CppNorth 2023

Object Lifetime

From Start To Finish

Thamara Andrade

Object Lifetime

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

Thamara Andrade | https://thamara.dev/

On this talk…

You can expect…	but not…
• Review of object lifetime	• Value categories
• (A Little of) RAII	• Unions/Arrays
• Beyond basic lifetime	• Any assembly code
• Common pitfalls	

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

FAQs

Q: Why is the standard hard to read? I'm having trouble learning C++ from reading it.

The standard is not intended to teach how to use C++. Rather, it is an international treaty – a formal, legal, and sometimes mind-numbingly detailed technical document intended primarily for people writing C++ compilers and standard library implementations.

(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).

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

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

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

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

Vacuous initialization

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

struct EmptyPoint {};
EmptyPoint ep;

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

Constructors & Destructors

Constructors & Destructors

```
struct Foo {
  Foo() { std::cout << "Foo()" << std::endl; }
  ~Foo() { std::cout << "~Foo()" << std::endl; }
};

int main() {
  Foo a;
  {
    Foo b;
  }
}
```

Constructors & Destructors

```cpp
struct Foo {
  Foo() { std::cout << "Foo()" << std::endl; }
  ~Foo() { std::cout << "~Foo()" << std::endl; }
};

int main() {
  Foo a;
  {
    Foo b;
  }
}
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
};

int main() {
    Foo a;
    {
        Foo b;
    }
}
```

https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
};

int main() {
    Foo a; // Foo()
    {
        Foo b;
    }
}
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
};

int main() {
    Foo a; // Foo()
    {
        Foo b;
    }
}
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
};

int main() {
    Foo a; // Foo()
    {
        Foo b;
    }
}
```

https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
};

int main() {
    Foo a; // Foo()
    {
        Foo b; // Foo()
    }
}
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```cpp
struct Foo {
   Foo() { std::cout << "Foo()" << std::endl; }
   ~Foo() { std::cout << "~Foo()" << std::endl; }
};

int main() {
   Foo a; // Foo()
   {
      Foo b; // Foo()
   }
}
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```cpp
struct Foo {
  Foo() { std::cout << "Foo()" << std::endl; }
  ~Foo() { std::cout << "~Foo()" << std::endl; }
};

int main() {
  Foo a; // Foo()
  {
    Foo b; // Foo()
  } // ~Foo() of b
}
```

https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```cpp
struct Foo {
  Foo() { std::cout << "Foo()" << std::endl; }
  ~Foo() { std::cout << "~Foo()" << std::endl; }
};

int main() {
  Foo a; // Foo()
  {
    Foo b; // Foo()
  } // ~Foo() of b
}
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors

```cpp
struct Foo {
  Foo() { std::cout << "Foo()" << std::endl; }
  ~Foo() { std::cout << "~Foo()" << std::endl; }
};

int main() {
  Foo a; // Foo()
  {
    Foo b; // Foo()
  } // ~Foo() of b
} // ~Foo() of a
```


https://godbolt.org/z/ffzG8hPT8

Constructors & Destructors (2)

Constructors & Destructors (2)

```
struct Foo {
   Foo() { std::cout << "Foo()" << std::endl; }
   ~Foo() { std::cout << "~Foo()" << std::endl; }
};
int main() {
   Foo* a = nullptr;
   {
      a = new Foo();
   }
   delete a;
}
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
};
int main() {
    Foo* a = nullptr;
    {
        a = new Foo();
    }
    delete a;
}
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```
struct Foo {
   Foo() { std::cout << "Foo()" << std::endl; }
   ~Foo() { std::cout << "~Foo()" << std::endl; }
};
int main() {
   Foo* a = nullptr;
   {
      a = new Foo();
   }
   delete a;
}
```

https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```cpp
struct Foo {
  Foo() { std::cout << "Foo()" << std::endl; }
  ~Foo() { std::cout << "~Foo()" << std::endl; }
};
int main() {
  Foo* a = nullptr;
  {
    a = new Foo();
  }
  delete a;
}
```

https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```
struct Foo {
  Foo() { std::cout << "Foo()" << std::endl; }
  ~Foo() { std::cout << "~Foo()" << std::endl; }
};
int main() {
  Foo* a = nullptr;
  {
    a = new Foo(); // Foo()
  }
  delete a;
}
```

https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```cpp
struct Foo {
   Foo() { std::cout << "Foo()" << std::endl; }
   ~Foo() { std::cout << "~Foo()" << std::endl; }
};
int main() {
   Foo* a = nullptr;
   {
      a = new Foo(); // Foo()
   }
   delete a;
}
```

https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
};
int main() {
    Foo* a = nullptr;
    {
        a = new Foo(); // Foo()
    } // Foo a still in memory/alive
    delete a;
}
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```cpp
struct Foo {
  Foo() { std::cout << "Foo()" << std::endl; }
  ~Foo() { std::cout << "~Foo()" << std::endl; }
};
int main() {
  Foo* a = nullptr;
  {
    a = new Foo(); // Foo()
  } // Foo a still in memory/alive
  delete a;
}
```


https://godbolt.org/z/hY7hr5Efr

Constructors & Destructors (2)

```cpp
struct Foo {
  Foo() { std::cout << "Foo()" << std::endl; }
  ~Foo() { std::cout << "~Foo()" << std::endl; }
};
int main() {
  Foo* a = nullptr;
  {
    a = new Foo(); // Foo()
  } // Foo a still in memory/alive
  delete a; // ~Foo()
}
```

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).

Incomplete initialization

```cpp
struct Obj {
    Obj() {
        ptr = new int[1000];
        throw std::runtime_error("Exception in constructor");
    }

    ~Obj() { delete[] ptr; }
  private:
    int* ptr;
};

int main() {
    try {
        Obj obj;
    } catch (const std::exception& e) {}
}
```

https://godbolt.org/z/P9rb9j69v

Incomplete initialization

```cpp
struct Obj {
    Obj() {
        ptr = new int[1000];
        throw std::runtime_error("Exception in constructor");
    }

    ~Obj() { delete[] ptr; }
  private:
    int* ptr;
};

int main() {
    try {
        Obj obj;
    } catch (const std::exception& e) {}
}
```

https://godbolt.org/z/P9rb9j69v

Incomplete initialization

```cpp
struct Obj {
    Obj() {
        ptr = new int[1000];
        throw std::runtime_error("Exception in constructor");
    }

    ~Obj() { delete[] ptr; }
  private:
    int* ptr;
};

int main() {
    try {
        Obj obj;
    } catch (const std::exception& e) {}
}
```

https://godbolt.org/z/P9rb9j69v

Incomplete initialization

```cpp
struct Obj {
    Obj() {
        ptr = new int[1000];
        throw std::runtime_error("Exception in constructor");
    }

    ~Obj() { delete[] ptr; }
  private:
    int* ptr;
};

int main() {
    try {
        Obj obj;
    } catch (const std::exception& e) {}
}
```

https://godbolt.org/z/P9rb9j69v

Incomplete initialization

```cpp
struct Obj {
    Obj() {
        ptr = new int[1000];
        throw std::runtime_error("Exception in constructor");
    }

    ~Obj() { delete[] ptr; }
  private:
    int* ptr;
};

int main() {
    try {
        Obj obj;
    } catch (const std::exception& e) {}
}
```


https://godbolt.org/z/P9rb9j69v

Incomplete initialization

```cpp
struct Obj {
    Obj() {
        ptr = new int[1000];
        throw std::runtime_error("Exception in constructor");
    }

    ~Obj() { delete[] ptr; }
  private:
    int* ptr;
};

int main() {
    try {
        Obj obj;
    } catch (const std::exception& e) {}
}
```

Obj is not fully initialized.
It’s lifetime doesn’t start.

https://godbolt.org/z/P9rb9j69v

Incomplete initialization

```
struct Obj {
    Obj() {
        ptr = new int[1000];
        throw std::runtime_error("Exception in constructor");
    }

    ~Obj() { delete[] ptr; }
  private:
    int* ptr;
};

int main() {
    try {
        Obj obj;
    } catch (const std::exception& e) {}
}
```

Obj is not fully initialized.
It’s lifetime doesn’t start.

https://godbolt.org/z/P9rb9j69v

Incomplete initialization

```
struct Obj {
    Obj() {
        ptr = new int[1000];
        throw std::runtime_error("Exception in constructor");
    }

    ~Obj() { delete[] ptr; }
  private:
    int* ptr;
};

int main() {
    try {
        Obj obj;
    } catch (const std::exception& e) {}
}
```

Obj is not fully initialized.
It’s lifetime doesn’t start.

https://godbolt.org/z/P9rb9j69v

Incomplete initialization

```cpp
struct Obj {
    Obj() {
        ptr = new int[1000];
        throw std::runtime_error("Exception in constructor");
    }

    ~Obj() { delete[] ptr; }
   private:
    int* ptr;
};

int main() {
    try {
        Obj obj;
    } catch (const std::exception& e) {}
}
```

Obj is not fully initialized.
It’s lifetime doesn’t start.

Delete is never called.

https://godbolt.org/z/P9rb9j69v

Incomplete initialization

```cpp
struct Obj {
    Obj() {
        ptr = new int[1000];
        throw std::runtime_error("Exception in constructor");
    }

    ~Obj() { delete[] ptr; }
  private:
    int* ptr;
};

int main() {
    try {
        Obj obj;
    } catch (const std
}
```

Obj is not fully initialized.
It's lifetime doesn't start.

Delete is never called.

```
=================================================================
==1==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 4000 byte(s) in 1 object(s) allocated from:
#0 0x55cfe6d703dd in operator new[](unsigned long) new_delete.cpp:98:3
#1 0x55cfe6d72a8d in Obj::Obj() /app/example.cpp:5:15
#2 0x55cfe6d7296b in main /app/example.cpp:19:13
#3 0x7f607d28c082 in __libc_start_main (libc.so.6+0x24082)


SUMMARY: AddressSanitizer: 4000 byte(s) leaked in 1 allocation(s).
```

https://godbolt.org/z/P9rb9j69v

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**.

```
struct DynamicArray {
    explicit DynamicArray(size_t sz) : m_d(new int[sz]) {}
    ~DynamicArray() { delete[] m_d; }
    // ...
    int* m_d;
};
int main() {
    DynamicArray arr(5);
    // Populate/work on array
    return 0;
}
```

More RAII usage

- Heap memory allocation
- Mutexes
- Threads
- File management

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

Object Lifetime

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

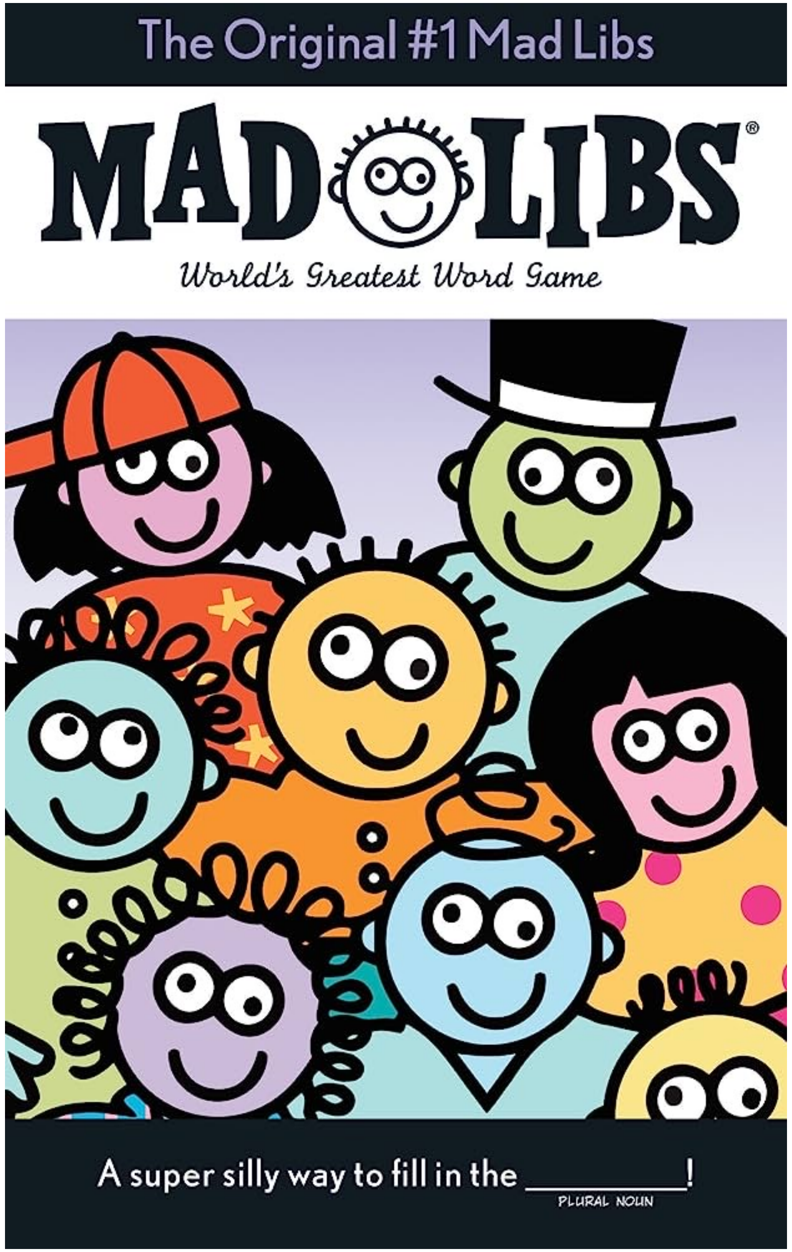
Thamara Andrade | https://thamara.dev/

Object Lifetime

From **Start** to **Finish**,
and the **tricky parts all around**

Thamara Andrade | https://thamara.dev/

tricky parts all around



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

```
struct Foo;
struct Obj {
    ____________ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    ____________ val = o.getFoo();
}
```

const Foo&   const Foo   Foo&   Foo

```
struct Foo;
struct Obj {
    ____________ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    ____________ val = o.getFoo();
}
```

const Foo&   const Foo   Foo&   Foo

```
struct Foo;
struct Obj {
    ____________ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    ____________ val = o.getFoo();
}
```


https://abseil.io/tips/101

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

Return
Initialize

1/9

```
struct Foo;
struct Obj {
    ____________ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    ____________ val = o.getFoo();
}
```


https://abseil.io/tips/101

const Foo& | const Foo | Foo& | Foo

Return
Initialize

1/9

```
struct Foo;
struct Obj {
    ____Foo____ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    ____Foo____ val = o.getFoo();
}
```


https://abseil.io/tips/101

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

Return
Initialize

1/9

```
struct Foo;
struct Obj {
    Foo getFoo() { return Foo(); }
    ...
};
int main() {
    Obj o;
    Foo val = o.getFoo();
}
```


https://abseil.io/tips/101

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

Return Initialize

1/9

```
struct Foo;
struct Obj {
    ___Foo___ getFoo() { return Foo(); }
    ...
};
int main() {
    Obj o;
    ___Foo___ val = o.getFoo();
}
```

Temporary

https://abseil.io/tips/101

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

Return
Initialize

1/9

```
struct Foo;
struct Obj {
    ____Foo____ getFoo() { return Foo(); }
    ...
};
int main() {
    Obj o;
    ____Foo____ val = o.getFoo();
}
```

Temporary is initialized directly in `val`'s storage

https://abseil.io/tips/101

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

Return Initialize

1/9

```
struct Foo;
struct Obj {
    Foo getFoo() { return Foo(); }
    ...
};
int main() {
    Obj o;
    Foo val = o.getFoo();
}
```

Temporary is initialized directly in `val`'s storage

Return Value Optimization (RVO)

https://abseil.io/tips/101

const Foo&   const Foo   Foo&   Foo

Return
Initialize

1/9

```
struct Foo;
struct Obj {
        Foo       getFoo() { Foo f; ...;
                             return f; }
};
int main() {
        Obj o;
        Foo       val = o.getFoo();
}
```


https://abseil.io/tips/101

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

Return
Initialize

1/9

```
struct Foo;
struct Obj {
    ___Foo___ getFoo() { Foo f; ...;
                         return f; }
};
int main() {
    Obj o;
    ___Foo___ val = o.getFoo();
}
```

Temporary

https://abseil.io/tips/101

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

Return
Initialize

1/9

```
struct Foo;
struct Obj {
    Foo getFoo() { Foo f; ...;
                   return f; }
};
int main() {
    Obj o;
    Foo val = o.getFoo();
}
```

Temporary v is initialized directly in `val`'s storage

https://abseil.io/tips/101

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

Return
Initialize

1/9

```
struct Foo;
struct Obj {
    ___Foo___ getFoo() { Foo f; ...;
                         return f; }
};
int main() {
    Obj o;
    ___Foo___ val = o.getFoo();
}
```

Temporary v is initialized directly in `val`'s storage

Named Return Value Optimization (NRVO)

https://abseil.io/tips/101

NRVO

https://godbolt.org/z/6c8n89aeP

NRVO

```
Foo getFoo(Foo* vPtr) {
    Foo v;
    v.m_i = 3
    assert (vPtr == &v);
    return v;
}

int main() {
    Foo val = getFoo(&val);
}
```


https://godbolt.org/z/6c8n89aeP

NRVO

```
Foo getFoo(Foo* vPtr) {
    Foo v;
    v.m_i = 3
    assert (vPtr == &v);
    return v;
}

int main() {
    Foo val = getFoo(&val);
}
```


https://godbolt.org/z/6c8n89aeP

NRVO

```cpp
Foo getFoo(Foo* vPtr) {
    Foo v;
    v.m_i = 3
    assert (vPtr == &v); ✓
    return v;
}

int main() {
    Foo val = getFoo(&val);
}
```

```cpp
Foo getFoo(Foo* vPtr) {
    Foo v;
    if (cond) { v.m_i = 3; }
    assert(vPtr == &v); ✓
    return v;
}

int main() {
    Foo val = getFoo(&val);
}
```


https://godbolt.org/z/6c8n89aeP

NRVO

```
Foo getFoo(Foo* vPtr) {
    if (cond) { return Foo(); }
    Foo v;
    return v;
}

int main() {
    Foo val = getFoo(&val);
}
```

RVO & NRVO

https://godbolt.org/z/6c8n89aeP

NRVO

clang

```
Foo getFoo(Foo* vPtr) {
    if (cond) { return Foo(); }
    Foo v;
    return v;
}

int main() {
    Foo val = getFoo(&val);
}
```

RVO & NRVO

https://godbolt.org/z/6c8n89aeP

NRVO

clang

```cpp
Foo getFoo(Foo* vPtr) {
    if (cond) { return Foo(); }
    Foo v;
    return v;
}

int main() {
    Foo val = getFoo(&val);
}
```

RVO & NRVO

gcc

```cpp
Foo getFoo(Foo* vPtr) {
    if (cond) { return Foo(); }
    Foo v;
    return v;
}

int main() {
    Foo val = getFoo(&val);
}
```

Foo(&&)

https://godbolt.org/z/6c8n89aeP

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

1 When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the constructor selected for the copy/move operation and/or the destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object. If the first parameter of the selected constructor is an rvalue reference to the object's type, the destruction of that object occurs when the target would have been destroyed; otherwise, the destruction occurs at the later of the times when the two objects would have been destroyed without the optimization.[122] This elision of copy/move operations, called *copy elision*, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

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

1 When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the constructor selected for the copy/move operation and/or the destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object. If the first parameter of the selected constructor is an rvalue reference to the object's type, the destruction of that object occurs when the target would have been destroyed; otherwise, the destruction occurs at the later of the times when the two objects would have been destroyed without the optimization.[122] This elision of copy/move operations, called *copy elision*, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

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

Compilers may optimize a copy/move of a object in a function that returns by value if the type of the local object is the same as that returned and the object is what’s being returned.

would have been destroyed without the optimization.[122] This elision of copy/move operations, called *copy elision*, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

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

Return
Initialize

1/9

```
struct Foo;
struct Obj {
    Foo getFoo() { ... }
    ...
};
int main() {
    Obj o;
    Foo val = o.getFoo();
}
```

At most Foo(Foo&&)

https://abseil.io/tips/101

const Foo&	const Foo	Foo&	Foo
		Return	Initialize

2/9

```
struct Foo;
struct Obj {
    Foo& getFoo() { ... }
    ...
};
int main() {
    Obj o;
    Foo val = o.getFoo();
}
```


https://abseil.io/tips/101

const Foo& | const Foo | Foo& | Foo

Return | Initialize

2/9

```
struct Foo;
struct Obj {
    ____Foo&____ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    ____Foo_____ val = o.getFoo();
}
```

Foo(const Foo&)

https://abseil.io/tips/101

const Foo&	const Foo	Foo&	Foo
Return			Initialize

3/9

```
struct Foo;
struct Obj {
    _const Foo&_ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    ____Foo_____ val = o.getFoo();
}
```

Foo(const Foo&)

https://abseil.io/tips/101

const Foo& const Foo Foo& Foo

Return Initialize

3/9

```
struct Foo;
struct Obj {
    const Foo& getFoo() { ... }
    ...
};
int main() {
    Obj o;
    Foo val = o.getFoo();
}
```


https://abseil.io/tips/101

const Foo& | const Foo | Foo& | Foo

Return
Initialize

4/9

```
struct Foo;
struct Obj {
    _const Foo&_ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    _const Foo&_ val = o.getFoo();
}
```


https://abseil.io/tips/101

const Foo& | const Foo | Foo& | Foo

Return
Initialize

4/9

```
struct Foo;
struct Obj {
    const Foo& getFoo() { return m_foo; }
    ...
};
int main() {
    Obj o;
    const Foo& val = o.getFoo();
}
```


https://abseil.io/tips/101

const Foo& | const Foo | Foo& | Foo

Return
Initialize

4/9

```
struct Foo;
struct Obj {
    const Foo& getFoo() { return m_foo; }
    ...
};
int main() {
    Obj o;
    const Foo& val = o.getFoo();
}
```

Binding lifetime of val to Obj's m_foo

https://abseil.io/tips/101

```cpp
struct Foo {
  Foo() { std::cout << "Foo()" << std::endl; }
  ~Foo() { std::cout << "~Foo()" << std::endl; }
  int i {3};
};

struct Obj {
  const Foo& getFoo() { return m_foo; }
  Foo m_foo;
};
```

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
    int i {3};
};

struct Obj {
    const Foo& getFoo() { return m_foo; }
    Foo m_foo;
};
int main() {
    Obj* o = new Obj();
    const Foo& val = o->getFoo();
    std::cout << val.i << std::endl;
    delete o;
    std::cout << val.i << std::endl;
}
```

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
    int i {3};
};

struct Obj {
    const Foo& getFoo() { return m_foo; }
    Foo m_foo;
};
int main() {
    Obj* o = new Obj();
    const Foo& val = o->getFoo();
    std::cout << val.i << std::endl;
    delete o;
    std::cout << val.i << std::endl;
}
```

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
    int i {3};
};

struct Obj {
    const Foo& getFoo() { return m_foo; }
    Foo m_foo;
};
int main() {
    Obj* o = new Obj(); // Foo()
    const Foo& val = o->getFoo();
    std::cout << val.i << std::endl;
    delete o;
    std::cout << val.i << std::endl;
}
```

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
    int i {3};
};

struct Obj {
    const Foo& getFoo() { return m_foo; }
    Foo m_foo;
};
int main() {
    Obj* o = new Obj(); // Foo()
    const Foo& val = o->getFoo();
    std::cout << val.i << std::endl;
    delete o;
    std::cout << val.i << std::endl;
}
```

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
    int i {3};
};

struct Obj {
    const Foo& getFoo() { return m_foo; }
    Foo m_foo;
};
int main() {
    Obj* o = new Obj(); // Foo()
    const Foo& val = o->getFoo();
    std::cout << val.i << std::endl; // 3
    delete o;
    std::cout << val.i << std::endl;
}
```

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
    int i {3};
};

struct Obj {
    const Foo& getFoo() { return m_foo; }
    Foo m_foo;
};
int main() {
    Obj* o = new Obj(); // Foo()
    const Foo& val = o->getFoo();
    std::cout << val.i << std::endl; // 3
    delete o;
    std::cout << val.i << std::endl;
}
```

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
    int i {3};
};

struct Obj {
    const Foo& getFoo() { return m_foo; }
    Foo m_foo;
};
int main() {
    Obj* o = new Obj(); // Foo()
    const Foo& val = o->getFoo();
    std::cout << val.i << std::endl; // 3
    delete o; // ~Foo()
    std::cout << val.i << std::endl;
}
```

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
    int i {3};
};

struct Obj {
    const Foo& getFoo() { return m_foo; }
    Foo m_foo;
};
int main() {
    Obj* o = new Obj(); // Foo()
    const Foo& val = o->getFoo();
    std::cout << val.i << std::endl; // 3
    delete o; // ~Foo()
    std::cout << val.i << std::endl;
}
```

```cpp
struct Foo {
    Foo() { std::cout << "Foo()" << std::endl; }
    ~Foo() { std::cout << "~Foo()" << std::endl; }
    int i {3};
};

struct Obj {
    const Foo& getFoo() { return m_foo; }
    Foo m_foo;
};
int main() {
    Obj* o = new Obj(); // Foo()
    const Foo& val = o->getFoo();
    std::cout << val.i << std::endl; // 3
    delete o; // ~Foo()
    std::cout << val.i << std::endl; // UB
}
```

const Foo& | const Foo | Foo& | Foo

Return
Initialize

4/9

```
struct Foo;
struct Obj {
    const Foo& getFoo() { return m_foo; }
    ...
};
int main() {
    Obj o;
    const Foo& val = o.getFoo();
}
```

Binding lifetime of `val` to Obj's `m_foo`

https://abseil.io/tips/101

const Foo& | const Foo | Foo& | Foo

Initialize | Return

5/9

```
struct Foo;
struct Obj {
    ___Foo&___ getFoo() { return m_foo; }
    ...
};
int main() {
    Obj o;
    _const Foo&_ val = o.getFoo();
}
```

Binding lifetime of val to Obj's m_foo

https://abseil.io/tips/101

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

Return
Initialize

6/9

```
struct Foo;
struct Obj {
    Foo& getFoo() { return m_foo; }
    ...
};
int main() {
    Obj o;
    Foo& val = o.getFoo();
}
```

Binding lifetime of val to Obj's m_foo

https://abseil.io/tips/101

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

Return
Initialize

6/9

```
struct Foo;
struct Obj {
    ___Foo&___ getFoo() { return m_foo; }
    ...
};
int main() {
    Obj o;
    ___Foo&___ val = o.getFoo();
}
```

Binding lifetime of `val` to Obj's `m_foo`

+

Any modifications on `val` will reflect on `m_foo`

https://abseil.io/tips/101

const Foo&	const Foo	Foo&	Foo
		Return Initialize	

6/9

```
struct Foo;
struct Obj {
         Foo&         getFoo() { return m_foo
```

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

```
int main() {
    Obj o;
    ____Foo&____ val = o.getFoo();
}
```

Any modifications on `val` will reflect on `m_foo`

https://abseil.io/tips/101

const Foo&   const Foo   Foo&   Foo

```
struct Foo;
struct Obj {
    ____________ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    ____________ val = o.getFoo();
}
```


https://abseil.io/tips/101

const Foo& | const Foo | Foo& | Foo

Return | | Initialize |

7/9

```
struct Foo;
struct Obj {
    const Foo&  getFoo() { ... }
    ...
};
int main() {
    Obj o;
    Foo&  val = o.getFoo();
}
```


https://abseil.io/tips/101

const Foo&   const Foo   Foo&   Foo

Return   Initialize

7/9

```
struct Foo;
struct Obj {
     const Foo&   getFoo() { ... }
     ...
};
int main() {
     Obj o;
     Foo&         val = o.getFoo();
}
```

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

https://abseil.io/tips/101

const Foo&	const Foo	Foo&	Foo
		Initialize	Return

8/9

```
struct Foo;
struct Obj {
    Foo getFoo() { ... }
    ...
};
int main() {
    Obj o;
    Foo& val = o.getFoo();
}
```


https://abseil.io/tips/101

const Foo&   const Foo   Foo&   Foo

Initialize   Return

8/9

```
struct Foo;
struct Obj {
    ___Foo___ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    ___Foo&___ val = o.getFoo();
}
```

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

https://abseil.io/tips/101

const Foo& | const Foo | Foo& | Foo

Initialize

Return

9/9

```
struct Foo;
struct Obj {
    ___Foo___ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    _const Foo&_ val = o.getFoo();
}
```


https://abseil.io/tips/101

const Foo&	const Foo	Foo&	Foo

Initialize

Return

9/9

```
struct Foo;
struct Obj {
    ___Foo___ getFoo() { ... }
    ...
};
int main() {
    Obj o;
    _const Foo&_ val = o.getFoo();
}
```

Reference

Temporary

https://abseil.io/tips/101

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

Initialize | Return

9/9

```
struct Foo;
struct Obj {
    Foo getFoo() { ... }
    ...
};
int main() {
    Obj o;
    const Foo& val = o.getFoo();
}
```

Reference lifetime extension

Reference | Temporary

https://abseil.io/tips/101

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.

```cpp
using namespace std;

struct Bar {
    Bar(int i) : m_i(i) {cout << "Bar(" << m_i << ")" << endl; }
    ~Bar() { cout << "~Bar(" << m_i << ")" << endl; }
    int m_i;
};

struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    Bar getBar() { return Bar(m_i); }
    int m_i;
};
```


https://godbolt.org/z/nvY6EbTjs

```cpp
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1),
                Foo(2).m_i,
                Foo(3).getBar(),
                string("World").c_str()),
    doSomethingElse();
}
```


https://godbolt.org/z/nvY6EbTjs

```cpp
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1),
                Foo(2).m_i,
                Foo(3).getBar(),
                string("World").c_str()),
    doSomethingElse();
}
```


https://godbolt.org/z/nvY6EbTjs

```cpp
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1), // Foo(1)
                Foo(2).m_i,
                Foo(3).getBar(),
                string("World").c_str()),
    doSomethingElse();
}
```


https://godbolt.org/z/nvY6EbTjs

```
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1), // Foo(1)
                Foo(2).m_i,
                Foo(3).getBar(),
                string("World").c_str()),
    doSomethingElse();
}
```


https://godbolt.org/z/nvY6EbTjs

```cpp
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1), // Foo(1)
                Foo(2).m_i, // Foo(2)
                Foo(3).getBar(),
                string("World").c_str()),
    doSomethingElse();
}
```


https://godbolt.org/z/nvY6EbTjs

```cpp
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1), // Foo(1)
                Foo(2).m_i, // Foo(2)
                Foo(3).getBar(),
                string("World").c_str()),
    doSomethingElse();
}
```


https://godbolt.org/z/nvY6EbTjs

```cpp
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1), // Foo(1)
                Foo(2).m_i, // Foo(2)
                Foo(3).getBar(), // Foo(3), Bar(3)
                string("World").c_str()),
    doSomethingElse();
}
```


https://godbolt.org/z/nvY6EbTjs

```
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1), // Foo(1)
                Foo(2).m_i, // Foo(2)
                Foo(3).getBar(), // Foo(3), Bar(3)
                string("World").c_str()),
    doSomethingElse();
}
```


https://godbolt.org/z/nvY6EbTjs

```cpp
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1), // Foo(1)
                Foo(2).m_i, // Foo(2)
                Foo(3).getBar(), // Foo(3), Bar(3)
                string("World").c_str()), // string
    doSomethingElse();
}
```

https://godbolt.org/z/nvY6EbTjs

```cpp
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1), // Foo(1)
                Foo(2).m_i, // Foo(2)
                Foo(3).getBar(), // Foo(3), Bar(3)
                string("World").c_str()), // string
    doSomethingElse();
}
```

https://godbolt.org/z/nvY6EbTjs

```cpp
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1), // Foo(1)
                Foo(2).m_i, // Foo(2)
                Foo(3).getBar(), // Foo(3), Bar(3)
                string("World").c_str()), // string
    doSomethingElse();
}
```

https://godbolt.org/z/nvY6EbTjs

```cpp
struct Foo;
struct Bar;

void doSomething(const Foo& v1, int i, const Bar& v2, const char* s);
void doSomethingElse();

int main() {
    doSomething(Foo(1), // Foo(1)
                Foo(2).m_i, // Foo(2)
                Foo(3).getBar(), // Foo(3), Bar(3)
                string("World").c_str()), // string
    doSomethingElse(); // ~string, ~Bar(3), ~Foo(3), ~Foo(2), ~Foo(1)
}
```

https://godbolt.org/z/nvY6EbTjs

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


https://godbolt.org/z/nvY6EbTjs

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


https://godbolt.org/z/nvY6EbTjs

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


https://godbolt.org/z/nvY6EbTjs

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

https://godbolt.org/z/nvY6EbTjs

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

https://godbolt.org/z/nvY6EbTjs

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

https://godbolt.org/z/nvY6EbTjs

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


https://godbolt.org/z/nvY6EbTjs

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

https://godbolt.org/z/nvY6EbTjs

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

https://godbolt.org/z/nvY6EbTjs

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:

```
using namespace std;
struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    int i() { return m_i; }
    int& iR() { return m_i; }

    int m_i;
};
```

https://godbolt.org/z/rbK1sP8n9

```cpp
using namespace std;
struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    int i() { return m_i; }
    int& iR() { return m_i; }

    int m_i;
};

int main() {
    const Foo& v1 = Foo(1);
    const int& v2 = Foo(2).m_i;
    const int& v3 = Foo(3).i();
    const int& v4 = Foo(4).iR();
}
```


https://godbolt.org/z/rbK1sP8n9

```cpp
using namespace std;
struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    int i() { return m_i; }
    int& iR() { return m_i; }

    int m_i;
};

int main() {
    const Foo& v1 = Foo(1);
    const int& v2 = Foo(2).m_i;
    const int& v3 = Foo(3).i();
    const int& v4 = Foo(4).iR();
}
```


https://godbolt.org/z/rbK1sP8n9

```cpp
using namespace std;
struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    int i() { return m_i; }
    int& iR() { return m_i; }

    int m_i;
};

int main() {
    const Foo& v1 = Foo(1); // Foo(1)
    const int& v2 = Foo(2).m_i;
    const int& v3 = Foo(3).i();
    const int& v4 = Foo(4).iR();
}
```


https://godbolt.org/z/rbK1sP8n9

```cpp
using namespace std;
struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    int i() { return m_i; }
    int& iR() { return m_i; }

    int m_i;
};

int main() {
    const Foo& v1 = Foo(1); // Foo(1)
    const int& v2 = Foo(2).m_i;
    const int& v3 = Foo(3).i();
    const int& v4 = Foo(4).iR();
}
```

https://godbolt.org/z/rbK1sP8n9

```cpp
using namespace std;
struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    int i() { return m_i; }
    int& iR() { return m_i; }

    int m_i;
};

int main() {
    const Foo& v1 = Foo(1); // Foo(1)
    const int& v2 = Foo(2).m_i; // Foo(2)
    const int& v3 = Foo(3).i();
    const int& v4 = Foo(4).iR();
}
```

https://godbolt.org/z/rbK1sP8n9

```cpp
using namespace std;
struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    int i() { return m_i; }
    int& iR() { return m_i; }

    int m_i;
};

int main() {
    const Foo& v1 = Foo(1); // Foo(1)
    const int& v2 = Foo(2).m_i; // Foo(2)
    const int& v3 = Foo(3).i();
    const int& v4 = Foo(4).iR();
}
```


https://godbolt.org/z/rbK1sP8n9

```cpp
using namespace std;
struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    int i() { return m_i; }
    int& iR() { return m_i; }

    int m_i;
};

int main() {
    const Foo& v1 = Foo(1); // Foo(1)
    const int& v2 = Foo(2).m_i; // Foo(2)
    const int& v3 = Foo(3).i(); // Foo(3), ~Foo(3)
    const int& v4 = Foo(4).iR();
}
```

https://godbolt.org/z/rbK1sP8n9

```cpp
using namespace std;
struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    int i() { return m_i; }
    int& iR() { return m_i; }

    int m_i;
};

int main() {
    const Foo& v1 = Foo(1); // Foo(1)
    const int& v2 = Foo(2).m_i; // Foo(2)
    const int& v3 = Foo(3).i(); // Foo(3), ~Foo(3)
    const int& v4 = Foo(4).iR();
}
```


https://godbolt.org/z/rbK1sP8n9

```cpp
using namespace std;
struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    int i() { return m_i; }
    int& iR() { return m_i; }

    int m_i;
};

int main() {
    const Foo& v1 = Foo(1); // Foo(1)
    const int& v2 = Foo(2).m_i; // Foo(2)
    const int& v3 = Foo(3).i(); // Foo(3), ~Foo(3)
    const int& v4 = Foo(4).iR(); // Foo(4), ~Foo(4)
}
```

https://godbolt.org/z/rbK1sP8n9

```cpp
using namespace std;
struct Foo {
    Foo(int i) : m_i(i) {cout << "Foo(" << m_i << ")" << endl; }
    ~Foo() { cout << "~Foo(" << m_i << ")" << endl; }
    int i() { return m_i; }
    int& iR() { return m_i; }

    int m_i;
};

int main() {
    const Foo& v1 = Foo(1); // Foo(1)
    const int& v2 = Foo(2).m_i; // Foo(2)
    const int& v3 = Foo(3).i(); // Foo(3), ~Foo(3)
    const int& v4 = Foo(4).iR(); // Foo(4), ~Foo(4)
}
```

Dangling references

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:

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


https://godbolt.org/z/nvY6EbTjs

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.

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.

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.

(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…

```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

```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

```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

```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







}
```


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

```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;

}
```

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

```
{ // for (const auto& v : Values().m_vec)







}
```


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

```
{ // for (const auto& v : Values().m_vec)
    vector<int, allocator<int>> && __range1 = Values().m_vec;
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
```


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

```
{ // for (const auto& v : Values().m_vec)
    vector<int, allocator<int>> && __range1 = Values().m_vec;
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
```


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

```cpp
{ // for (const auto& v : Values().m_vec)
    vector<int, allocator<int>> && __range1 = Values().m_vec;
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
```

Reference lifetime extension

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

```
{ // for (const auto& v : Values().m_vec)
    vector<int, allocator<int>> && __range1 = Values().m_vec;
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
```

Reference lifetime extension

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

```
{ // for (const auto& v : Values().m_vec)
    vector<int, allocator<int>> && __range1 = Values().m_vec;
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
```

Reference lifetime extension

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

```
{ // for (const auto& v : Values().m_vec)
    vector<int, allocator<int>> && __range1 = Values().m_vec;
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
{ // for (const auto& v : Values().get())
}
```

Reference lifetime extension

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

```
{ // for (const auto& v : Values().m_vec)
    vector<int, allocator<int>> && __range1 = Values().m_vec;
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
{ // for (const auto& v : Values().get())
    const vector<int, allocator<int>> & __range1 = Values().get();
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
```

Reference lifetime extension

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

```
{ // for (const auto& v : Values().m_vec)
    vector<int, allocator<int>> && __range1 = Values().m_vec;
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
{ // for (const auto& v : Values().get())
    const vector<int, allocator<int>> & __range1 = Values().get();
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
```

Reference lifetime extension

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

```cpp
{ // for (const auto& v : Values().m_vec)
    vector<int, allocator<int>> && __range1 = Values().m_vec;
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
{ // for (const auto& v : Values().get())
    const vector<int, allocator<int>> & __range1 = Values().get();
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
```

Reference lifetime extension

Values() lifetime will **not** be extended

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

```
{ // for (const auto& v : Values().m_vec)
    vector<int, allocator<int>> && __range1 = Values().m_vec;
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
```

Reference lifetime extension

```
{ // for (const auto& v : Values().get())
    const vector<int, allocator<int>> & __range1 = Values().get();
    auto __begin1 = __range1.begin();
    auto __end1 = __range1.end();
    for(; operator!=(__begin1, __end1); __begin1.operator++()) {
        int const & v = __begin1.operator*();
        cout.operator<<(v).operator<<(endl);
    }
}
```

Values() lifetime will **not** be extended

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

Review

- What’s object lifetime, when it begins and when it ends
- RAII is great and is everwhere
- RVO/NRVO
- Dangling references
- Lifetime of temporaries and its exceptions and pitfalls
- How to save your future-self time by learning about object lifetime

Thank you!