Object Lifetime:
From Start to Finish

THAMARA ANDRADE

Object Lifetime

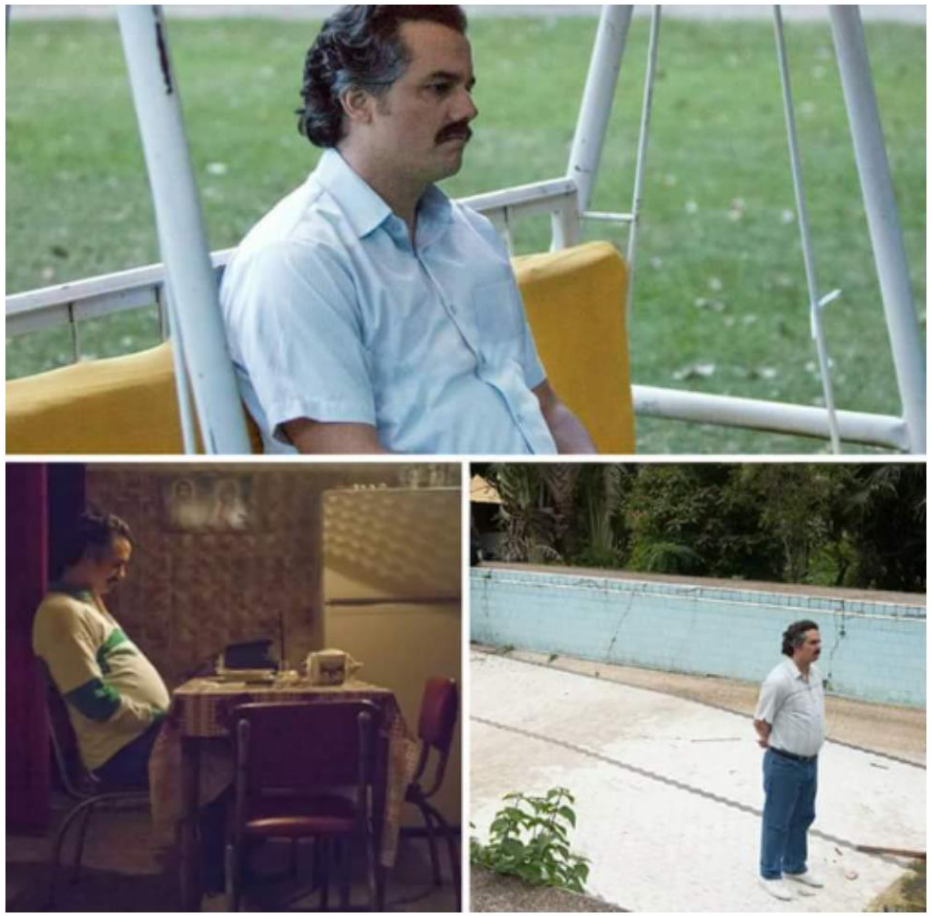
From Start to Finish

Thamara Andrade | https://thamara.dev/

```
int main() {
    if (doSomething(Foo().getBar())) {
        // ...
    }
}
```

```
int main() {
    if (doSomething(Foo().getBar())) {
        // ...
    }
}
```

Me finally understanding why my code was failing…



… and realizing I didn’t really understand the rules for object lifetime.

What is object lifetime anyway?

6.8 Object lifetime [basic.life]

1 The *lifetime* of an object or reference is a runtime property of the object or reference. An object is said to have *non-vacuous initialization* if it is of a class or aggregate type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [*Note:* Initialization by a trivial copy/move constructor is non-vacuous initialization. — *end note*] The lifetime of an object of type `T` begins when:

(1.1) — storage with the proper alignment and size for type `T` is obtained, and

(1.2) — if the object has non-vacuous initialization, its initialization is complete,

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (11.6.1, 15.6.2), or as described in 12.3. The lifetime of an object *o* of type `T` ends when:

(1.3) — if `T` is a class type with a non-trivial destructor (15.4), the destructor call starts, or

(1.4) — the storage which the object occupies is released, or is reused by an object that is not nested within *o* (4.5).

6.8 Object lifetime [basic.life]

1 The *lifetime* of an object or reference is a runtime property of the object or reference. An object is said to have *non-vacuous initialization* if it is of a class or aggregate type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [*Note:* Initialization by a trivial copy/move constructor is non-vacuous initialization. — *end note*] The lifetime of an object of type `T` begins when:

(1.1) — storage with the proper alignment and size for type `T` is obtained, and

(1.2) — if the object has non-vacuous initialization, its initialization is complete,

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (11.6.1, 15.6.2), or as described in 12.3. The lifetime of an object *o* of type `T` ends when:

(1.3) — if `T` is a class type with a non-trivial destructor (15.4), the destructor call starts, or

(1.4) — the storage which the object occupies is released, or is reused by an object that is not nested within *o* (4.5).

On this talk…

You can expect…

- Review of object lifetime
- (A Little of) RAII
- Beyond basic lifetime
- Common pitfalls

but not…

- Value categories
- Unions/Arrays
- Any assembly code

Hi, I’m Thamara (she/her)

- Principal Software Engineer @ Cadence Design Systems
- Learning C++ since 2013
- Can’t decide if ‘`std::`’ is pronounced */stʌd/* or *s-t-d*

`https://thamara.dev/`

6.8 Object lifetime [basic.life]

1 The *lifetime* of an object or reference is a runtime property of the object or reference. An object is said to have *non-vacuous initialization* if it is of a class or aggregate type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [*Note:* Initialization by a trivial copy/move constructor is non-vacuous initialization. — *end note*] The lifetime of an object of type `T` begins when:

(1.1) — storage with the proper alignment and size for type `T` is obtained, and

(1.2) — if the object has non-vacuous initialization, its initialization is complete,

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (11.6.1, 15.6.2), or as described in 12.3. The lifetime of an object o of type `T` ends when:

(1.3) — if `T` is a class type with a non-trivial destructor (15.4), the destructor call starts, or

(1.4) — the storage which the object occupies is released, or is reused by an object that is not nested within o (4.5).

6.8 Object lifetime [basic.life]

1 The *lifetime* of an object or reference is a runtime property of the object or reference. An object is said to have *non-vacuous initialization* if it is of a class or aggregate type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [*Note:* Initialization by a trivial copy/move constructor is non-vacuous initialization. — *end note*] The lifetime of an object of type `T` begins when:

(1.1) — storage with the proper alignment and size for type `T` is obtained, and

(1.2) — if the object has non-vacuous initialization, its initialization is complete,

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (11.6.1, 15.6.2), or as described in 12.3. The lifetime of an object o of type `T` ends when:

(1.3) — if `T` is a class type with a non-trivial destructor (15.4), the destructor call starts, or

(1.4) — the storage which the object occupies is released, or is reused by an object that is not nested within o (4.5).

Storage Duration

	Allocation when	Deallocation when
`static`	program begins	program ends
`thread_local`	thread begins	thread ends
Dynamic	`new`	`delete`
Automatic	enclosing block begins*	enclosing block ends*

Cppcon

The C++ Conference

2023

October 01-06

Aurora, Colorado, USA

Open Content: Jason Turner "C++ Conversation: Storage Durations, Lifetimes, and Puzzlers"

Want to learn more about C++ in a highly interactive way?
We will have a lunchtime group conversation about C++ Object Lifetime.
We will discuss the types of Object Lifetime that exist in C++, and go through some fun puzzle-like exercises together.
Expect a casual, relaxed, fun, interactive setting to spend your lunch time!

Jason Turner

Trainer/Speaker/YouTuber, Jason Turner

Storage Duration

	Allocation when	Deallocation when
`static`	program begins	program ends
`thread_local`	thread begins	thread ends
Dynamic	**`new`**	**`delete`**
Automatic	**enclosing block begins***	**enclosing block ends***

Non-vacous initialization

6.8 Object lifetime [basic.life]

1 The *lifetime* of an object or reference is a runtime property of the object or reference. An object is said to have *non-vacuous initialization* if it is of a class or aggregate type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [*Note:* Initialization by a trivial copy/move constructor is non-vacuous initialization. — *end note*]

Non-vacuous initialization

```
struct ObjWithConstructor {
    ObjWithConstructor() {}
};
ObjWithConstructor o1;

struct ObjWithVirtualFunction {
    virtual void foo() {}
};
ObjWithVirtualFunction o2;
```

Vacuous initialization

```
int num;
float pi;
bool flag;

struct EmptyPoint {};
EmptyPoint ep;

struct Point { int x, y; };
Point p;
```

When lifetime starts?

When lifetime starts?

6.8 Object lifetime [basic.life]

1 The *lifetime* of an object or reference is a runtime property of the object or reference. An object is said to have *non-vacuous initialization* if it is of a class or aggregate type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [*Note:* Initialization by a trivial copy/move constructor is non-vacuous initialization. — *end note*] The lifetime of an object of type `T` begins when:

(1.1) — storage with the proper alignment and size for type `T` is obtained, and

(1.2) — if the object has non-vacuous initialization, its initialization is complete,

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (11.6.1, 15.6.2), or as described in 12.3. The lifetime of an object o of type `T` ends when:

(1.3) — if `T` is a class type with a non-trivial destructor (15.4), the destructor call starts, or

(1.4) — the storage which the object occupies is released, or is reused by an object that is not nested within o (4.5).

Constructors & Destructors

Constructors & Destructors

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5
 6 int main() {
 7    Foo a;
 8    {
 9       Foo b;
10    }
11 }
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```
 1 struct Foo {
 2     Foo() { std::cout << "Foo()" << std::endl; }
 3     ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5
 6 int main() {
 7     Foo a;
 8     {
 9         Foo b;
10     }
11 }
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5
 6 int main() {
 7    Foo a; // Foo()
 8    {
 9       Foo b;
10    }
11 }
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```
 1 struct Foo {
 2     Foo() { std::cout << "Foo()" << std::endl; }
 3     ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5
 6 int main() {
 7     Foo a; // Foo()
 8     {
 9         Foo b;
10     }
11 }
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```
 1 struct Foo {
 2     Foo() { std::cout << "Foo()" << std::endl; }
 3     ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5
 6 int main() {
 7     Foo a; // Foo()
 8     {
 9         Foo b;
10     }
11 }
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```
 1 struct Foo {
 2     Foo() { std::cout << "Foo()" << std::endl; }
 3     ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5
 6 int main() {
 7     Foo a; // Foo()
 8     {
 9         Foo b; // Foo()
10     }
11 }
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```
 1 struct Foo {
 2     Foo() { std::cout << "Foo()" << std::endl; }
 3     ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5
 6 int main() {
 7     Foo a; // Foo()
 8     {
 9         Foo b; // Foo()
10     }
11 }
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```
 1 struct Foo {
 2     Foo() { std::cout << "Foo()" << std::endl; }
 3     ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5
 6 int main() {
 7     Foo a; // Foo()
 8     {
 9         Foo b; // Foo()
10     } // ~Foo() of b
11 }
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5
 6 int main() {
 7    Foo a; // Foo()
 8    {
 9       Foo b; // Foo()
10    } // ~Foo() of b
▶11 }
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5
 6 int main() {
 7    Foo a; // Foo()
 8    {
 9       Foo b; // Foo()
10    } // ~Foo() of b
▶11 } // ~Foo() of a
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors (2)

Constructors & Destructors (2)

```
1  struct Foo {
2     Foo() { std::cout << "Foo()" << std::endl; }
3     ~Foo() { std::cout << "~Foo()" << std::endl; }
4  };
5  int main() {
6     Foo* a = nullptr;
7     {
8        a = new Foo();
9     }
10    delete a;
11 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5 int main() {
 6    Foo* a = nullptr;
 7    {
 8       a = new Foo();
 9    }
10    delete a;
11 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5 int main() {
 6    Foo* a = nullptr;
 7    {
 8       a = new Foo();
 9    }
10    delete a;
11 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5 int main() {
 6    Foo* a = nullptr;
 7    {
 8       a = new Foo();
 9    }
10    delete a;
11 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```
 1 struct Foo {
 2     Foo() { std::cout << "Foo()" << std::endl; }
 3     ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5 int main() {
 6     Foo* a = nullptr;
 7     {
 8         a = new Foo(); // Foo()
 9     }
10     delete a;
11 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5 int main() {
 6    Foo* a = nullptr;
 7    {
 8       a = new Foo(); // Foo()
 9    }
10    delete a;
11 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5 int main() {
 6    Foo* a = nullptr;
 7    {
 8       a = new Foo(); // Foo()
 9    } // Foo a still in memory
10    delete a;
11 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5 int main() {
 6    Foo* a = nullptr;
 7    {
 8       a = new Foo(); // Foo()
 9    } // Foo a still in memory
10    delete a;
11 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5 int main() {
 6    Foo* a = nullptr;
 7    {
 8       a = new Foo(); // Foo()
 9    } // Foo a still in memory
10    delete a; // ~Foo()
11 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```
 1 struct Foo {
 2    Foo() { std::cout << "Foo()" << std::endl; }
 3    ~Foo() { std::cout << "~Foo()" << std::endl; }
 4 };
 5 int main() {
 6    Foo* a = nullptr;
 7    {
 8       a = new Foo(); // Foo()
 9    } // Foo a still in memory
10    delete a; // ~Foo()
▶11 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (3)

```
 1 struct Foo {
 2     Foo() {
 3         std::cout << "Foo()" << std::endl;
 4         throw std::runtime_error("Exception in constructor");
 5     }
 6     ~Foo() { std::cout << "~Foo()" << std::endl; }
 7 };
 8 int main() {
 9     try {
10         Foo a;
11     } catch (...) {}
12 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (3)

```
 1 struct Foo {
 2     Foo() {
 3         std::cout << "Foo()" << std::endl;
 4         throw std::runtime_error("Exception in constructor");
 5     }
 6     ~Foo() { std::cout << "~Foo()" << std::endl; }
 7 };
 8 int main() {
 9     try {
10         Foo a;
11     } catch (...) {}
12 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (3)

```
 1 struct Foo {
 2     Foo() {
 3         std::cout << "Foo()" << std::endl;
 4         throw std::runtime_error("Exception in constructor");
 5     }
 6     ~Foo() { std::cout << "~Foo()" << std::endl; }
 7 };
 8 int main() {
 9     try {
10         Foo a;
11     } catch (...) {}
12 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (3)

```
 1 struct Foo {
 2     Foo() {
 3         std::cout << "Foo()" << std::endl;
 4         throw std::runtime_error("Exception in constructor");
 5     }
 6     ~Foo() { std::cout << "~Foo()" << std::endl; }
 7 };
 8 int main() {
 9     try {
10         Foo a;
11     } catch (...) {}
12 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (3)

```
 1 struct Foo {
 2     Foo() {
 3         std::cout << "Foo()" << std::endl;
 4         throw std::runtime_error("Exception in constructor");
 5     }
 6     ~Foo() { std::cout << "~Foo()" << std::endl; }
 7 };
 8 int main() {
 9     try {
10         Foo a; // Foo()
11     } catch (...) {}
12 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (3)

```
 1 struct Foo {
 2     Foo() {
 3         std::cout << "Foo()" << std::endl;
 4         throw std::runtime_error("Exception in constructor");
 5     }
 6     ~Foo() { std::cout << "~Foo()" << std::endl; }
 7 };
 8 int main() {
 9     try {
10         Foo a; // Foo() <<< But not really initialized
11     } catch (...) {}
12 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (3)

```
 1 struct Foo {
 2     Foo() {
 3         std::cout << "Foo()" << std::endl;
 4         throw std::runtime_error("Exception in constructor");
 5     }
 6     ~Foo() { std::cout << "~Foo()" << std::endl; }
 7 };
 8 int main() {
 9     try {
10         Foo a; // Foo() <<< But not really initialized
11     } catch (...) {}
12 }
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (3)

```
 1 struct Foo {
 2     Foo() {
 3         std::cout << "Foo()" << std::endl;
 4         throw std::runtime_error("Exception in constructor");
 5     }
 6     ~Foo() { std::cout << "~Foo()" << std::endl; }
 7 };
 8 int main() {
 9     try {
10         Foo a; // Foo() <<< But not really initialized
11     } catch (...) {}
12 }
```


https://godbolt.org/z/hY7hr5Efr

When lifetime starts?

6.8 Object lifetime [basic.life]

1 The *lifetime* of an object or reference is a runtime property of the object or reference. An object is said to have *non-vacuous initialization* if it is of a class or aggregate type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [*Note:* Initialization by a trivial copy/move constructor is non-vacuous initialization. — *end note*] The lifetime of an object of type `T` begins when:

(1.1) — storage with the proper alignment and size for type `T` is obtained, and

(1.2) — if the object has non-vacuous initialization, its initialization is complete,

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (11.6.1, 15.6.2), or as described in 12.3. The lifetime of an object o of type `T` ends when:

(1.3) — if `T` is a class type with a non-trivial destructor (15.4), the destructor call starts, or

(1.4) — the storage which the object occupies is released, or is reused by an object that is not nested within o (4.5).

When lifetime starts and finishes?

6.8 Object lifetime [basic.life]

1 The *lifetime* of an object or reference is a runtime property of the object or reference. An object is said to have *non-vacuous initialization* if it is of a class or aggregate type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [*Note:* Initialization by a trivial copy/move constructor is non-vacuous initialization. — *end note*] The lifetime of an object of type `T` begins when:

(1.1) — storage with the proper alignment and size for type `T` is obtained, and

(1.2) — if the object has non-vacuous initialization, its initialization is complete,

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (11.6.1, 15.6.2), or as described in 12.3. The lifetime of an object *o* of type `T` ends when:

(1.3) — if `T` is a class type with a non-trivial destructor (15.4), the destructor call starts, or

(1.4) — the storage which the object occupies is released, or is reused by an object that is not nested within *o* (4.5).

```
1  using namespace std;
2
3  struct Foo {
4      Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
5      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
6      int m_i;
7  };
8
9  int main() {
10     auto v1 = Foo(1);
11     {
12         auto v2 = Foo(2);
13     }
14     auto v3 = Foo(3);
15     auto v4 = new Foo(4);
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);
11     {
12         auto v2 = Foo(2);
13     }
14     auto v3 = Foo(3);
15     auto v4 = new Foo(4);
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);          // Foo(1)
11     {
12         auto v2 = Foo(2);
13     }
14     auto v3 = Foo(3);
15     auto v4 = new Foo(4);
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);          // Foo(1)
11     {
12         auto v2 = Foo(2);
13     }
14     auto v3 = Foo(3);
15     auto v4 = new Foo(4);
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);          // Foo(1)
11     {
12         auto v2 = Foo(2);      // Foo(2)
13     }
14     auto v3 = Foo(3);
15     auto v4 = new Foo(4);
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);          // Foo(1)
11     {
12         auto v2 = Foo(2);      // Foo(2)
13     }
14     auto v3 = Foo(3);
15     auto v4 = new Foo(4);
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);          // Foo(1)
11     {
12         auto v2 = Foo(2);      // Foo(2)
13     }                          // ~Foo(2)
14     auto v3 = Foo(3);
15     auto v4 = new Foo(4);
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);          // Foo(1)
11     {
12         auto v2 = Foo(2);      // Foo(2)
13     }                          // ~Foo(2)
14     auto v3 = Foo(3);
15     auto v4 = new Foo(4);
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);          // Foo(1)
11     {
12         auto v2 = Foo(2);      // Foo(2)
13     }                          // ~Foo(2)
14     auto v3 = Foo(3);          // Foo(3)
15     auto v4 = new Foo(4);
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);          // Foo(1)
11     {
12         auto v2 = Foo(2);      // Foo(2)
13     }                          // ~Foo(2)
14     auto v3 = Foo(3);          // Foo(3)
15     auto v4 = new Foo(4);
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);          // Foo(1)
11     {
12         auto v2 = Foo(2);      // Foo(2)
13     }                          // ~Foo(2)
14     auto v3 = Foo(3);          // Foo(3)
15     auto v4 = new Foo(4);      // Foo(4)
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);          // Foo(1)
11     {
12         auto v2 = Foo(2);      // Foo(2)
13     }                          // ~Foo(2)
14     auto v3 = Foo(3);          // Foo(3)
15     auto v4 = new Foo(4);      // Foo(4)
16 }
```


https://godbolt.org/z/4WroY5311

```
 1 using namespace std;
 2
 3 struct Foo {
 4     Foo(int i) : m_i(i) { cout << "Foo(" << m_i << ")" << endl; }
 5     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
 6     int m_i;
 7 };
 8
 9 int main() {
10     auto v1 = Foo(1);          // Foo(1)
11     {
12         auto v2 = Foo(2);      // Foo(2)
13     }                          // ~Foo(2)
14     auto v3 = Foo(3);          // Foo(3)
15     auto v4 = new Foo(4);      // Foo(4)
16 }                              // ~Foo(3), ~Foo(1)
```


https://godbolt.org/z/4WroY5311

When lifetime starts and finishes?

6.8 Object lifetime [basic.life]

1 The *lifetime* of an object or reference is a runtime property of the object or reference. An object is said to have *non-vacuous initialization* if it is of a class or aggregate type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [*Note:* Initialization by a trivial copy/move constructor is non-vacuous initialization. — *end note*] The lifetime of an object of type `T` begins when:

(1.1) — storage with the proper alignment and size for type `T` is obtained, and

(1.2) — if the object has non-vacuous initialization, its initialization is complete,

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (11.6.1, 15.6.2), or as described in 12.3. The lifetime of an object *o* of type `T` ends when:

(1.3) — if `T` is a class type with a non-trivial destructor (15.4), the destructor call starts, or

(1.4) — the storage which the object occupies is released, or is reused by an object that is not nested within *o* (4.5).

RAII

Resource Acquisition Is Initialization

RAII

Resource Acquisition Is Initialization

Tying resource acquisition and deallocation to **object lifetime**.

```
1  struct DynamicArray {
2     explicit DynamicArray(size_t sz) : m_d(new int[sz]) {}
3     ~DynamicArray() { delete[] m_d; }
4     // ...
5     int* m_d;
6  };
7  int main() {
8     DynamicArray arr(5);
9     // Populate/work on array
10    return 0;
11 }
```

```
1  struct DynamicArray {
2      explicit DynamicArray(size_t sz) : m_d(new int[sz]) {}
3      ~DynamicArray() { delete[] m_d; }
4      // ...
5      int* m_d;
6  };
7  int main() {
8      DynamicArray arr(5);
9      // Populate/work on array
10     return 0;
11 }
```

More RAII usage

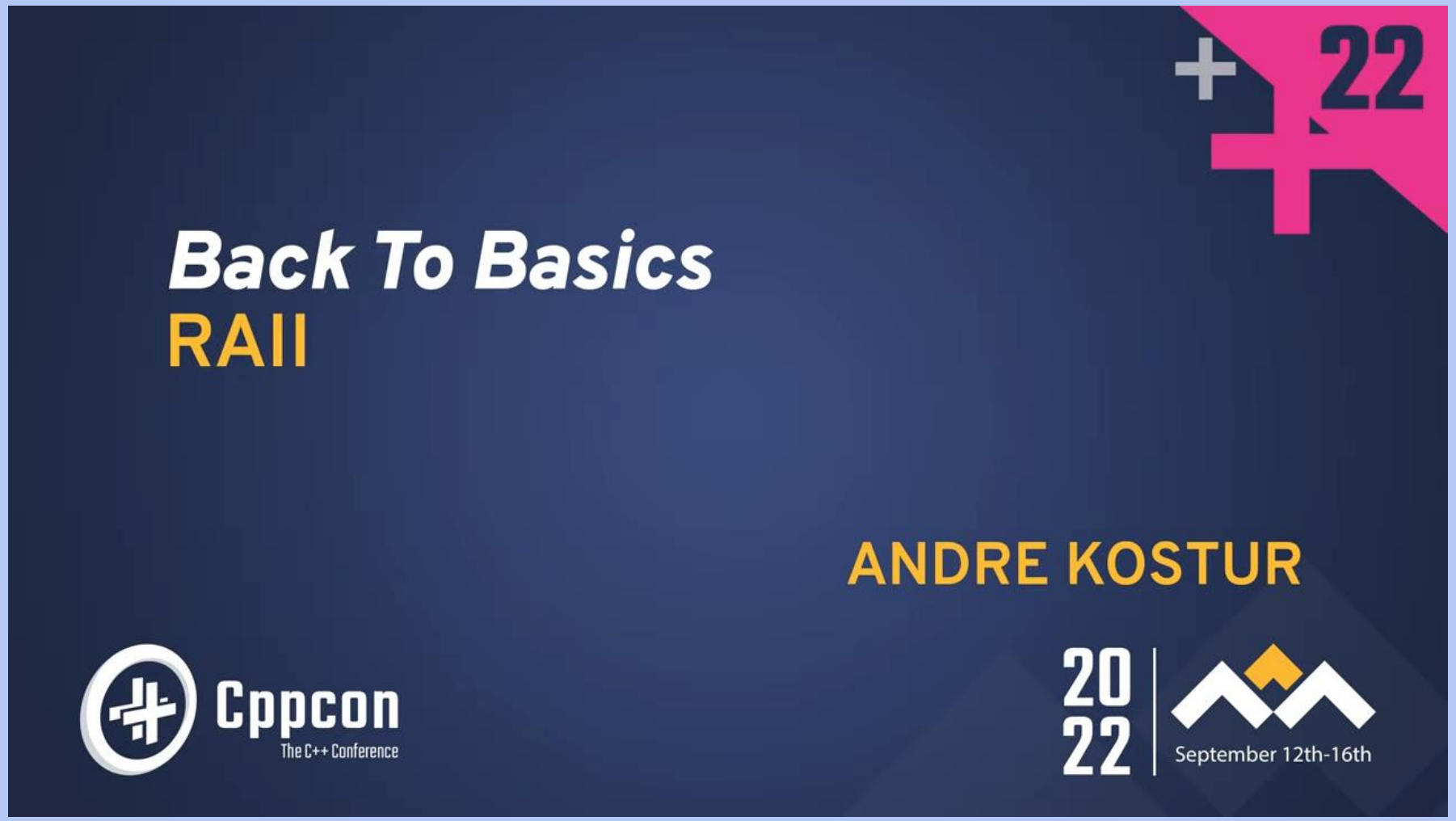
Heap memory allocation	`std::shared_ptr` `std::unique_ptr`
Mutexes	`std::lock_guard` `std::unique/shared_lock`
Threads	`std::jthread`
File management	`std::ofstream`*

Including on the STL

More RAII usage

Heap memory allocation	`std::shared_ptr` `std::unique_ptr`
Mutexes	`std::lock_guard` `std::unique/shared_lock`
Threads	`std::jthread`
File management	`std::ofstream`*

Including most of the STL



https://youtu.be/Rfu06XAhx90?si=GGa_g9w18RN_xUpG

Object Lifetime

From Start to Finish

Thamara Andrade | https://thamara.dev/

Object Lifetime

From **Start** to **Finish**,

and the **tricky parts all around**

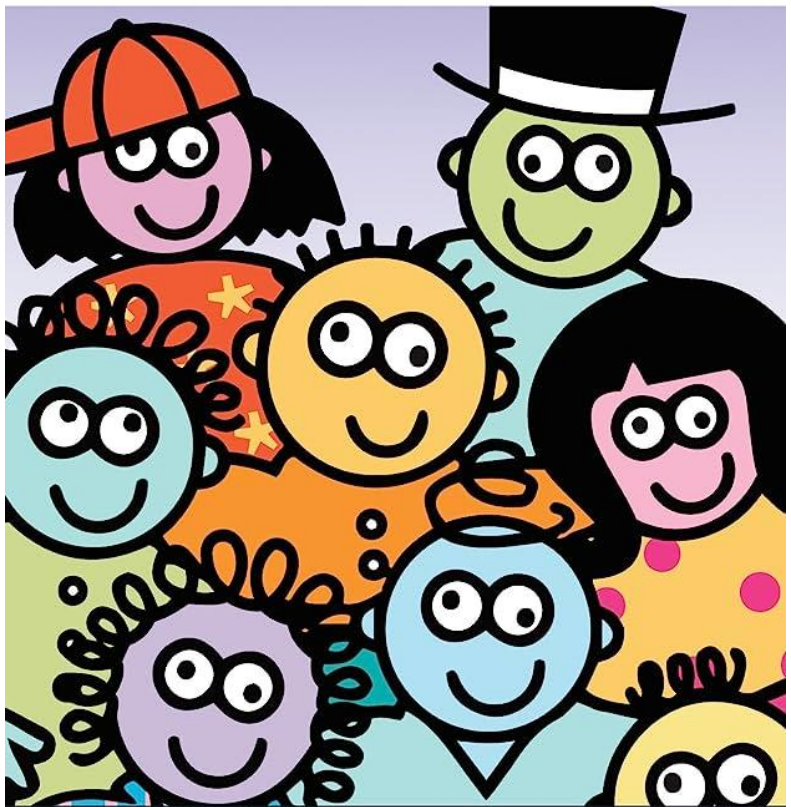
Thamara Andrade | https://thamara.dev/

tricky parts all around

The Original #1 Mad Libs

MAD LIBS®

World's Greatest Word Game



A super silly way to fill in the _____!
PLURAL NOUN

Whether chipping away at a/an _____ statue or stitching
ADJECTIVE
a patchwork _____, crafting is always a labor of
NOUN
_____. But sometimes the most _____ part of
NOUN ADJECTIVE
producing art is deciding what to _____ next! Luckily, the
VERB
Internet can lend a helping _____. There are plenty
PART OF THE BODY
of mood boards and photo _____ online to consult for
PLURAL NOUN
inspiration. It doesn't matter if you're redesigning (the) _____,
A PLACE
painting a/an _____, or hosting a dinner party for
NOUN
_____, the Internet will have plenty of _____
NUMBER ADJECTIVE
advice. And if you're feeling _____, you can create your own
ADJECTIVE
_____ board and inspire dozens of followers with your
NOUN
_____ designs. With an infinite number of new projects to
ADJECTIVE
_____, the only challenge will be finding the
VERB
_____ to complete them all!
NOUN

```
1  struct Foo;
2  struct Obj {
3      _____ getFoo() { ... }
4      ...
5  };
6  int main() {
7      Obj o;
8      _____ val = o.getFoo();
9  }
```

const Foo&　　const Foo　　Foo&　　Foo

```
1  struct Foo;
2  struct Obj {
3      _____________ getFoo() { ... }
4      ...
5  };
6  int main() {
7      Obj o;
8      _____________ val = o.getFoo();
9  }
```

const Foo&　　const Foo　　Foo&　　Foo

```
1  struct Foo;
2  struct Obj {
3      _____________ getFoo() { ... }
4      ...
5  };
6  int main() {
7      Obj o;
8      _____________ val = o.getFoo();
9  }
```


https://abseil.io/tips/101

const Foo& const Foo Foo& **Foo**

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      _____________ getFoo() { ... }
4      ...
5  };
6  int main() {
7      Obj o;
8      _____________ val = o.getFoo();
9  }
```

1/9

https://abseil.io/tips/101

const Foo& const Foo Foo& **Foo**

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      ____Foo____ getFoo() { ... }
4      ...
5  };
6  int main() {
7      Obj o;
8      ____Foo____ val = o.getFoo();
9  }
```

1/9

https://abseil.io/tips/101

const Foo& const Foo Foo& **Foo**

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      ____Foo____ getFoo() { return Foo(); }
4      ...
5  };
6  int main() {
7      Obj o;
8      ____Foo____ val = o.getFoo();
9  }
```

1/9

https://abseil.io/tips/101

const Foo&　　const Foo　　Foo&　　**Foo**

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      Foo getFoo() { return Foo(); }
4      ...
5  };
6  int main() {
7      Obj o;
8      Foo val = o.getFoo();
9  }
```

1/9

https://abseil.io/tips/101

const Foo& const Foo Foo& **Foo**

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      Foo getFoo() { return Foo(); }
4      ...
5  };
6  int main() {
7      Obj o;
8      Foo val = o.getFoo();
9  }
```

1/9

https://abseil.io/tips/101

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      Foo  getFoo() { return Foo(); }
4      ...
5  };
6  int main() {
7      Obj o;
8      Foo  val = o.getFoo();
9  }
```

Temporary is initialized directly in `val`'s storage

Return Value Optimization (RVO)

1/9

https://abseil.io/tips/101

const Foo&　　const Foo　　Foo&　　**Foo**

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      Foo  getFoo() { ... }
4      ...
5  };
6  int main() {
7      Obj o;
8      Foo  val = o.getFoo();
9  }
```

Usually one object,
at most Foo(Foo&&)

https://abseil.io/tips/101

const Foo&　　const Foo　　**Foo&**　　**Foo**

Return　　Initialize

```
1  struct Foo;
2  struct Obj {
3        Foo&      getFoo() { ... }
4     ...
5  };
6  int main() {
7     Obj o;
8        Foo       val = o.getFoo();
9  }
```

const Foo&	const Foo	Foo&	Foo
		Return	Initialize

```
1  struct Foo;
2  struct Obj {
3      Foo&  getFoo() { ... }
4    ...
5  };
6  int main() {
7    Obj o;
8      Foo  val = o.getFoo();
9  }
```

Foo(const Foo&)

https://abseil.io/tips/101

const Foo& const Foo Foo& **Foo**

Return Initialize

```
1  struct Foo;
2  struct Obj {
3      const Foo& getFoo() { ... }
4      ...
5  };
6  int main() {
7      Obj o;
8      Foo val = o.getFoo();
9  }
```

Foo(const Foo&)

https://abseil.io/tips/101

Cppcon

The C++ Conference

2023

October 01-06

Aurora, Colorado, USA

Back to Basics: The Rule of Five

Designing a class to behave correctly when copied and moved takes a lot of thought. The Core Guidelines provide guidance to streamline that work. In this talk we are going to look at the Core Guideline known as "the Rule of Five", how it came about, and is there anything better.

Andre Kostur

Software Engineer, Arista Networks

Friday October 6, 2023 09:00 - 10:00 MDT

Maple 3/4/5

• Back to Basics

const Foo& const Foo Foo& Foo

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      const Foo& getFoo() { ... }
4      ...
5  };
6  int main() {
7      Obj o;
8      const Foo& val = o.getFoo();
9  }
```


https://abseil.io/tips/101

const Foo& const Foo Foo& Foo

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      const Foo& getFoo() { return m_foo; }
4      ...
5  };
6  int main() {
7      Obj o;
8      const Foo& val = o.getFoo();
9  }
```


https://abseil.io/tips/101

const Foo& const Foo Foo& Foo

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      const Foo& getFoo() { return m_foo; }
4      ...
5  };
6  int main() {
7      Obj o;
8      const Foo& val = o.getFoo();
9  }
```

Dangling references

Dangling references

```
1  struct Bar {
2      Bar() { ... }
3      ~Bar() { ... }
4      int m_i{3};
5  };
5
6  struct Foo {
7      Foo() { ... }
8      ~Foo() { ... }
9      const Bar& getBar() {
10        return m_bar; }
11     Bar m_bar;
12 };
```

Dangling references

```
1  struct Bar {
2      Bar() { ... }
3      ~Bar() { ... }
4      int m_i{3};
5  };
5
6  struct Foo {
7      Foo() { ... }
8      ~Foo() { ... }
9      const Bar& getBar() {
10        return m_bar; }
11     Bar m_bar;
12 };
```

```
13 int main() {
14     Foo* f = new Foo();
15     const Bar& b = f->getBar();
16     auto i1 = b.m_i;
17     delete f;
18     auto i2 = b.m_i;
19 }
```

Dangling references

```
1  struct Bar {
2      Bar() { ... }
3      ~Bar() { ... }
4      int m_i{3};
5  };
5
6  struct Foo {
7      Foo() { ... }
8      ~Foo() { ... }
9      const Bar& getBar() {
10        return m_bar; }
11     Bar m_bar;
12 };
```

```
13 int main() {
14     Foo* f = new Foo();
15     const Bar& b = f->getBar();
16     auto i1 = b.m_i;
17     delete f;
18     auto i2 = b.m_i;
19 }
```

Dangling references

```
1  struct Bar {
2      Bar() { ... }
3      ~Bar() { ... }
4      int m_i{3};
5  };
5
6  struct Foo {
7      Foo() { ... }
8      ~Foo() { ... }
9      const Bar& getBar() {
10        return m_bar; }
11     Bar m_bar;
12 };
```

```
13 int main() {
14     Foo* f = new Foo(); // Bar(), Foo()
15     const Bar& b = f->getBar();
16     auto i1 = b.m_i;
17     delete f;
18     auto i2 = b.m_i;
19 }
```

Dangling references

```
1  struct Bar {
2      Bar() { ... }
3      ~Bar() { ... }
4      int m_i{3};
5  };
5
6  struct Foo {
7      Foo() { ... }
8      ~Foo() { ... }
9      const Bar& getBar() {
10        return m_bar; }
11     Bar m_bar;
12 };
```

```
13 int main() {
14     Foo* f = new Foo(); // Bar(), Foo()
15     const Bar& b = f->getBar();
16     auto i1 = b.m_i;
17     delete f;
18     auto i2 = b.m_i;
19 }
```

Dangling references

```
1  struct Bar {
2      Bar() { ... }
3      ~Bar() { ... }
4      int m_i{3};
5  };
5
6  struct Foo {
7      Foo() { ... }
8      ~Foo() { ... }
9      const Bar& getBar() {
10        return m_bar; }
11     Bar m_bar;
12 };
```

```
13 int main() {
14     Foo* f = new Foo(); // Bar(), Foo()
15     const Bar& b = f->getBar();
16     auto i1 = b.m_i;
17     delete f;
18     auto i2 = b.m_i;
19 }
```

Dangling references

```
1  struct Bar {
2      Bar() { ... }
3      ~Bar() { ... }
4      int m_i{3};
5  };
5
6  struct Foo {
7      Foo() { ... }
8      ~Foo() { ... }
9      const Bar& getBar() {
10        return m_bar; }
11     Bar m_bar;
12 };
```

```
13 int main() {
14     Foo* f = new Foo(); // Bar(), Foo()
15     const Bar& b = f->getBar();
16     auto i1 = b.m_i; // <<< OK!
17     delete f;
18     auto i2 = b.m_i;
19 }
```

Dangling references

```
1  struct Bar {
2      Bar() { ... }
3      ~Bar() { ... }
4      int m_i{3};
5  };
5
6  struct Foo {
7      Foo() { ... }
8      ~Foo() { ... }
9      const Bar& getBar() {
10        return m_bar; }
11     Bar m_bar;
12 };
```

```
13 int main() {
14     Foo* f = new Foo(); // Bar(), Foo()
15     const Bar& b = f->getBar();
16     auto i1 = b.m_i; // <<< OK!
17     delete f;
18     auto i2 = b.m_i;
19 }
```

Dangling references

```
1  struct Bar {
2      Bar() { ... }
3      ~Bar() { ... }
4      int m_i{3};
5  };
5
6  struct Foo {
7      Foo() { ... }
8      ~Foo() { ... }
9      const Bar& getBar() {
10        return m_bar; }
11     Bar m_bar;
12 };
```

```
13 int main() {
14     Foo* f = new Foo(); // Bar(), Foo()
15     const Bar& b = f->getBar();
16     auto i1 = b.m_i; // <<< OK!
17     delete f; // ~Foo(), ~Bar()
18     auto i2 = b.m_i;
19 }
```

Dangling references

```
1  struct Bar {
2      Bar() { ... }
3      ~Bar() { ... }
4      int m_i{3};
5  };
5
6  struct Foo {
7      Foo() { ... }
8      ~Foo() { ... }
9      const Bar& getBar() {
10        return m_bar; }
11     Bar m_bar;
12 };
```

```
13 int main() {
14     Foo* f = new Foo(); // Bar(), Foo()
15     const Bar& b = f->getBar();
16     auto i1 = b.m_i; // <<< OK!
17     delete f; // ~Foo(), ~Bar()
18     auto i2 = b.m_i;
19 }
```

Dangling references

```
1  struct Bar {
2      Bar() { ... }
3      ~Bar() { ... }
4      int m_i{3};
5  };
5
6  struct Foo {
7      Foo() { ... }
8      ~Foo() { ... }
9      const Bar& getBar() {
10        return m_bar; }
11     Bar m_bar;
12 };
```

```
13 int main() {
14     Foo* f = new Foo(); // Bar(), Foo()
15     const Bar& b = f->getBar();
16     auto i1 = b.m_i; // <<< OK!
17     delete f; // ~Foo(), ~Bar()
18     auto i2 = b.m_i; // <<< Oh no! UB!
19 }
```

Dangling references

= UB ☹

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      const Foo& getFoo() { return m_foo; }
4      ...
5  };
6  int main() {
7      Obj o;
8      const Foo& val = o.getFoo();
9  }
```

const Foo& const Foo **Foo&** Foo

Initialize Return

```
1  struct Foo;
2  struct Obj {
3      Foo&       getFoo() { return m_foo; }
4      ...
5  };
6  int main() {
7      Obj o;
8      const Foo& val = o.getFoo();
9  }
```

5/9

https://abseil.io/tips/101

const Foo& const Foo **Foo&** Foo

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      ____Foo&____ getFoo() { return m_foo; }
4      ...
5  };
6  int main() {
7      Obj o;
8      ____Foo&____ val = o.getFoo();
9  }
```

123

https://abseil.io/tips/101

const Foo& const Foo **Foo&** Foo

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      ____Foo&____ getFoo() { return m_foo; }
4      ...
5  };
6  int main() {
7      Obj o;
8      ____Foo&____ val = o.getFoo();
9  }
```

https://abseil.io/tips/101

Return
Initialize

```
1  struct Foo;
2  struct Obj {
3      Foo&  getFoo() { return m_foo; }
4      ...
5  };
6  int main() {
7      Obj o;
8      Foo&  val = o.getFoo();
9  }
```

Be careful with objects you don't know/control the lifetime!

const Foo&　　const Foo　　Foo&　　Foo

```
1  struct Foo;
2  struct Obj {
3      _____________ getFoo() { ... }
4       ...
5  };
6  int main() {
7       Obj o;
8      _____________ val = o.getFoo();
9  }
```


https://abseil.io/tips/101

`const Foo&`	`const Foo`	`Foo&`	`Foo`
Return		Initialize	

```
1  struct Foo;
2  struct Obj {
3        const Foo& getFoo() { ... }
4        ...
5  };
6  int main() {
7        Obj o;
8        Foo&       val = o.getFoo();
9  }
```

const Foo& const Foo **Foo&** Foo

Return Initialize

```
1  struct Foo;
2  struct Obj {
3      const Foo& getFoo() { ... }
4      ...
5  };
6  int main() {
7      Obj o;
8      Foo& val = o.getFoo();
9  }
```

```
error: binding reference of type ‘Foo' to value of type 'const
Foo' drops 'const' qualifier
```


https://abseil.io/tips/101

const Foo&	const Foo	Foo&	Foo
		Initialize	Return

```
1  struct Foo;
2  struct Obj {
3        ____Foo____ getFoo() { ... }
4        ...
5  };
6  int main() {
7        Obj o;
8        ____Foo&____ val = o.getFoo();
9  }
```


https://abseil.io/tips/101

const Foo& const Foo **Foo&** **Foo**

Initialize Return

```
1  struct Foo;
2  struct Obj {
3        Foo      getFoo() { ... }
4        ...
5  };
6  int main() {
7        Obj o;
8        Foo&     val = o.getFoo();
9  }
```

error: non-const lvalue reference to type ‘Foo' cannot bind to a temporary of type ‘Foo'

8/9

https://abseil.io/tips/101

Initialize

Return

```
1  struct Foo;
2  struct Obj {
3        Foo      getFoo() { ... }
4        ...
5  };
6  int main() {
7        Obj o;
8        const Foo& val = o.getFoo();
9  }
```

const Foo& | const Foo | Foo& | **Foo**

Initialize | Return

```
1  struct Foo;
2  struct Obj {
3        Foo      getFoo() { ... }
4        ...
5  };
6  int main() {
7        Obj o;
8        const Foo& val = o.getFoo();
9  }
```


https://abseil.io/tips/101

const Foo&	const Foo	Foo&	Foo
Initialize			Return

```
1  struct Foo;
2  struct Obj {
3        Foo      getFoo() { ... }
4        ...
5  };
6  int main() {
7        Obj o;
8        const Foo& val = o.getFoo();
9  }
```

https://abseil.io/tips/101

Initialize

Return

```
1  struct Foo;
2  struct Obj {
3        Foo  getFoo() { ... }
4        ...
5  };
6  int main() {
7        Obj o;
8        const Foo& val = o.getFoo();
9  }
```

Reference lifetime extension

15.2 Temporary objects [class.temporary]

4 When an implementation introduces a temporary object of a class that has a non-trivial constructor (15.1, 15.8), it shall ensure that a constructor is called for the temporary object. Similarly, the destructor shall be called for a temporary with a non-trivial destructor (15.4). Temporary objects are destroyed as the last step in evaluating the full-expression (4.6) that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception. The value computations and side effects of destroying a temporary object are associated only with the full-expression, not with any specific subexpression.

```
1  using namespace std;
2
3  struct Bar {
4      Bar(int i) : m_i(i) {cout << "Bar(" << m_i << ")" << endl; }
5      ~Bar() { cout << "~Bar(" << m_i << ")" << endl; }
6      int m_i;
7  };
8
9  struct Foo {
10     Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
11     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
12     Bar getBar() { return Bar(m_i); }
13     int m_i;
14 };
```


https://godbolt.org/z/nvY6EbTjs

```
1  using namespace std;
2
3  struct Bar {
4      Bar(int i) : m_i(i) {cout << "Bar(" << m_i << ")" << endl; }
5      ~Bar() { cout << "~Bar(" << m_i << ")" << endl; }
6      int m_i;
7  };
8
9  struct Foo {
10     Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
11     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
12     Bar getBar() { return Bar(m_i); }
13     int m_i;
14 };
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1),
9                  Foo(2).m_i,
10                 Foo(3).getBar(),
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1),
9                  Foo(2).m_i,
10                 Foo(3).getBar(),
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1),
9                  Foo(2).m_i,
10                 Foo(3).getBar(),
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1),
9                  Foo(2).m_i,
10                 Foo(3).getBar(),
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1),
9                  Foo(2).m_i,
10                 Foo(3).getBar(),
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1),
9                  Foo(2).m_i,
10                 Foo(3).getBar(),
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```

The order of evaluation of parameters is not defined. Each implementation can define their rule.

https://godbolt.org/z/nvY6EbTjs

```
 1 struct Foo;
 2 struct Bar;
 3
 4 void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
 5 void doSomethingElse();
 6
 7 int main() {
 8     doSomething(Foo(1),
 9                 Foo(2).m_i,
10                 Foo(3).getBar(),
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
 1 struct Foo;
 2 struct Bar;
 3
 4 void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
 5 void doSomethingElse();
 6
 7 int main() {
 8     doSomething(Foo(1), // Foo(1)
 9                 Foo(2).m_i,
10                 Foo(3).getBar(),
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1), // Foo(1)
9                  Foo(2).m_i,
10                 Foo(3).getBar(),
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1), // Foo(1)
9                  Foo(2).m_i, // Foo(2)
10                 Foo(3).getBar(),
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1), // Foo(1)
9                  Foo(2).m_i, // Foo(2)
10                 Foo(3).getBar(),
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1), // Foo(1)
9                  Foo(2).m_i, // Foo(2)
10                 Foo(3).getBar(), // Foo(3), Bar(3)
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1), // Foo(1)
9                  Foo(2).m_i, // Foo(2)
10                 Foo(3).getBar(), // Foo(3), Bar(3)
11                 string("World").c_str()),
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1), // Foo(1)
9                  Foo(2).m_i, // Foo(2)
10                 Foo(3).getBar(), // Foo(3), Bar(3)
11                 string("World").c_str()), // string
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1), // Foo(1)
9                  Foo(2).m_i, // Foo(2)
10                 Foo(3).getBar(), // Foo(3), Bar(3)
11                 string("World").c_str()), // string
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1), // Foo(1)
9                  Foo(2).m_i, // Foo(2)
10                 Foo(3).getBar(), // Foo(3), Bar(3)
11                 string("World").c_str()), // string
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1), // Foo(1)
9                  Foo(2).m_i, // Foo(2)
10                 Foo(3).getBar(), // Foo(3), Bar(3)
11                 string("World").c_str()), // string
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1), // Foo(1)
9                  Foo(2).m_i, // Foo(2)
10                 Foo(3).getBar(), // Foo(3), Bar(3)
11                 string("World").c_str()), // string
12     doSomethingElse();
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1  struct Foo;
2  struct Bar;
3
4  void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
5  void doSomethingElse();
6
7  int main() {
8      doSomething(Foo(1), // Foo(1)
9                  Foo(2).m_i, // Foo(2)
10                 Foo(3).getBar(), // Foo(3), Bar(3)
11                 string("World").c_str()), // string
12     doSomethingElse(); // ~string, ~Bar(3), ~Foo(3), ~Foo(2), ~Foo(1)
13 }
```


https://godbolt.org/z/nvY6EbTjs

```
1 const Foo& retTempRef(const Foo& f) { return f; }
2 void doSomething(const Foo& f);
3 int main() {
4    doSomething(
5       retTempRef(
6          Foo(1)
7       )
8    );
9 }
```


https://godbolt.org/z/nvY6EbTjs

```
1 const Foo& retTempRef(const Foo& f) { return f; }
2 void doSomething(const Foo& f);
3 int main() {
4    doSomething(
5       retTempRef(
6          Foo(1)
7       )
8    );
9 }
```


https://godbolt.org/z/nvY6EbTjs

```
1 const Foo& retTempRef(const Foo& f) { return f; }
2 void doSomething(const Foo& f);
3 int main() {
4    doSomething(
5       retTempRef(
6          Foo(1)
7       )
8    );
9 }
```


https://godbolt.org/z/nvY6EbTjs

```
1 const Foo& retTempRef(const Foo& f) { return f; }
2 void doSomething(const Foo& f);
3 int main() {
4    doSomething(
5       retTempRef(
6          Foo(1) // Foo(1)
7       )
8    );
9 }
```


https://godbolt.org/z/nvY6EbTjs

```
1 const Foo& retTempRef(const Foo& f) { return f; }
2 void doSomething(const Foo& f);
3 int main() {
4    doSomething(
5       retTempRef(
6          Foo(1) // Foo(1)
7       )
8    );
9 }
```


https://godbolt.org/z/nvY6EbTjs

```
1 const Foo& retTempRef(const Foo& f) { return f; }
2 void doSomething(const Foo& f);
3 int main() {
4     doSomething(
5         retTempRef(
6             Foo(1) // Foo(1)
7         )
8     );
9 }
```


https://godbolt.org/z/nvY6EbTjs

```
1 const Foo& retTempRef(const Foo& f) { return f; }
2 void doSomething(const Foo& f);
3 int main() {
4    doSomething(
5       retTempRef(
6          Foo(1) // Foo(1)
7       )
8    );
9 }
```


https://godbolt.org/z/nvY6EbTjs

```
1 const Foo& retTempRef(const Foo& f) { return f; }
2 void doSomething(const Foo& f);
3 int main() {
4    doSomething(
5       retTempRef(
6          Foo(1) // Foo(1)
7       )
8    ); // ~Foo(1)
9 }
```


https://godbolt.org/z/nvY6EbTjs

15.2 Temporary objects [class.temporary]

4 When an implementation introduces a temporary object of a class that has a non-trivial constructor (15.1, 15.8), it shall ensure that a constructor is called for the temporary object. Similarly, the destructor shall be called for a temporary with a non-trivial destructor (15.4). Temporary objects are destroyed as the last step in evaluating the full-expression (4.6) that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception. The value computations and side effects of destroying a temporary object are associated only with the full-expression, not with any specific subexpression.

15.2 Temporary objects [class.temporary]

5 There are three contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array with no corresponding initializer (11.6). The second context is when a copy constructor is called to copy an element of an array while the entire array is copied (8.1.5.2, 15.8). In either case, if the constructor has one or more default arguments, the destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any.

15.2 Temporary objects [class.temporary]

5 There are three contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array with no corresponding initializer (11.6). The second context is when a copy constructor is called to copy an element of an array while the entire array is copied (8.1.5.2, 15.8). In either case, if the constructor has one or more default arguments, the destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any.

6 The third context is when a reference is bound to a temporary.[116] The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except:

```
1 using namespace std;
2 struct Foo {
3     Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5     int i() { return m_i; }
6     int& iR() { return m_i; }
7
8     int m_i;
9 };
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
```


https://godbolt.org/z/rbK1sP8n9

```
1 using namespace std;
2 struct Foo {
3     Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4     ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5     int i() { return m_i; }
6     int& iR() { return m_i; }
7
8     int m_i;
9 };
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1);
13     const int& v2 = Foo(2).m_i;
14     const int& v3 = Foo(3).i();
15     const int& v4 = Foo(4).iR();
16 }
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1);
13     const int& v2 = Foo(2).m_i;
14     const int& v3 = Foo(3).i();
15     const int& v4 = Foo(4).iR();
16 }
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1); // Foo(1)
13     const int& v2 = Foo(2).m_i;
14     const int& v3 = Foo(3).i();
15     const int& v4 = Foo(4).iR();
16 }
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1); // Foo(1)
13     const int& v2 = Foo(2).m_i;
14     const int& v3 = Foo(3).i();
15     const int& v4 = Foo(4).iR();
16 }
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1); // Foo(1)
13     const int& v2 = Foo(2).m_i; // Foo(2)
14     const int& v3 = Foo(3).i();
15     const int& v4 = Foo(4).iR();
16 }
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1); // Foo(1)
13     const int& v2 = Foo(2).m_i; // Foo(2)
14     const int& v3 = Foo(3).i();
15     const int& v4 = Foo(4).iR();
16 }
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1); // Foo(1)
13     const int& v2 = Foo(2).m_i; // Foo(2)
14     const int& v3 = Foo(3).i(); // Foo(3), ~Foo(3)
15     const int& v4 = Foo(4).iR();
16 }
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1); // Foo(1)
13     const int& v2 = Foo(2).m_i; // Foo(2)
14     const int& v3 = Foo(3).i(); // Foo(3), ~Foo(3) <<< 'int' has its lifetime ext.
15     const int& v4 = Foo(4).iR();
16 }
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1); // Foo(1)
13     const int& v2 = Foo(2).m_i; // Foo(2)
14     const int& v3 = Foo(3).i(); // Foo(3), ~Foo(3) <<< 'int' has its lifetime ext.
15     const int& v4 = Foo(4).iR();
16 }
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1); // Foo(1)
13     const int& v2 = Foo(2).m_i; // Foo(2)
14     const int& v3 = Foo(3).i(); // Foo(3), ~Foo(3) <<< 'int' has its lifetime ext.
15     const int& v4 = Foo(4).iR(); // Foo(4), ~Foo(4)
16 }
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1); // Foo(1)
13     const int& v2 = Foo(2).m_i; // Foo(2)
14     const int& v3 = Foo(3).i(); // Foo(3), ~Foo(3) <<< 'int' has its lifetime ext.
15     const int& v4 = Foo(4).iR(); // Foo(4), ~Foo(4) <<< Dangling reference
16 }
```


https://godbolt.org/z/rbK1sP8n9

```
1  using namespace std;
2  struct Foo {
3      Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
4      ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
5      int i() { return m_i; }
6      int& iR() { return m_i; }
7
8      int m_i;
9  };
10
11 int main() {
12     const Foo& v1 = Foo(1); // Foo(1)
13     const int& v2 = Foo(2).m_i; // Foo(2)
14     const int& v3 = Foo(3).i(); // Foo(3), ~Foo(3) <<< 'int' has its lifetime ext.
15     const int& v4 = Foo(4).iR(); // Foo(4), ~Foo(4) <<< Dangling reference
16 } // '~int', ~Foo(2), ~Foo(1)
```


https://godbolt.org/z/rbK1sP8n9

15.2 Temporary objects [class.temporary]

5 There are three contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array with no corresponding initializer (11.6). The second context is when a copy constructor is called to copy an element of an array while the entire array is copied (8.1.5.2, 15.8). In either case, if the constructor has one or more default arguments, the destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any.

6 The third context is when a reference is bound to a temporary.[116] The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except:

15.2 Temporary objects [class.temporary]

5 There are three contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array with no corresponding initializer (11.6). The second context is when a copy constructor is called to copy an element of an array while the entire array is copied (8.1.5.2, 15.8). In either case, if the constructor has one or more default arguments, the destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any.

6 The third context is when a reference is bound to a temporary.[116] The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except:

15.2 Temporary objects [class.temporary]

5 There are three contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array with no corresponding initializer (11.6). The second context is when a copy constructor is called to copy an element of an array while the entire array is copied (8.1.5.2, 15.8). In either case, if the constructor has one or more default arguments, the destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any.

6 The third context is when a reference is bound to a temporary.[116] The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except:

(6.1) — A temporary object bound to a reference parameter in a function call (8.2.2) persists until the completion of the full-expression containing the call.

```
const Foo& retTempRef(const Foo& f) { return f; }
void doSomething(const Foo& f);
int main() {
    doSomething(retTempRef(Foo(1)));
}
```

15.2 Temporary objects [class.temporary]

5 There are three contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array with no corresponding initializer (11.6). The second context is when a copy constructor is called to copy an element of an array while the entire array is copied (8.1.5.2, 15.8). In either case, if the constructor has one or more default arguments, the destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any.

6 The third context is when a reference is bound to a temporary.[116] The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except:

(6.1) — A temporary object bound to a reference parameter in a function call (8.2.2) persists until the completion of the full-expression containing the call.

(6.2) — The lifetime of a temporary bound to the returned value in a function return statement (9.6.3) is not extended; the temporary is destroyed at the end of the full-expression in the return statement.

```
const Foo& getFoo() { return Foo(); }
```

5 There are three contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array with no corresponding initializer (11.6). The second context is when a copy constructor is called to copy an element of an array while the entire array is copied (8.1.5.2, 15.8). In either case, if the constructor has one or more default arguments, the destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any.

6 The third context is when a reference is bound to a temporary.[116] The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except:

(6.1) — A temporary object bound to a reference parameter in a function call (8.2.2) persists until the completion of the full-expression containing the call.

(6.2) — The lifetime of a temporary bound to the returned value in a function return statement (9.6.3) is not extended; the temporary is destroyed at the end of the full-expression in the return statement.

(6.3) — A temporary bound to a reference in a *new-initializer* (8.3.4) persists until the completion of the full-expression containing the *new-initializer*. [*Example:*

```
struct S { int mi; const std::pair<int,int>& mp; };
S a { 1, {2,3} };
S* p = new S{ 1, {2,3} };    // Creates dangling reference
```

— *end example*] [*Note:* This may introduce a dangling reference, and implementations are encouraged to issue a warning in such a case. — *end note*]

And when you least expect...

```
1  using namespace std;
2  std::vector<int> getValues() { return {1, 2, 3}; }
3
4
5
6
7
8  int main() {
9      const vector<int> vec = getValues();
10     for (const auto& v : vec) cout << v << endl;
11
12
13
14
15
16
17 }
```


https://godbolt.org/z/rcTKEh8KE

```cpp
using namespace std;
std::vector<int> getValues() { return {1, 2, 3}; }



int main() {
    const vector<int> vec = getValues();
    for (const auto& v : vec) cout << v << endl;


}
```


https://godbolt.org/z/rcTKEh8KE

```
1  using namespace std;
2  std::vector<int> getValues() { return {1, 2, 3}; }
3
4
5
6
7
8  int main() {
9      const vector<int> vec = getValues();
10     for (const auto& v : vec) cout << v << endl;
11
12
13
14
15
16
17 }
```


https://godbolt.org/z/rcTKEh8KE

```cpp
using namespace std;
std::vector<int> getValues() { return {1, 2, 3}; }



int main() {
    const vector<int> vec = getValues();
    for (const auto& v : vec) cout << v << endl;


}
```


https://godbolt.org/z/rcTKEh8KE

```
1  using namespace std;
2  std::vector<int> getValues() { return {1, 2, 3}; }
3
4
5
6
7
8  int main() {
9      const vector<int> vec = getValues();
10     for (const auto& v : vec) cout << v << endl;
11
12
13
14
15
16
17 }
```


https://godbolt.org/z/rcTKEh8KE

```
1  using namespace std;
2  std::vector<int> getValues() { return {1, 2, 3}; }
3
4
5
6
7
8  int main() {
9      const vector<int> vec = getValues();
10     for (const auto& v : vec) cout << v << endl; // 1, 2, 3
11
12
13
14
15
16
17 }
```


https://godbolt.org/z/rcTKEh8KE

```cpp
using namespace std;
std::vector<int> getValues() { return {1, 2, 3}; }



int main() {
    const vector<int> vec = getValues();
    for (const auto& v : vec) cout << v << endl; // 1, 2, 3
    for (const auto& v : getValues()) cout << v << endl; // 1, 2, 3


}
```


https://godbolt.org/z/rcTKEh8KE

```cpp
using namespace std;
std::vector<int> getValues() { return {1, 2, 3}; }
struct Values {
    vector<int> m_vec {1, 2, 3};
    const vector<int>& get() { return m_vec; }
};

int main() {
    const vector<int> vec = getValues();
    for (const auto& v : vec) cout << v << endl; // 1, 2, 3
    for (const auto& v : getValues()) cout << v << endl; // 1, 2, 3


}
```


https://godbolt.org/z/rcTKEh8KE

```cpp
using namespace std;
std::vector<int> getValues() { return {1, 2, 3}; }
struct Values {
    vector<int> m_vec {1, 2, 3};
    const vector<int>& get() { return m_vec; }
};

int main() {
    const vector<int> vec = getValues();
    for (const auto& v : vec) cout << v << endl; // 1, 2, 3
    for (const auto& v : getValues()) cout << v << endl; // 1, 2, 3

    Values values;
    for (const auto& v : values.get()) cout << v << endl;


}
```


https://godbolt.org/z/rcTKEh8KE

```cpp
using namespace std;
std::vector<int> getValues() { return {1, 2, 3}; }
struct Values {
    vector<int> m_vec {1, 2, 3};
    const vector<int>& get() { return m_vec; }
};

int main() {
    const vector<int> vec = getValues();
    for (const auto& v : vec) cout << v << endl; // 1, 2, 3
    for (const auto& v : getValues()) cout << v << endl; // 1, 2, 3

    Values values;
    for (const auto& v : values.get()) cout << v << endl;

}
```


https://godbolt.org/z/rcTKEh8KE

```cpp
using namespace std;
std::vector<int> getValues() { return {1, 2, 3}; }
struct Values {
    vector<int> m_vec {1, 2, 3};
    const vector<int>& get() { return m_vec; }
};

int main() {
    const vector<int> vec = getValues();
    for (const auto& v : vec) cout << v << endl; // 1, 2, 3
    for (const auto& v : getValues()) cout << v << endl; // 1, 2, 3

    Values values;
    for (const auto& v : values.get()) cout << v << endl; // 1, 2, 3

}
```


https://godbolt.org/z/rcTKEh8KE

```
1  using namespace std;
2  std::vector<int> getValues() { return {1, 2, 3}; }
3  struct Values {
4      vector<int> m_vec {1, 2, 3};
5      const vector<int>& get() { return m_vec; }
6  };
7
8  int main() {
9      const vector<int> vec = getValues();
10     for (const auto& v : vec) cout << v << endl; // 1, 2, 3
11     for (const auto& v : getValues()) cout << v << endl; // 1, 2, 3
12
13     Values values;
14     for (const auto& v : values.get()) cout << v << endl; // 1, 2, 3
15     for (const auto& v : Values().m_vec) cout << v << endl;
16
17 }
```


https://godbolt.org/z/rcTKEh8KE

```cpp
using namespace std;
std::vector<int> getValues() { return {1, 2, 3}; }
struct Values {
    vector<int> m_vec {1, 2, 3};
    const vector<int>& get() { return m_vec; }
};

int main() {
    const vector<int> vec = getValues();
    for (const auto& v : vec) cout << v << endl; // 1, 2, 3
    for (const auto& v : getValues()) cout << v << endl; // 1, 2, 3

    Values values;
    for (const auto& v : values.get()) cout << v << endl; // 1, 2, 3
    for (const auto& v : Values().m_vec) cout << v << endl; // 1, 2, 3

}
```


https://godbolt.org/z/rcTKEh8KE

```cpp
using namespace std;
std::vector<int> getValues() { return {1, 2, 3}; }
struct Values {
    vector<int> m_vec {1, 2, 3};
    const vector<int>& get() { return m_vec; }
};

int main() {
    const vector<int> vec = getValues();
    for (const auto& v : vec) cout << v << endl; // 1, 2, 3
    for (const auto& v : getValues()) cout << v << endl; // 1, 2, 3

    Values values;
    for (const auto& v : values.get()) cout << v << endl; // 1, 2, 3
    for (const auto& v : Values().m_vec) cout << v << endl; // 1, 2, 3
    for (const auto& v : Values().get()) cout << v << endl;
}
```


https://godbolt.org/z/rcTKEh8KE

```cpp
using namespace std;
std::vector<int> getValues() { return {1, 2, 3}; }
struct Values {
    vector<int> m_vec {1, 2, 3};
    const vector<int>& get() { return m_vec; }
};

int main() {
    const vector<int> vec = getValues();
    for (const auto& v : vec) cout << v << endl; // 1, 2, 3
    for (const auto& v : getValues()) cout << v << endl; // 1, 2, 3

    Values values;
    for (const auto& v : values.get()) cout << v << endl; // 1, 2, 3
    for (const auto& v : Values().m_vec) cout << v << endl; // 1, 2, 3
    for (const auto& v : Values().get()) cout << v << endl; // UB
}
```


https://godbolt.org/z/rcTKEh8KE

```
1 { // for (const auto& v : Values().m_vec)
2
3
4
5
6
7
8
9 }
```


https://cppinsights.io/s/4e9696e1

```
1 { // for (const auto& v : Values().m_vec)
2     vector<int, allocator<int>> && __range1 = Values().m_vec;
3     auto __begin1 = __range1.begin();
4     auto __end1 = __range1.end();
5     for(; operator!=(__begin1, __end1); __begin1.operator++()) {
6         int const & v = __begin1.operator*();
7         cout.operator<<(v).operator<<(endl);
8     }
9 }
```


https://cppinsights.io/s/4e9696e1

```
1 { // for (const auto& v : Values().m_vec)
2     vector<int, allocator<int>> && __range1 = Values().m_vec;
3     auto __begin1 = __range1.begin();
4     auto __end1 = __range1.end();
5     for(; operator!=(__begin1, __end1); __begin1.operator++()) {
6         int const & v = __begin1.operator*();
7         cout.operator<<(v).operator<<(endl);
8     }
9 }
```


https://cppinsights.io/s/4e9696e1

```
1 { // for (const auto& v : Values().m_vec)
2     vector<int, allocator<int>> && __range1 = Values().m_vec;
3     auto __begin1 = __range1.begin();
4     auto __end1 = __range1.end();
5     for(; operator!=(__begin1, __end1); __begin1.operator++()) {
6         int const & v = __begin1.operator*();
7         cout.operator<<(v).operator<<(endl);
8     }
9 }
```


https://cppinsights.io/s/4e9696e1

```
1 { // for (const auto& v : Values().m_vec)
2     vector<int, allocator<int>> && __range1 = Values().m_vec;
3     auto __begin1 = __range1.begin();
4     auto __end1 = __range1.end();
5     for(; operator!=(__begin1, __end1); __begin1.operator++()) {
6         int const & v = __begin1.operator*();
7         cout.operator<<(v).operator<<(endl);
8     }
9 }
```


https://cppinsights.io/s/4e9696e1

```
1 { // for (const auto& v : Values().m_vec)
2     vector<int, allocator<int>> && __range1 = Values().m_vec;
3     auto __begin1 = __range1.begin();
4     auto __end1 = __range1.end();
5     for(; operator!=(__begin1, __end1); __begin1.operator++()) {
6         int const & v = __begin1.operator*();
7         cout.operator<<(v).operator<<(endl);
8     }
9 }
```

https://cppinsights.io/s/4e9696e1

```
1 { // for (const auto& v : Values().m_vec)
2     vector<int, allocator<int>> && __range1 = Values().m_vec;
3     auto __begin1 = __range1.begin();
4     auto __end1 = __range1.end();
5     for(; operator!=(__begin1, __end1); __begin1.operator++()) {
6         int const & v = __begin1.operator*();
7         cout.operator<<(v).operator<<(endl);
8     }
9 }
10
11
12
13
14
15
16
17
18
```


https://cppinsights.io/s/4e9696e1

```
1  { // for (const auto& v : Values().m_vec)
2      vector<int, allocator<int>> && __range1 = Values().m_vec;
3      auto __begin1 = __range1.begin();
4      auto __end1 = __range1.end();
5      for(; operator!=(__begin1, __end1); __begin1.operator++()) {
6          int const & v = __begin1.operator*();
7          cout.operator<<(v).operator<<(endl);
8      }
9  }
10 { // for (const auto& v : Values().get())
11
12                                  const vector<int>& get() { return m_vec; }
13
14
15
16
17     -
18 }
```


https://cppinsights.io/s/4e9696e1

```
1  { // for (const auto& v : Values().m_vec)
2      vector<int, allocator<int>> && __range1 = Values().m_vec;
3      auto __begin1 = __range1.begin();
4      auto __end1 = __range1.end();
5      for(; operator!=(__begin1, __end1); __begin1.operator++()) {
6          int const & v = __begin1.operator*();
7          cout.operator<<(v).operator<<(endl);
8      }
9  }
10 { // for (const auto& v : Values().get())
11     const vector<int, allocator<int>> & __range1 = Values().get();
12     auto __begin1 = __range1.begin();
13     auto __end1 = __range1.end();
14     for(; operator!=(__begin1, __end1); __begin1.operator++()) {
15         int const & v = __begin1.operator*();
16         cout.operator<<(v).operator<<(endl);
17     }
18 }
```


https://cppinsights.io/s/4e9696e1

```
1  { // for (const auto& v : Values().m_vec)
2      vector<int, allocator<int>> && __range1 = Values().m_vec;
3      auto __begin1 = __range1.begin();
4      auto __end1 = __range1.end();
5      for(; operator!=(__begin1, __end1); __begin1.operator++()) {
6          int const & v = __begin1.operator*();
7          cout.operator<<(v).operator<<(endl);
8      }
9  }
10 { // for (const auto& v : Values().get())
11     const vector<int, allocator<int>> & __range1 = Values().get();
12     auto __begin1 = __range1.begin();
13     auto __end1 = __range1.end();
14     for(; operator!=(__begin1, __end1); __begin1.operator++()) {
15         int const & v = __begin1.operator*();
16         cout.operator<<(v).operator<<(endl);
17     }
18 }
```


https://cppinsights.io/s/4e9696e1

```
1  { // for (const auto& v : Values().m_vec)
2      vector<int, allocator<int>> && __range1 = Values().m_vec;
3      auto __begin1 = __range1.begin();
4      auto __end1 = __range1.end();
5      for(; operator!=(__begin1, __end1); __begin1.operator++()) {
6          int const & v = __begin1.operator*();
7          cout.operator<<(v).operator<<(endl);
8      }
9  }
10 { // for (const auto& v : Values().get())
11     const vector<int, allocator<int>> & __range1 = Values().get();
12     auto __begin1 = __range1.begin();
13     auto __end1 = __range1.end();
14     for(; operator!=(__begin1, __end1); __begin1.operator++()) {
15         int const & v = __begin1.operator*();
16         cout.operator<<(v).operator<<(endl);
17     }
18 }
```

https://cppinsights.io/s/4e9696e1

Wording for P2644R1 Fix for Range-based for Loop

C++23

Document#: P2718R0 **Date:** 2022-11-11

Project: ISO JTC1/SC22/WG21

Reply-to: Nicolai Josuttis <nico@josuttis.de>
Joshua Berne <jberne4@bloomberg.net>

In subclause 6.7.7 [class.temporary], modify p5:

> There are ~~three~~four contexts in which temporaries are destroyed at a different point than the end of the full-expression. ...

In subclause 6.7.7 [class.temporary], add a paragraph after p6

> The fourth context is when a temporary object other than a function parameter object is created in the *for-range-initializer* of a range-based for statement. If such a temporary object would otherwise be destroyed at the end of the *for-range-initializer* full-expression, the object persists for the lifetime of the reference initialized by the *for-range-initializer*.

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2718r0.html

```
int main() {
    if (doSomething(Foo().getBar())) {
        // ...
    }
}
```

```cpp
std::mutex mtx;
struct Bar {};
struct Foo {
    Foo() : m_lock(mtx) {}
    Bar getBar() { return Bar(); }
    std::lock_guard<std::mutex> m_lock;
};

bool doSomething(const Bar& v) {
    std::lock_guard<std::mutex> lock(mtx);
    // ...
}

int main() {
    if (doSomething(Foo().getBar())) {
        // ...
    }
}
```

```cpp
std::mutex mtx;
struct Bar {};
struct Foo {
    Foo() : m_lock(mtx) {}
    Bar getBar() { return Bar(); }
    std::lock_guard<std::mutex> m_lock;
};

bool doSomething(const Bar& v) {
    std::lock_guard<std::mutex> lock(mtx);
    // ...
}

int main() {
    if (doSomething(Foo().getBar())) {
        // ...
    }
}
```

```cpp
std::mutex mtx;
struct Bar {};
struct Foo {
    Foo() : m_lock(mtx) {}
    Bar getBar() { return Bar(); }
    std::lock_guard<std::mutex> m_lock;
};

bool doSomething(const Bar& v) {
    std::lock_guard<std::mutex> lock(mtx);
    // ...
}

int main() {
    if (doSomething(Foo().getBar())) {
        // ...
    }
}
```

```cpp
std::mutex mtx;
struct Bar {};
struct Foo {
    Foo() : m_lock(mtx) {}
    Bar getBar() { return Bar(); }
    std::lock_guard<std::mutex> m_lock;
};

bool doSomething(const Bar& v) {
    std::lock_guard<std::mutex> lock(mtx);
    // ...
}

int main() {
    if (doSomething(Foo().getBar())) {
        // ...
    }
}
```

```cpp
std::mutex mtx;
struct Bar {};
struct Foo {
    Foo() : m_lock(mtx) {}
    Bar getBar() { return Bar(); }
    std::lock_guard<std::mutex> m_lock;
};

bool doSomething(const Bar& v) {
    std::lock_guard<std::mutex> lock(mtx);
    // ...
}

int main() {
    if (doSomething(Foo().getBar())) {
        // ...
    }
}
```

```cpp
std::mutex mtx;
struct Bar {};
struct Foo {
    Foo() : m_lock(mtx) {}
    Bar getBar() { return Bar(); }
    std::lock_guard<std::mutex> m_lock;
};

bool doSomething(const Bar& v) {
    std::lock_guard<std::mutex> lock(mtx);
    // ...
}

int main() {
    if (doSomething(Foo().getBar())) {
        // ...
    }
}
```

```cpp
std::mutex mtx;
struct Bar {};
struct Foo {
    Foo() : m_lock(mtx) {}
    Bar getBar() { return Bar(); }
    std::lock_guard<std::mutex> m_lock;
};

bool doSomething(const Bar& v) {
    std::lock_guard<std::mutex> lock(mtx);
    // ...
}

int main() {
    if (doSomething(Foo().getBar())) {
        // ...
    }
}
```

```cpp
std::mutex mtx;
struct Bar {};
struct Foo {
    Foo() : m_lock(mtx) {}
    Bar getBar() { return Bar(); }
    std::lock_guard<std::mutex> m_lock;
};

bool doSomething(const Bar& v) {
    std::lock_guard<std::mutex> lock(mtx);
    // ...
}

int main() {
    if (doSomething(Foo().getBar())) {
        // ...
    }
}
```

15.2 Temporary objects [class.temporary]

4 When an implementation introduces a temporary object of a class that has a non-trivial constructor (15.1, 15.8), it shall ensure that a constructor is called for the temporary object. Similarly, the destructor shall be called for a temporary with a non-trivial destructor (15.4). Temporary objects are destroyed as the last step in evaluating the full-expression (4.6) that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception. The value computations and side effects of destroying a temporary object are associated only with the full-expression, not with any specific subexpression.

Review

- What’s object lifetime, when it begins and when it ends
- Dangling references
- Lifetime extension
- Lifetime of temporaries and its exceptions and pitfalls

Thank you!