/z/ycZulQ

Why?

RVO!

Ben Deane: “Perhaps the most important optimization the compiler does”

Copy Elision

```
1 struct S {  
2     S() = default;  
3     S(const S &) = delete;  
4     S &operator=(const S &) = delete;  
5 };  
6  
7 S f() { return S{}; }  
8  
9 int main() {  
10     S s(S{});  
11     auto s1 = f();  
12     auto s2{f()};  
13 }
```

godbolt.org/z/jcWUdd

Mandatory elision of copy/move operations since C++17):

- Return statement: when operand is a prvalue of same class type as return type
- Initialization of a variable: when initializer expression is a prvalue of same class type as the variable type

...even if the copy/move constructor and the destructor has observable side-effects!

Rule of thumb: avoid naming return values

RVO in Depth

C++ Objects in Assembly

```
1 struct S final {  
2     int a, b, c;  
3  
4     S(int a, int b, int c) noexcept:  
5         a(a), b(b), c(c) {}  
6  
7     ~S() noexcept {}  
8  
9     auto sum() noexcept {  
10         return a + b + c;  
11     }  
12 };  
13  
14 int main() {  
15     S s(1, 2, 3);  
16     return s.sum();  
17 }
```

godbolt.org/z/x-qj8_

```
| main:  
14| push rbp  
14| mov rbp, rsp  
14| sub rsp, 16  
14| mov dword ptr [rbp - 4], 0  
15| lea rdi, [rbp - 16]  
15| mov esi, 1  
15| mov edx, 2  
15| mov ecx, 3  
15| call S::S(int, int, int)  
16| lea rdi, [rbp - 16]  
16| call S::sum()  
16| mov dword ptr [rbp - 4], eax  
17| lea rdi, [rbp - 16]  
17| call S::~~S()  
17| mov eax, dword ptr [rbp - 4]  
17| add rsp, 16  
17| pop rbp  
17| ret
```

C++ Objects in Assembly

```
1 struct S final {  
2     int a, b, c;  
3  
4     S(int a, int b, int c) noexcept:  
5         a(a), b(b), c(c) {}  
6  
7     ~S() noexcept {}  
8  
9     auto sum() noexcept {  
10         return a + b + c;  
11     }  
12 };  
13  
14 int main() {  
15     S s(1, 2, 3);  
16     return s.sum();  
17 }
```

godbolt.org/z/x-qj8_

```
| S::S(int, int, int):  
5|     push rbp  
5|     mov rbp, rsp  
5|     mov qword ptr [rbp - 8], rdi  
5|     mov dword ptr [rbp - 12], esi  
5|     mov dword ptr [rbp - 16], edx  
5|     mov dword ptr [rbp - 20], ecx  
5|     mov rax, qword ptr [rbp - 8]  
5|     mov ecx, dword ptr [rbp - 12]  
5|     mov dword ptr [rax], ecx  
5|     mov ecx, dword ptr [rbp - 16]  
5|     mov dword ptr [rax + 4], ecx  
5|     mov ecx, dword ptr [rbp - 20]  
5|     mov dword ptr [rax + 8], ecx  
5|     pop rbp  
5|     ret
```

C++ Objects in Assembly

```
1 struct S final {  
2     int a, b, c;  
3  
4     S(int a, int b, int c) noexcept:  
5         a(a), b(b), c(c) {}  
6  
7     ~S() noexcept {}  
8  
9     auto sum() noexcept {  
10         return a + b + c;  
11     }  
12 };  
13  
14 int main() {  
15     S s(1, 2, 3);  
16     return s.sum();  
17 }
```

godbolt.org/z/x-qj8_

```
| S::sum():  
9| push rbp  
9| mov rbp, rsp  
9| mov qword ptr [rbp - 8], rdi  
9| mov rax, qword ptr [rbp - 8]  
10| mov ecx, dword ptr [rax]  
10| add ecx, dword ptr [rax + 4]  
10| add ecx, dword ptr [rax + 8]  
10| mov eax, ecx  
10| pop rbp  
10| ret
```


RVO in Assembly

```
1 struct S final {  
2     int x;  
3     explicit S(int x) noexcept;  
4     S(const S&) noexcept;  
5     S(S&&) noexcept;  
6     ~S() noexcept;  
7 };  
8  
9 S f() {  
10     return S{42};  
11 }  
12  
13 auto g() {  
14     auto s = f();  
15     return s.x;  
16 }
```

godbolt.org/z/BMgWSl

```
# g92 -fno-elide-constructors  
| g():  
|     [...]  
14|     lea rax, [rbp-20]  
14|     mov rdi, rax  
14|     call f()  
14|     lea rdx, [rbp-20]  
14|     lea rax, [rbp-24]  
14|     mov rsi, rdx  
14|     mov rdi, rax  
14|     call S::S(S&&)  
14|     lea rax, [rbp-20]  
14|     mov rdi, rax  
14|     call S::~S()  
15|     mov ebx, DWORD PTR [rbp-24]  
14|     lea rax, [rbp-24]  
14|     mov rdi, rax  
14|     call S::~S()  
15|     mov eax, ebx  
|     [...]
```

RVO in Assembly

```
1 struct S final {  
2     int x;  
3     explicit S(int x) noexcept;  
4     S(const S&) noexcept;  
5     S(S&&) noexcept;  
6     ~S() noexcept;  
7 };  
8  
9 S f() {  
10     return S{42};  
11 }  
12  
13 auto g() {  
14     auto s = f();  
15     return s.x;  
16 }
```

godbolt.org/z/BMgWSl

```
# g92 -fno-elide-constructors  
| f():  
|     [...]  
9 |     mov QWORD PTR [rbp-24], rdi  
10 |     lea rax, [rbp-4]  
10 |     mov esi, 42  
10 |     mov rdi, rax  
10 |     call S::S(int)  
10 |     lea rdx, [rbp-4]  
10 |     mov rax, QWORD PTR [rbp-24]  
10 |     mov rsi, rdx  
10 |     mov rdi, rax  
10 |     call S::S(S&&)  
10 |     lea rax, [rbp-4]  
10 |     mov rdi, rax  
10 |     call S::~S()  
|     [...]
```

RVO in Assembly

```
1 # g92 -fno-elide-constructors
2 f():
3     [...]
4     mov QWORD PTR [rbp-24], rdi
5     lea rax, [rbp-4]
6     mov esi, 42
7     mov rdi, rax
8     call S::S(int)
9     lea rdx, [rbp-4]
10    mov rax, QWORD PTR [rbp-24]
11    mov rsi, rdx
12    mov rdi, rax
13    call S::S(S&&)
14    lea rax, [rbp-4]
15    mov rdi, rax
16    call S::~~S()
17    nop
18    mov rax, QWORD PTR [rbp-24]
19    leave
20    ret
```

```
1 # g92
2 f():
3     [...]
4     mov QWORD PTR [rbp-8], rdi
5     mov rax, QWORD PTR [rbp-8]
6     mov esi, 42
7     mov rdi, rax
8     call S::S(int)
9     mov rax, QWORD PTR [rbp-8]
10    leave
11    ret
```

RVO in Assembly

```
1 # g92 -fno-elide-constructors
2 g():
3     [...]
4     lea rax, [rbp-20]
5     mov rdi, rax
6     call f()
7     lea rdx, [rbp-20]
8     lea rax, [rbp-24]
9     mov rsi, rdx
10    mov rdi, rax
11    call S::S(S&&)
12    lea rax, [rbp-20]
13    mov rdi, rax
14    call S::~~S()
15    mov ebx, DWORD PTR [rbp-24]
16    lea rax, [rbp-24]
17    mov rdi, rax
18    call S::~~S()
19    mov eax, ebx
20    [...]
```

```
1 # g92
2 g():
3     [...]
4     lea rax, [rbp-20]
5     mov rdi, rax
6     call f()
7     mov ebx, DWORD PTR [rbp-20]
8     lea rax, [rbp-20]
9     mov rdi, rax
10    call S::~~S()
11    mov eax, ebx
```

(N)RVO or no (N)RVO?

```
1 #include <utility>
2
3 struct S final {
4     S() noexcept;
5     S(const S&) noexcept;
6     S(S&&) noexcept;
7     ~S() noexcept;
8 };
9
10 S f1() { S s; return s; }
11 S f2() { S s; return std::move(s); }
12 S f3() { const S s; return s; }
13 S f4() { const S s; return std::move(s); }
```

godbolt.org/z/ogqfb4

→ f1: ???

→ f2: ???

→ f3: ???

→ f4: ???

(N)RVO or no (N)RVO?

```
1 #include <utility>
2
3 struct S final {
4     S() noexcept;
5     S(const S&) noexcept;
6     S(S&&) noexcept;
7     ~S() noexcept;
8 };
9
10 S f1() { S s; return s; }
11 S f2() { S s; return std::move(s); }
12 S f3() { const S s; return s; }
13 S f4() { const S s; return std::move(s); }
```

godbolt.org/z/ogqfb4

- f1: RVO
- f2: call `S::S(S&&)`
- f3: RVO
- f4: call `S::S(const S&)` (silently revert to a copy!)

(N)RVO or no (N)RVO?

```
1 struct S {  
2     S() noexcept;  
3     S(const S&) noexcept;  
4     S(const S&&) noexcept;  
5     ~S() noexcept;  
6 };  
7  
8 S f1(S s) { return s; }  
9 S f2(S& s) { return s; }  
10 S f3(const S& s) { return s; }
```

godbolt.org/z/Tak_DF

→ f1: ???

→ f2: ???

→ f3: ???

(N)RVO or no (N)RVO?

```
1 struct S {  
2     S() noexcept;  
3     S(const S&) noexcept;  
4     S(const S&&) noexcept;  
5     ~S() noexcept;  
6 };  
7  
8 S f1(S s) { return s; }  
9 S f2(S& s) { return s; }  
10 S f3(const S& s) { return s; }
```

godbolt.org/z/Tak_DF

- f1: No RVO: the object has to be constructed inside the function
- f2: same as f1
- f3: same as f1

(N)RVO or no (N)RVO?

```
1 struct S final {  
2     S() noexcept;  
3     S(const S&) noexcept;  
4     S(S&&) noexcept;  
5     ~S() noexcept;  
6 };  
7  
8 S f() {  
9     S s;  
10    auto& t = s;  
11    return t;  
12 }
```

godbolt.org/z/ofinrc

(N)RVO or no (N)RVO?

```
1 struct S final {  
2     S() noexcept;  
3     S(const S&) noexcept;  
4     S(S&&) noexcept;  
5     ~S() noexcept;  
6 };  
7  
8 S f() {  
9     S s;  
10    auto& t = s;  
11    return t;  
12 }
```

godbolt.org/z/ofinrc

No RVO: call `S::S(S const&)`

(N)RVO or no (N)RVO?

```
1 struct S {  
2     S(int) noexcept;  
3     S(const S&) noexcept;  
4     S(const S&&) noexcept;  
5     ~S() noexcept;  
6 };  
7  
8 S f1(bool x) { return x ? S{1} : S{2}; }  
9 S f2(bool x) { S s{1}; return x ? s : S{2}; }
```

godbolt.org/z/AYeCQU

→ f1: ???

→ f2: ???

(N)RVO or no (N)RVO?

```
1 struct S {  
2     S(int) noexcept;  
3     S(const S&) noexcept;  
4     S(const S&&) noexcept;  
5     ~S() noexcept;  
6 };  
7  
8 S f1(bool x) { return x ? S{1} : S{2}; }  
9 S f2(bool x) { S s{1}; return x ? s : S{2}; }
```

godbolt.org/z/AYeCQU

- f1: RVO: type of ternary is prvalue
- f2: No RVO: type of ternary is lvalue reference

(N)RVO or no (N)RVO?

```
1 #include <utility>
2
3 struct S final {
4     S() noexcept;
5     S(const S&) noexcept;
6     S(S&&) noexcept;
7     ~S() noexcept;
8 };
9 auto f() { return std::pair<S, S>{}; }
10 S g1() { auto [s1, s2] = f(); return s1; }
11 S g2() { auto&& [s1, s2] = f(); return s1; }
12 S g3() { auto [s1, s2] = f(); return std::move(s1); }
13 S g4() { auto&& [s1, s2] = f(); return std::move(s1); }
```

godbolt.org/z/Yoggsd

→ g1: ???

→ g2: ???

→ g3: ???

→ g4: ???

(N)RVO or no (N)RVO?

```
1 #include <utility>
2
3 struct S final {
4     S() noexcept;
5     S(const S&) noexcept;
6     S(S&&) noexcept;
7     ~S() noexcept;
8 };
9 auto f() { return std::pair<S, S>{}; }
10 S g1() { auto [s1, s2] = f(); return s1; }
11 S g2() { auto&& [s1, s2] = f(); return s1; }
12 S g3() { auto [s1, s2] = f(); return std::move(s1); }
13 S g4() { auto&& [s1, s2] = f(); return std::move(s1); }
```

godbolt.org/z/Yoggsd

- g1: no RVO: call `S::S(const&)`
- g2: same as g1
- g3: no RVO: call `S::S(S&&)`
- g4: same as g3

(N)RVO or no (N)RVO?

(derived from CppCon 2019: *Jason Turner* “Great C++ is_trivial”)

No (N)RVO in any of these examples!

```
1 S g1() { auto [s1, s2] = f(); return s1; } // copy
2 S g2() { auto&& [s1, s2] = f(); return s1; } // copy: no implicit move yet (?)
3 S g3() { auto [s1, s2] = f(); return std::move(s1); } // move
4 S g4() { auto&& [s1, s2] = f(); return std::move(s1); } // move
```

...return std::move is not always bad

Why? Structured bindings:

- Creation of temporary object `e`
- Like a reference: structured binding is an alias into `e`

`return std::move` is not yet necessarily a code smell
(use `-Wpessimizing-move`)

Automatic move from local variables and parameters if:

- return expression names a variable whose type is either
 - an object type or (since C++11)
 - an rvalue reference to object type (since C++20^{*})
- ...and that variable is declared
 - in the body or
 - as a parameter of
- ...the innermost enclosing function or lambda expression

^{*} P1825R0 (not yet implemented in GCC or Clang: cpreference.com/w/cpp/compiler_support)

Perfect Backwarding

Forwarding Values and Preserving Value Category

(derived from CppCon 2018: *Hayun Ezra Chung* “Forwarding Values... and Backwarding Them Too?”)

```
1 #include <iostream>
2 #include <memory>
3
4 struct Resource {};
5
6 struct Target {
7     Target(const Resource&) { std::cout << 'a'; }
8     Target(Resource&&) { std::cout << 'b'; }
9 };
10
11 auto make_target(??? resource) {
12     return std::make_unique<Target>(???);
13 }
14
15 int main() {
16     Resource resource;
17     make_target(resource);           // should print 'a'
18     make_target(Resource(resource)); // should print 'b'
19     make_target(std::move(resource)); // should print 'b'
20 }
```

godbolt.org/z/UAUh_2

Forwarding Values and Preserving Value Category

(derived from CppCon 2018: *Hayun Ezra Chung* “Forwarding Values... and Backwarding Them Too?”)

```
1 #include <iostream>
2 #include <memory>
3
4 struct Resource {};
5
6 struct Target {
7     Target(const Resource&) { std::cout << 'a'; }
8     Target(Resource&&) { std::cout << 'b'; }
9 };
10
11 template <typename T> auto make_target(T&& resource) {
12     return std::make_unique<Target>(std::forward<T>(resource));
13 }
14
15 int main() {
16     Resource resource;
17     make_target(resource);           // lvalue: T = Resource&
18     make_target(Resource(resource)); // prvalue: T = Resource
19     make_target(std::move(resource)); // xvalue: T = Resource
20 }
```

godbolt.org/z/GL4MHb

Forwarding Values and Preserving Value Category

(derived from CppCon 2018: *Hayun Ezra Chung* “Forwarding Values... and Backwarding Them Too?”)

```
1 #include <iostream>
2 #include <memory>
3
4 struct Resource {};
5
6 struct Target {
7     Target(const Resource&) { std::cout << 'a'; }
8     Target(Resource&&) { std::cout << 'b'; }
9 };
10
11 auto make_target(auto&& resource) {
12     return std::make_unique<Target>(std::forward<decltype(resource)>(resource));
13 }
14
15 int main() {
16     Resource resource;
17     make_target(resource);
18     make_target(Resource(resource));
19     make_target(std::move(resource));
20 }
```

godbolt.org/z/YF5i-w

Backwarding Values and Preserving Value Category

```
1 #include <iostream>
2
3 struct Resource {};
4 struct ResourceManager {
5     Resource resource;
6
7     decltype(auto) visit(auto visitor) { return visitor(resource); }
8 };
9 struct Target {
10     Target(const Resource&) { std::cout << 'a'; }
11     Target(Resource&&) { std::cout << 'b'; }
12 };
13
14 int main() {
15     ResourceManager rm;
16     Target(rm.visit([](Resource& r) -> Resource& { return r; }));
17     Target(rm.visit([](Resource& r) -> Resource { return r; }));
18     Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
19 }
```

godbolt.org/z/sm_HLQ

Backwarding Values and Preserving Value Category

```
1  #include <iostream>
2
3  struct Resource {};
4  struct ResourceManager {
5      Resource resource;
6
7      // what if we want to do sth. with the result before returning?
8      decltype(auto) visit(auto visitor) { return visitor(resource); }
9  };
10 struct Target {
11     Target(const Resource&) { std::cout << 'a'; }
12     Target(Resource&&) { std::cout << 'b'; }
13 };
14
15 int main() {
16     ResourceManager rm;
17     Target(rm.visit([](Resource& r) -> Resource& { return r; }));
18     Target(rm.visit([](Resource& r) -> Resource { return r; }));
19     Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
20 }
```

godbolt.org/z/WFk96v

Backwarding Values and Preserving Value Category

Q: Why is this a bad idea?

```
1 auto&& visit(auto visitor) {  
2     auto&& result = visitor(resource);  
3     return result;  
4 }
```

Backwarding Values and Preserving Value Category

Q: Why is this a bad idea?

```
1 auto&& visit(auto visitor) {  
2     auto&& result = visitor(resource);  
3     return result;  
4 }
```

A: Dangling reference for

```
visit([](Resource& r) -> Resource { return r; }));
```

auto&& is always a reference!

Backwarding Values and Preserving Value Category

```
1  #include <iostream>
2
3  struct Resource {};
4  struct ResourceManager {
5      Resource resource;
6
7      ??? visit(auto visitor) {
8          ??? result = visitor(resource);
9          return ???;
10     }
11 };
12 struct Target {
13     Target(const Resource&) { std::cout << 'a'; }
14     Target(Resource&&) { std::cout << 'b'; }
15 };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource { return r; }));
20 }
```

godbolt.org/z/T638sY

Backwarding Values and Preserving Value Category

```
1 #include <iostream>
2
3 struct Resource {};
4 struct ResourceManager {
5     Resource resource;
6
7     Resource visit(auto visitor) {
8         Resource result = visitor(resource);
9         return result;
10    }
11 };
12 struct Target {
13     Target(const Resource&) { std::cout << 'a'; }
14     Target(Resource&&) { std::cout << 'b'; }
15 };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource { return r; }));
20 }
```

godbolt.org/z/eGvxXc

Backwarding Values and Preserving Value Category

```
1 #include <iostream>
2
3 struct Resource {};
4 struct ResourceManager {
5     Resource resource;
6
7     ??? visit(auto visitor) {
8         ??? result = visitor(resource);
9         return ???;
10    }
11 };
12 struct Target {
13     Target(const Resource&) { std::cout << 'a'; }
14     Target(Resource&&) { std::cout << 'b'; }
15 };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource& { return r; }));
20 }
```

godbolt.org/z/nUaqdG

Backwarding Values and Preserving Value Category

```
1 #include <iostream>
2
3 struct Resource {};
4 struct ResourceManager {
5     Resource resource;
6
7     Resource& visit(auto visitor) {
8         Resource& result = visitor(resource);
9         return result;
10    }
11 };
12 struct Target {
13     Target(const Resource&) { std::cout << 'a'; }
14     Target(Resource&&) { std::cout << 'b'; }
15 };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource& { return r; }));
20 }
```

godbolt.org/z/c-ZjBA

Backwarding Values and Preserving Value Category

```
1  #include <iostream>
2
3  struct Resource {};
4  struct ResourceManager {
5      Resource resource;
6
7      ??? visit(auto visitor) {
8          ??? result = visitor(resource);
9          return ???;
10     }
11 };
12 struct Target {
13     Target(const Resource&) { std::cout << 'a'; };
14     Target(Resource&&) { std::cout << 'b'; }
15 };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
20 }
```

godbolt.org/z/b-6EBB

Backwarding Values and Preserving Value Category

```
1 #include <iostream>
2
3 struct Resource {};
4 struct ResourceManager {
5     Resource resource;
6
7     Resource&& visit(auto visitor) {
8         Resource&& result = visitor(resource);
9         return result;
10    }
11 };
12 struct Target {
13     Target(const Resource&) { std::cout << 'a'; };
14     Target(Resource&&) { std::cout << 'b'; }
15 };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
20 }
```

godbolt.org/z/NUxJ9a

Backwarding Values and Preserving Value Category

error: cannot bind rvalue reference of type “Resource&&” to lvalue of type “Resource”

```
1 #include <iostream>
2
3 struct Resource {};
4 struct ResourceManager {
5     Resource resource;
6
7     Resource&& visit(auto visitor) {
8         Resource&& result = visitor(resource);
9         return result;
10    }
11 };
12 struct Target {
13     Target(const Resource&) { std::cout << 'a'; };
14     Target(Resource&&) { std::cout << 'b'; };
15 };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
20 }
```

godbolt.org/z/NUxJ9a

Backwarding Values and Preserving Value Category

```
1 #include <iostream>
2
3 struct Resource {};
4 struct ResourceManager {
5     Resource resource;
6
7     Resource&& visit(auto visitor) {
8         Resource&& result = visitor(resource);
9         return std::move(result); // static_cast<Resource&&>(result)
10    }
11 };
12 struct Target {
13     Target(const Resource&) { std::cout << 'a'; };
14     Target(Resource&&) { std::cout << 'b'; }
15 };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
20 }
```

godbolt.org/z/gPjzkF

How do we fuse these implementations?

```
1 Resource visit(auto visitor) {  
2     Resource result = visitor(resource);  
3     return result;  
4 }  
5  
6 Resource& visit(auto visitor) {  
7     Resource& result = visitor(resource);  
8     return result;  
9 }  
10  
11 Resource&& visit(auto visitor) {  
12     Resource&& result = visitor(resource);  
13     return static_cast<Resource&&>(result);  
14 }
```

```
1 Target(rm.visit([](Resource& r) -> Resource { return r; }));  
2 Target(rm.visit([](Resource& r) -> Resource& { return r; }));  
3 Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
```

How do we fuse these implementations?

```
1 Resource visit(auto visitor) {  
2     Resource result = visitor(resource);  
3     return result;  
4 }  
5  
6 Resource& visit(auto visitor) {  
7     Resource& result = visitor(resource);  
8     return result;  
9 }  
10  
11 Resource&& visit(auto visitor) {  
12     Resource&& result = visitor(resource);  
13     return static_cast<Resource&&>(result);  
14 }
```

```
1 decltype(auto) visit(auto visitor) {  
2     decltype(auto) result = visitor(resource);  
3     return static_cast<decltype(result)>(result);  
4 }
```

```
1 #include <iostream>
2
3 struct Resource {};
4 struct Target {
5     Target(const Resource&) { std::cout << 'a'; }
6     Target(Resource&&) { std::cout << 'b'; }
7 };
8 struct ResourceManager {
9     Resource resource;
10
11     decltype(auto) visit(auto visitor) {
12         decltype(auto) result = visitor(resource);
13         return static_cast<decltype(result)>(result);
14     }
15 };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource& { return r; }));
20     Target(rm.visit([](Resource& r) -> Resource { return r; }));
21     Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
22 }
```



Q: What is the output of the program?

```
1 #include <iostream>
2
3 struct Resource {
4     Resource() {}
5     Resource(const Resource&) { std::cout << 'a'; }
6 };
7 struct ResourceManager {
8     Resource resource;
9
10    Resource visit(auto visitor) {
11        Resource result = visitor(resource);
12        return result;
13    }
14 };
15 struct Target { Target(const Resource&) {} };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource { return r; }));
20 }
```

godbolt.org/z/07JV2m

A: a

```
1 #include <iostream>
2
3 struct Resource {
4     Resource() {}
5     Resource(const Resource&) { std::cout << 'a'; }
6 };
7 struct ResourceManager {
8     Resource resource;
9
10    Resource visit(auto visitor) {
11        Resource result = visitor(resource);
12        return result;
13    }
14 };
15 struct Target { Target(const Resource&) {} };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource { return r; }));
20 }
```

godbolt.org/z/07JV2m

Q: What is the output of the program?

```
1 #include <iostream>
2
3 struct Resource {
4     Resource() {}
5     Resource(const Resource&) { std::cout << 'a'; }
6 };
7 struct ResourceManager {
8     Resource resource;
9
10    Resource visit(auto visitor) {
11        Resource result = visitor(resource);
12        return static_cast<Resource>(result);
13    }
14 };
15 struct Target { Target(const Resource&) {} };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource { return r; }));
20 }
```

godbolt.org/z/A35w4d

```
1 #include <iostream>
2
3 struct Resource {
4     Resource() {}
5     Resource(const Resource&) { std::cout << 'a'; }
6 };
7 struct ResourceManager {
8     Resource resource;
9
10    Resource visit(auto visitor) {
11        Resource result = visitor(resource);
12        return static_cast<Resource>(result);
13    }
14 };
15 struct Target { Target(const Resource&) {} };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource { return r; }));
20 }
```

godbolt.org/z/A35w4d

Missing (N)RVO

```
1 struct Resource {  
2     [...]   
3     Resource(const Resource&) { std::cout << 'a'; }  
4 };  
5  
6 Resource visit(auto visitor) {  
7     Resource result = visitor(resource);  
8     return static_cast<Resource>(result);  
9 }
```

Neither RVO nor NRVO!

- `static_cast` is not the name of a variable (c-style cast does not work either)
- compiler cannot elide observable side effects of copy construction
- “Solution”
 - remove explicit cast, or
 - remove side effect (`std::cout`)

Missing (N)RVO

```
1 template <typename T>
2 decltype(auto) visit(T visitor) {
3     decltype(auto) result = visitor(resource);
4     if constexpr (std::is_same_v<decltype(result), Resource&&>) {
5         return std::move(result);
6     } else {
7         return result;
8     }
9 }
```

...works for GCC (without auto concept), not for Clang though

Missing (N)RVO

```
1  template <typename T>
2  static constexpr bool returns_rref = std::is_same_v<std::invoke_result_t<T,
   ↪ Resource&>, Resource&&>;
3
4  template <typename T, std::enable_if_t<returns_rref<T>, int> = 0>
5  decltype(auto) visit(T visitor) {
6      decltype(auto) result = visitor(resource);
7      return std::move(result);
8  }
9
10 template <typename T, std::enable_if_t<not returns_rref<T>, int> = 0>
11 decltype(auto) visit(T visitor) {
12     decltype(auto) result = visitor(resource);
13     return result;
14 }
```

...still, no NRVO with Clang but this time due to the deduced return type!

Missing (N)RVO

```
1  template <typename T>
2  static constexpr bool returns_rref = std::is_same_v<std::invoke_result_t<T,
   ↪ Resource&>, Resource&&>;
3
4  template <typename T, std::enable_if_t<returns_rref<T>, int> = 0>
5  decltype(auto) visit(T visitor) {
6      decltype(auto) result = visitor(resource);
7      return std::move(result);
8  }
9
10 template <typename T, std::enable_if_t<not returns_rref<T>, int> = 0>
11 auto visit(T visitor) -> decltype(visitor(resource)) {
12     decltype(auto) result = visitor(resource);
13     return result;
14 }
```

...now works for GCC and Clang!

```

1  #include <iostream>
2
3  struct Resource {
4      Resource() {}
5      Resource(const Resource&) { std::cout << 'a'; }
6  };
7  struct ResourceManager {
8      Resource resource;
9
10     template <typename T>
11     static constexpr bool returns_rref = std::is_same_v<std::invoke_result_t<T,
        ↪ Resource&>, Resource&>;
12
13     template <typename T, std::enable_if_t<returns_rref<T>, int> = 0>
14     decltype(auto) visit(T visitor) {
15         decltype(auto) result = visitor(resource);
16         return std::move(result);
17     }
18
19     template <typename T, std::enable_if_t<not returns_rref<T>, int> = 0>
20     [...]

```

godbolt.org/z/Zk66uh

More things that don't work

Missing (N)RVO

```
1 #include <iostream>
2
3 struct Resource {
4     Resource() {}
5     Resource(const Resource&) { std::cout << 'a'; }
6 };
7 struct ResourceManager {
8     Resource resource;
9
10    template <typename T>
11    auto visit(T visitor) -> decltype(visitor(resource)) {
12        using R = std::invoke_result_t<T, Resource&>;
13
14        decltype(auto) result = visitor(resource);
15        if constexpr (std::is_same_v<R, Resource&&>) {
16            return std::move(result);
17        } else {
18            return result;
19        }
20    [...]
21 }
```

godbolt.org/z/uAQaQW

...works with GCC, fails with Clang

Missing (N)RVO

```
1 #include <iostream>
2
3 struct Resource {
4     Resource() {}
5     Resource(const Resource&) { std::cout << 'a'; }
6 };
7 struct ResourceManager {
8     Resource resource;
9
10    template <typename T>
11    auto visit(T visitor) -> decltype(visitor(resource)) {
12        using R = std::invoke_result_t<T, Resource&>;
13        if constexpr (std::is_same_v<R, Resource&&>) {
14            decltype(auto) result = visitor(resource);
15            return std::move(result);
16        } else {
17            decltype(auto) result = visitor(resource);
18            return result;
19        }
20    }
21    [...]
22 }
```

godbolt.org/z/jpCb6h

...works with Clang, fails with GCC

Missing (N)RVO

```
1 #include <iostream>
2
3 struct Resource {
4     Resource() {}
5     Resource(const Resource&) { std::cout << 'a'; }
6 };
7 struct ResourceManager {
8     Resource resource;
9
10    template <typename T>
11    auto visit(T visitor) -> decltype(visitor(resource)) {
12        using R = std::invoke_result_t<T, Resource&>;
13        if constexpr (decltype(auto) result = visitor(resource);
14                        std::is_same_v<R, Resource&&>) {
15            return std::move(result);
16        } else {
17            return result;
18        }
19    }
20    [...]
```

godbolt.org/z/6_YwYf

...fails with GCC and Clang

(shamelessly copied from CppCon 2018: *Hayun Ezra Chung* “Forwarding Values... and Backwarding Them Too?”)

Forwarding

- Parameter Type: `T&&`
- Function Argument: `std::forward<E>(e)`
- Alternatively: `static_cast<decltype(e)&&>(e)`

Backwarding

- Parameter Type: `decltype(auto)`
- Function Argument: `decltype(e)(e)`
- Alternatively: `static_cast<decltype(e)>(e)*`