Constructors and destructors: A few things you might want to know

Pavel Novikov

MUREX

CODE RECKONS

Science to the CORE

Constructors and destructors: A few things you might want to know

Pavel Novikov

 @cpp_ape

R&D Align Technology

Let’s discuss

- rule of zero, rule of three, rule of five, and when to break them
- copying and moving, and equivalency
- constructor overloading
- `explicit` constructors
- `constexpr` constructors and destructors

- destructors are `noexcept` by default
- destructors should be `virtual` for polymorphic types
- C++20: constrained destructors

itempool.com/cpp-ape/live

Pop quiz!

itempool.com/cpp-ape/live

Pop quiz!

For people watching a recording:

itempool.com/cpp-ape/c/FGj25GlvhwU

Rule of zero special functions

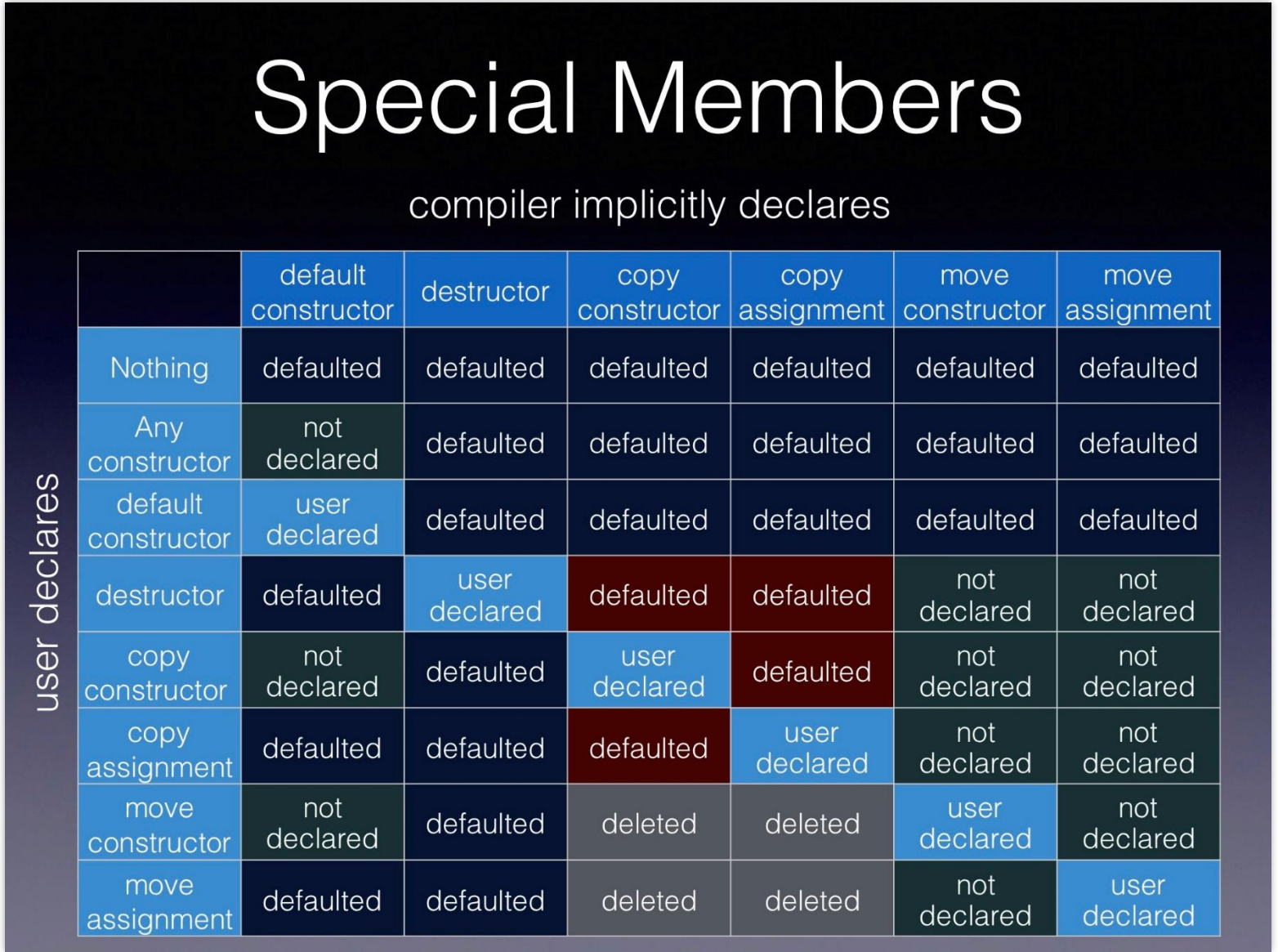
```
struct Widget {
  std::string name;
  int awesomeness;
};
```

C++ Core Guidelines:
C.20: If you can avoid defining default operations, do

```
Widget dummy;
auto widget = Widget{ "Cool widget😎", 9000 };
auto copy = widget;
auto moved = std::move(widget);
```

Rule of zero special functions

Howard Hinnant's table



	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Rule of zero special functions

Howard Hinnant's table

easy

hard!

Special Members

compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Rule of zero special functions

Howard Hinnant's table

easy

hard!

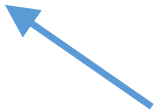
Special Members

compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Rule of zero special functions

```
struct Widget {
  std::string name;
  int awesomeness;
//inline Widget(const Widget&) noexcept(false) = default;
//inline Widget(Widget&&) noexcept = default;
//inline ~Widget() noexcept = default;
//inline Widget() noexcept = default;
};

Widget dummy;
auto widget = Widget{ "Cool widget😎", 9000 };
auto copy = widget;
auto moved = std::move(widget);
```



generated by compiler

Rule of zero special functions

```
struct CopyableMovable {
  void set(std::string_view q, int a) {
    theQuestion = q;
    theAnswer = a;
  }
  void print() {
    std::cout << "Q: " << theQuestion << "\n"
      "A: " << theAnswer << '\n';
  }

private:
  std::string theQuestion;
  int theAnswer;
};
```

Rule of zero special functions

```
CopyableMovable a;
a.set("What's the answer to the ultimate question of life?", 42);
auto b = a;
a.print();
auto c = std::move(a);
a.print();  // do not use moved from objects!
```

outputs:

```
Q: What's the answer to the ultimate question of life?
A: 42
Q:
A: 42
```

Rule of zero special functions

```
CopyableMovable a;
a.set("What's the answer to the ultimate question of life?", 42);
auto b = a;
a.print();
auto c = std::move(a);
a.print();  // do not use moved from objects!
```

outputs:

```
Q: What's the answer to the ultimate question of life?
A: 42
Q:
A: 42
```

Rule of zero special functions

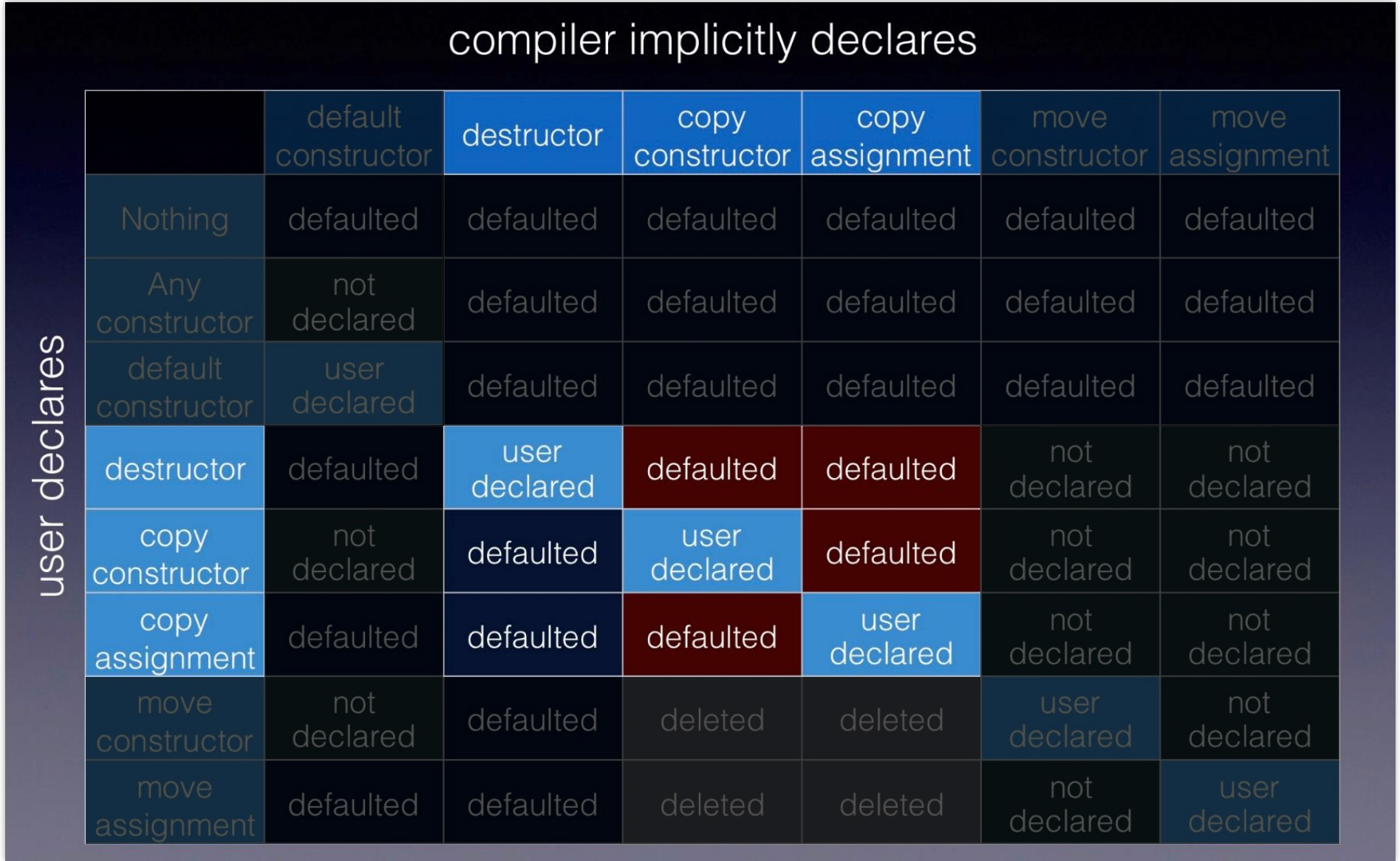
```
CopyableMovable a;
a.set("What's the answer to the ultimate question of life?", 42);
auto b = a;
a.print();
auto c = std::move(a);
a.print();  // do not use moved from objects!
```

outputs:

```
Q: What's the answer to the ultimate question of life?
A: 42
Q:
A: 42
```

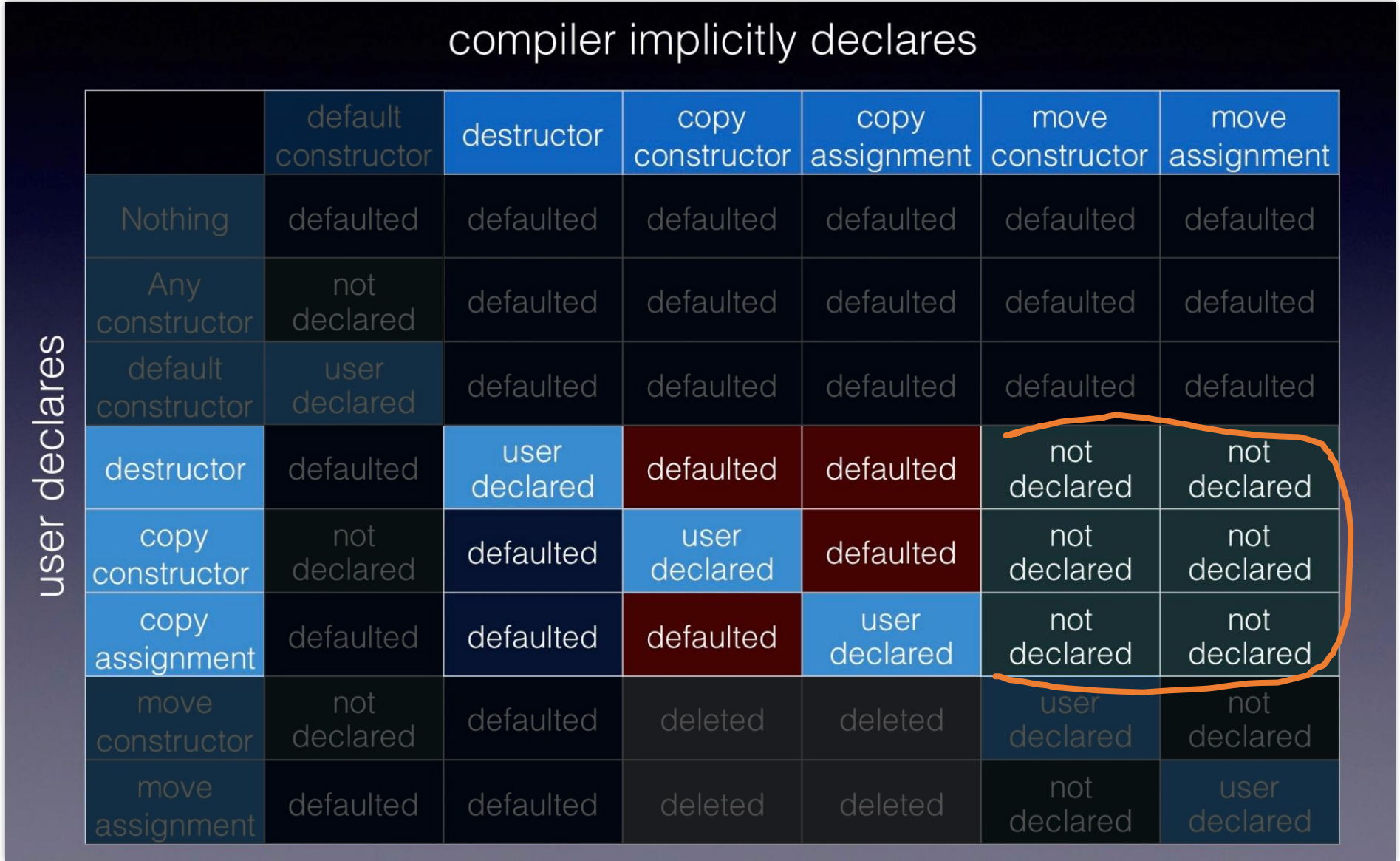
Rule of three special functions

Howard Hinnant's table



Rule of three special functions

Howard Hinnant's table



Rule of three special functions

```
struct Copyable {
  Copyable() = default;
  ~Copyable() {
    free(resource);
  }
  Copyable(const Copyable &other) :
    resource{ clone(other.resource) } {
  }
  Copyable &operator=(const Copyable &other) {
    auto oldResource = std::exchange(resource, clone(other.resource));
    free(oldResource);
    return *this;
  }
private:
  Resource resource = acquireResource();
};
```

default ctor for convenience
(and possibly other ctors)

Rule of three special functions

```
struct Copyable {
  Copyable() = default;
  ~Copyable() {
    free(resource);
  }
  Copyable(const Copyable &other) :
    resource{ clone(other.resource) } {
  }
  Copyable &operator=(const Copyable &other) {
    auto oldResource = std::exchange(resource, clone(other.resource));
    free(oldResource);
    return *this;
  }
private:
  Resource resource = acquireResource();
};
```

the three *special* special functions

Rule of three special functions

default ctor

```
Copyable gadget, fidget;
```

copy ctor

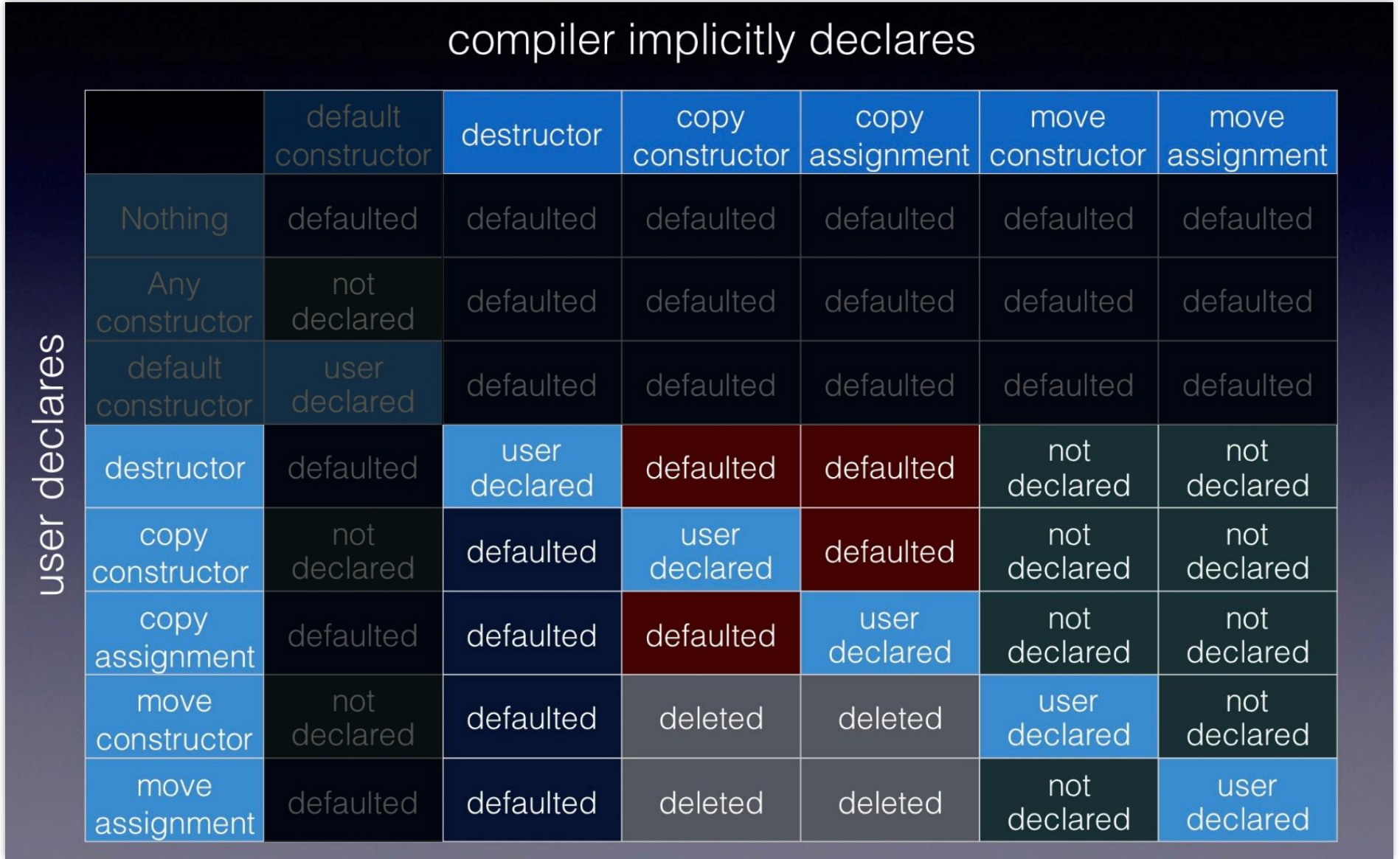
```
auto copy = gadget;
auto copy2 = std::move(gadget);
```

copy assignment

```
copy = fidget;
copy2 = std::move(fidget);
```

Rule of five special functions

Howard Hinnant's table



Rule of five special functions

C++ Core Guidelines:
C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all

```
struct CopyableMovable {
  CopyableMovable();
  ~CopyableMovable();
  CopyableMovable(const CopyableMovable &other);
  CopyableMovable &operator=(const CopyableMovable &other);
  CopyableMovable(CopyableMovable &&other) noexcept;
  CopyableMovable &operator=(CopyableMovable &&other) noexcept;
};
```

Rule of five special functions

C++ Core Guidelines:
C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all

```
struct CopyableMovable {
  CopyableMovable();
  ~CopyableMovable();
  CopyableMovable(const CopyableMovable &other);
  CopyableMovable &operator=(const CopyableMovable &other);
  CopyableMovable(CopyableMovable &&other) noexcept;
  CopyableMovable &operator=(CopyableMovable &&other) noexcept;
};
```

the five *special* special functions

Rule of five special functions

default ctor	`CopyableMovable gadget, fidget;`
copy ctor	`auto copy = gadget;`
move ctor	`auto moved = std::move(gadget);`
copy assignment	`copy = fidget;`
move assignment	`moved = std::move(fidget);`

Rules are there to break them

Rule of three

+

Rule of five (C++ Core Guidelines):

C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all

Let’s break them!

Rules are there to break them

```
struct MovableNonCopyable {
  MovableNonCopyable();
  ~MovableNonCopyable();
  MovableNonCopyable(const MovableNonCopyable&) = delete;
  MovableNonCopyable &operator=(const MovableNonCopyable&) = delete;
  MovableNonCopyable(MovableNonCopyable&&) noexcept;
  MovableNonCopyable &operator=(MovableNonCopyable&&) noexcept;
};

MovableNonCopyable gadget, fidget;

auto copy = gadget;  // does not work
auto moved = std::move(gadget);

copy = fidget;  // does not work
moved = std::move(fidget);
```

itempool.com/cpp-ape/live

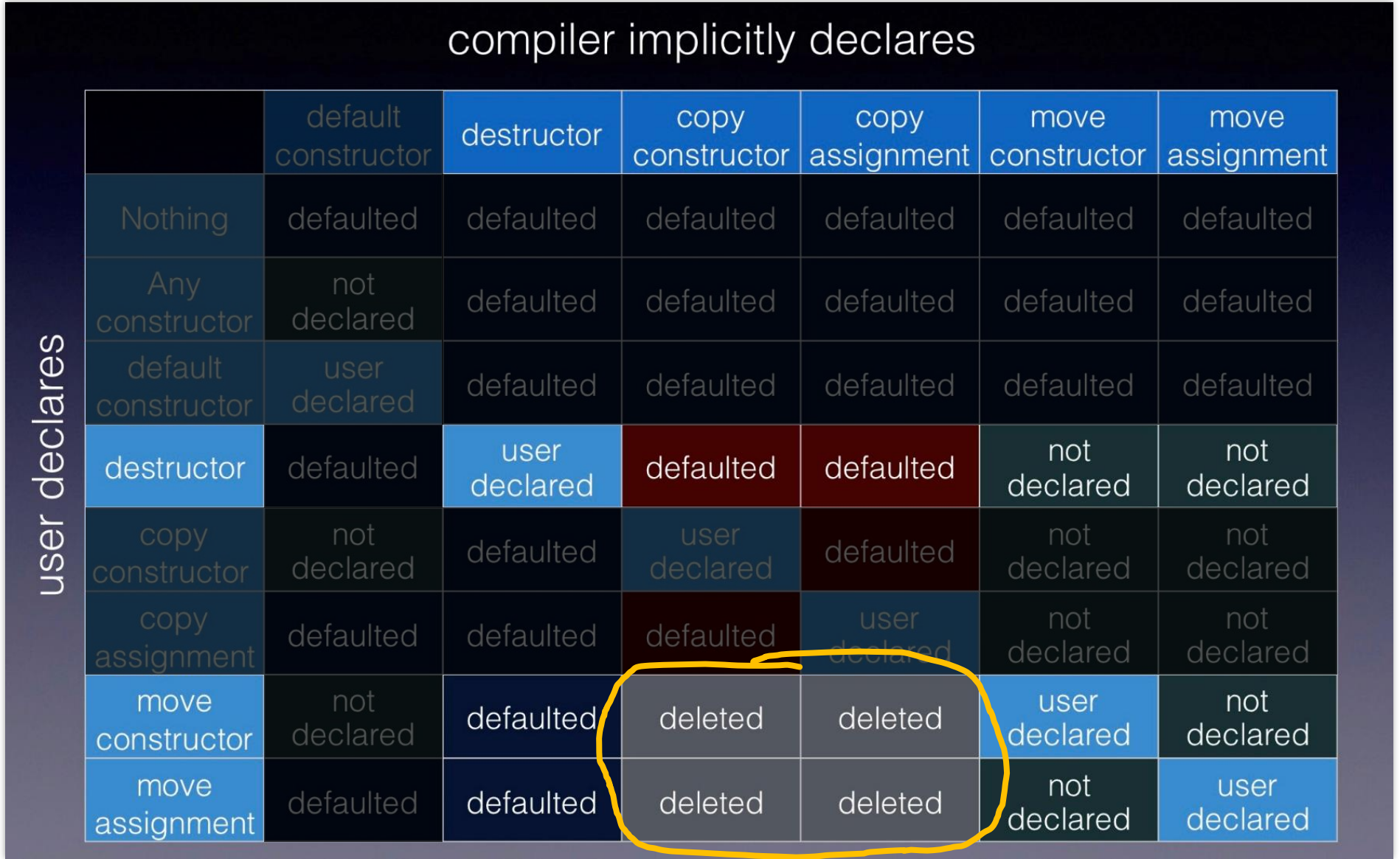
Rules are there to break them

Item 1

```
std::vector<MovableNonCopyable> items;

MovableNonCopyable item;

items.push_back(item);
items.push_back(std::move(item));
items.push_back(MovableNonCopyable{});
```

Rules are there to break them

Howard Hinnant's table



Rules are there to break them

```
struct MovableNonCopyable {
  MovableNonCopyable();
  ~MovableNonCopyable();
  MovableNonCopyable(const MovableNonCopyable&) = delete;
  MovableNonCopyable &operator=(const MovableNonCopyable&) = delete;
  MovableNonCopyable(MovableNonCopyable&&) noexcept;
  MovableNonCopyable &operator=(MovableNonCopyable&&) noexcept;
};
```

Rules are there to break them

```
struct MovableNonCopyable {
  MovableNonCopyable();
  ~MovableNonCopyable();
  MovableNonCopyable(const MovableNonCopyable&) = delete;
  MovableNonCopyable &operator=(const MovableNonCopyable&) = delete;
  MovableNonCopyable(MovableNonCopyable&&) noexcept;
  MovableNonCopyable &operator=(MovableNonCopyable&&) noexcept;
};
```

Rules are there to break them

```
struct MovableNonCopyable {
  MovableNonCopyable();
  ~MovableNonCopyable();
  MovableNonCopyable(const MovableNonCopyable&) = delete;
  MovableNonCopyable &operator=(const MovableNonCopyable&) = delete;
  MovableNonCopyable(MovableNonCopyable&&) noexcept;
  MovableNonCopyable &operator=(MovableNonCopyable&&) noexcept;
};
```

rule of three?

members of the primary template, unique_ptr<T>

```
constexpr unique_ptr() noexcept;                                          (1)
constexpr unique_ptr( nullptr_t ) noexcept;
explicit unique_ptr( pointer p ) noexcept;                                (2)
unique_ptr( pointer p, /* see below */ d1 ) noexcept;                     (3)
unique_ptr( pointer p, /* see below */ d2 ) noexcept;                     (4)
unique_ptr( unique_ptr&& u ) noexcept;                                    (5)
template< class U, class E >
unique_ptr( unique_ptr<U, E>&& u ) noexcept;                              (6)
template< class U >
unique_ptr( std::auto_ptr<U>&& u ) noexcept;                              (7)   (removed in C++17)
```

members of the specialization for arrays, unique_ptr<T[]>

```
constexpr unique_ptr() noexcept;
```

```
unique_ptr( unique_ptr&& u ) noexcept;
~unique_ptr();
unique_ptr& operator=( unique_ptr&& r ) noexcept;
```

```
unique_ptr( unique_ptr&& u ) noexcept;                                    (5)
template< class U, class E >
unique_ptr( unique_ptr<U, E>&&                                            (6)   (since C++17)
```

rule of three?

```
~unique_ptr();     (since C++11)
```

members of the primary template, unique_ptr<T>

```
unique_ptr& operator=( unique_ptr&& r ) noexcept;                         (1)
template< class U, class E >
unique_ptr& operator=( unique_ptr<U,E>&& r ) noexcept;                    (1)
unique_ptr& operator=( nullptr_t ) noexcept;                              (2)
```

members of the specialization for arrays, unique_ptr<T[]>

```
unique_ptr& operator=( unique_ptr&& r ) noexcept;                         (1)
template< class U, class E >
unique_ptr& operator=( unique_ptr<U,E>&& r ) noexcept;                    (1)   (since C++17)
unique_ptr& operator=( nullptr_t ) noexcept;                              (2)
```

Rules are there to break them

```
struct CopyableNonMovable {
  CopyableNonMovable();
  ~CopyableNonMovable();
  CopyableNonMovable(const CopyableNonMovable&);
  CopyableNonMovable &operator=(const CopyableNonMovable&);
  CopyableNonMovable(CopyableNonMovable&&) = delete;
  CopyableNonMovable &operator=(CopyableNonMovable&&) = delete;
};

CopyableNonMovable gadget, fidget;

auto copy = gadget;
auto moved = std::move(gadget);  // does not work

copy = fidget;
moved = std::move(fidget);  // does not work
```

itempool.com/cpp-ape/live

Rules are there to break them

Item 2

```
std::vector<CopyableNonMovable> items;

CopyableNonMovable item;

items.push_back(item);   // ❌MSVC std lib   ✔stdlibc++   ❌libc++

items.push_back(std::move(item));       // ❌ all standard libs
items.push_back(CopyableNonMovable{}); // ❌ all standard libs
```

std::vector<T,Allocator>::push_back

```
void push_back( const T& value );                    (1)   (until C++20)
constexpr void push_back( const T& value );                (since C++20)
void push_back( T&& value );                         (2)   (since C++11)
                                                           (until C++20)
constexpr void push_back( T&& value );                     (since C++20)
```

Appends the given element value to the end of the container.

1) The new element is initialized as a copy of value

Type requirements

- T must meet the requirements of *CopyInsertable* in order to use overload (1).
- T must meet the requirements of *MoveInsertable* in order to use overload (2).

Type requirements

- T must meet the requirements of *CopyInsertable* in order to use overload (1).
- T must meet the requirements of *MoveInsertable* in order to use overload (2).

std::vector<T,Allocator>::push_back

```
void push_back( const T& value );                (1)   (until C++20)
constexpr void push_back( const T& value );            (since C++20)
void push_back( T&& value );                     (2)   (since C++11)
                                                       (until C++20)
constexpr void push_back( T&& value );                 (since C++20)
```

Appends the given element value to the end of the container.

1) The new element is initialized as a copy of value

Type requirements

- T must meet the requirements of *CopyInsertable* in order to use overload (1).
- T must meet the requirements of *MoveInsertable* in order to use overload (2).

Type requirements

- T must meet the requirements of *CopyInsertable* in order to use overload (1).
- T must meet the requirements of *MoveInsertable* in order to use overload (2).

C++ named requirements: *CopyInsertable*

Specifies that an instance of the type can be copy-constructed in-place by a given allocator.

Requirements

The type **T** is ***CopyInsertable*** into the container **X** whose `value_type` is identical to T if **T** is *MoveInsertable* into **X**, and, given

Requirements

The type **T** is ***CopyInsertable*** into the container **X** whose `value_type` is identical to T if **T** is *MoveInsertable* into **X**…

```
std::allocator_traits<A>::construct(m, p, v);
```

And after evaluation, the value of `*p` is equivalent to the value of `v`. The value of `v` is unchanged.

If **X** is not allocator-aware, the term is defined as if **A** were `std::allocator<T>`, except that no allocator object needs to be created, and user-defined specializations of `std::allocator` are not instantiated.

Rules are there to break them

```
struct Widget {
  Widget();
  ~Widget();
  Widget(const Widget&);
  Widget &operator=(const Widget&);
  Widget(Widget&&);
  Widget &operator=(Widget&&);
};
```

the five *special* special functions

Rules are there to break them

```
struct Widget {
  Widget();
  ~Widget();
  Widget(const Widget&);
  Widget &operator=(const Widget&);
  Widget(Widget&&) = default;
  Widget &operator=(Widget&&) = default;
};
```

the five *special* special functions

Rules are there to break them

```
struct Widget {
  Widget();
  ~Widget();
  Widget(const Widget&);
  Widget &operator=(const Widget&);



};
```

the three *special* special functions

Copying and moving, and equivalency

```
struct Gadget {
  Gadget() { std::cout << "Ctor;"; }
  ~Gadget() { std::cout << "Dtor;"; }
  Gadget(const Gadget &other) { std::cout << "Copy;"; }
  //...
};
```

Copying and moving, and equivalency

Item 3

```
struct Gadget {
  Gadget() { std::cout << "Ctor;"; }
  ~Gadget() { std::cout << "Dtor;"; }
  Gadget(const Gadget &other) { std::cout << "Copy;"; }
  //...
};
```

```
Gadget getGadget() {
  return {};
}

int main() {
  auto g1 = getGadget();
}
```

```
Gadget getGadgetCopy() {
  Gadget gadget;
  return gadget;
}
int main() {
  auto g2 = getGadgetCopy();
}
```

Copying and moving, and equivalency

11 Classes [class]

11.10 Initialization [class.init]

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

1 When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the constructor selected for the copy/move operation and/or the destructor for the object have side effects.

In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object.

If the first parameter of the selected constructor is an rvalue reference to the object's type, the destruction of that object occurs when the target would have been destroyed; otherwise, the destruction occurs at the later of the times when the two objects would have been destroyed without the optimization.

Copying and moving, and equivalency

11 Classes [class]

11.10 Initialization [class.init]

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

1 When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the constructor selected for the copy/move operation and/or the destructor for the object have side effects.

In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object.

If the first parameter of the selected constructor is an rvalue reference to the object's type, the destruction of that object occurs when the target would have been destroyed; otherwise, the destruction occurs at the later of the times when the two objects would have been destroyed without the optimization.

Constructor overloading

```
struct Fidget {
  Fidget() = default;
  //...
private:
  Fidget(const Fidget&);
};
```

```
struct Fidget {
  Fidget() = default;
  Fidget(const Fidget&) = delete;
  //...
};
```

```
Fidget f;
auto g = f;
```

Constructor overloading

```
struct Fidget {
  Fidget() = default;
  //...
private:
  Fidget(const Fidget&);
};
```

```
struct Fidget {
  Fidget() = default;
  Fidget(const Fidget&) = delete;
  //...
};
```

```
Fidget f;
auto g = f;
```

MSVC

'Fidget::Fidget': cannot access private member declared in class 'Fidget'

'Fidget::Fidget(const Fidget &)': attempting to reference a deleted function

Constructor overloading

```
struct Fidget {
  Fidget() = default;
  //...
private:
  Fidget(const Fidget&);
};
```

```
struct Fidget {
  Fidget() = default;
  Fidget(const Fidget&) = delete;
  //...
};
```

```
Fidget f;
auto g = f;
```

Clang

calling a private constructor of class 'Fidget'

call to deleted constructor of 'Fidget'

Constructor overloading

```
struct Fidget {
  Fidget() = default;
  //...
private:
  Fidget(const Fidget&);
};
```

overload set: Fidget

```
struct Fidget {
  Fidget() = default;
  Fidget(const Fidget&) = delete;
  //...
};
```

overload set: Fidget

```
Fidget f;
auto g = f;
```

overload resolution

Constructor overloading

```
struct Fidget {
  Fidget() { init(); }
  Fidget(std::string_view featureList) :
    coolness{ getCoolnessFromFeatures(featureList) } {
    init();
  }
  Fidget(double speed) : coolness{ getCoolnessFromSpeed(speed) } {
    init();
  }

private:
  void init();

  int coolness = 0;
  //...
};
```

```
Fidget f{ 9000 };
```

Constructor overloading

```
struct Fidget {
  Fidget() { init(); }
  Fidget(std::string_view featureList) :
    coolness{ getCoolnessFromFeatures(featureList) } {
    init();
  }
  Fidget(double speed) : coolness{ getCoolnessFromSpeed(speed) } {
    init();
  }

private:
  void init();

  int coolness = 0;
  //...
};
```

```
Fidget f{ 9000 };
```

itempool.com/cpp-ape/live

Constructor overloading

Item 4

```
struct Fidget {
  Fidget() : Fidget{ 0 } {}
  Fidget(std::string_view featureList) :
    Fidget{ getCoolnessFromFeatures(featureList) } {}

  Fidget(double speed) : Fidget{ getCoolnessFromSpeed(speed) } {}

private:
  Fidget(int coolness) : coolness{ coolness } {
    // init
  }

  int coolness;
  //...
};
```

delegating constructors

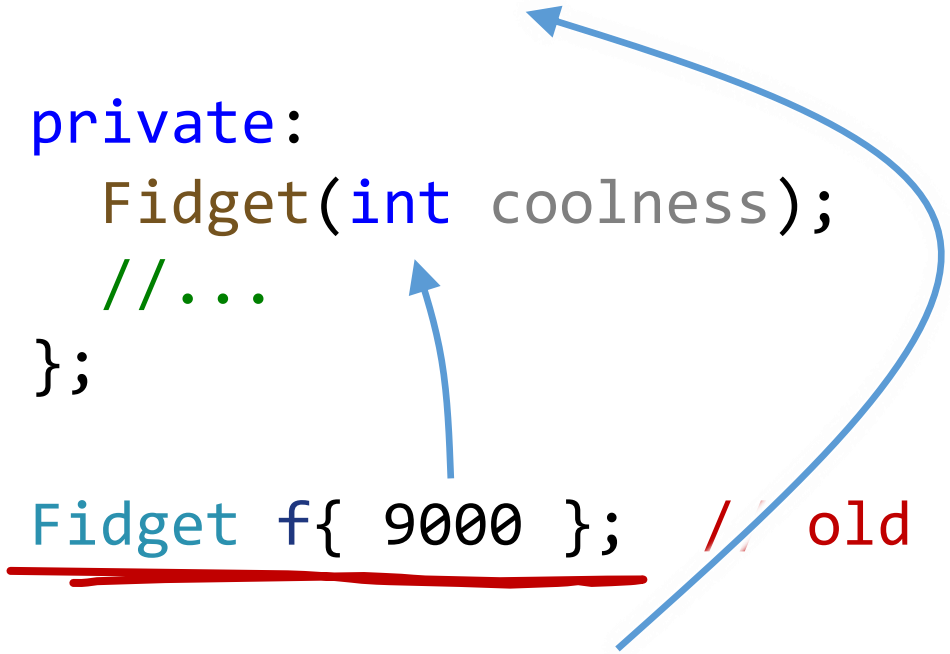
```
Fidget f{ 9000 };
```

Constructor overloading

```
struct Fidget {
  Fidget();
  Fidget(std::string_view featureList);
  Fidget(double speed);

private:
  Fidget(int coolness);
  //...
};

Fidget f{ 9000 };  // old code is broken
```

Constructor overloading

```
struct Fidget {
  Fidget();
  Fidget(std::string_view featureList);
  Fidget(double speed);

private:
  Fidget(int coolness);
  //...
};

Fidget f{ 9000 };  // old code is broken

Fidget f{ 9000.0 };
```




REFAKTORINK

Constructor overloading

```
struct Fidget {
  Fidget();
  Fidget(std::string_view);

private:
  Fidget(int);
};

struct Gizmo : Fidget {
  using Fidget::Fidget;
  Gizmo(double);
};

Fidget f{ 42 };
Gizmo g{ 42 };
```

Constructor overloading

```
struct Fidget {
  Fidget();
  Fidget(std::string_view);

private:
  Fidget(int);
};

struct Gizmo : Fidget {
  using Fidget::Fidget;
  Gizmo(double);
};

Fidget f{ 42 };
Gizmo g{ 42 };
```

cannot access private constructor

Constructor overloading

```
struct Fidget {
  Fidget() : Fidget{ Coolness{ 0 } } {}
  Fidget(std::string_view featureList) :
    Fidget{ Coolness{ getCoolnessFromFeatures(featureList) } } {}
  Fidget(double speed) :
    Fidget{ Coolness{ getCoolnessFromSpeed(speed) } } {}

private:
  struct Coolness { int value; };
  Fidget(Coolness coolness) : coolness{ coolness.value } {
    // init
  }

  int coolness;
  //...
};
```

Constructor overloading

```
struct Fidget {
  Fidget() : Fidget{ Coolness{ 0 } } {}
  Fidget(std::string_view featureList) :
    Fidget{ Coolness{ getCoolnessFromFeatures(featureList) } } {}
  Fidget(double speed) :
    Fidget{ Coolness{ getCoolnessFromSpeed(speed) } } {}

private:
  struct Coolness { int value; };
  Fidget(Coolness coolness) : coolness{ coolness.value } {
    // init
  }

  int coolness;
  //...
};
```

```
struct Gizmo : Fidget {
  using Fidget::Fidget;
  Gizmo(double);
};

Fidget f{ 42 }; // ✔
Gizmo g{ 42 };  // ✔
```

- =`delete` unwanted functions/oveloads

```
struct NonCopyable {
  NonCopyable() {}
  NonCopyable(const NonCopyable &) = delete;
  NonCopyable &operator=(const NonCopyable &) = delete;
};
```

- when overloading, make behavior consistent (the principle of least surprise)
  - remember that private member functions participate in overload resolution

```
struct Fidget {
  Fidget(double speed);

private:
  Fidget(int coolness);
};
```

```
Fidget f{ 9000 }; // broken!
```

  - avoid using `std::initializer_list` in your interfaces or use with extreme caution!

```
std::vector<int> v1(42);   // 42 '0' elements
std::vector<int> v2{ 42 }; // one '42' element
```

Make constructors `explicit` by default

```
struct Fidget {
  Fidget();
  explicit Fidget(std::string_view featureList);
  explicit Fidget(double speed);
  //...
};

void fiddleWith(const Fidget &f);


fiddleWith(42);  // can't convert from number to Fidget

fiddleWith(Fidget{ 42 });  // ✔
```

Make constructors `explicit` by default

```
struct Foo {
  //...
  explicit Foo(double, double);
};
```

Let’s refactor some more!

```
void setupOrder(
  std::string_view name,
  uint64_t priceToBuyInDollars,  // buy if cheaper than priceToBuyInDollars
  uint64_t sumToSpendInDollars,  // amount of money to spend buying stocks
  uint64_t priceToSellInDollars, // sell if more expensive than priceToSellInDollars
  uint32_t stocksToSell          // number of stocks to sell
);

void setupOrder(std::string_view, uint64_t, uint64_t, uint64_t, uint32_t);
```

Let’s refactor some more!

```
void setupOrder(
  std::string_view name,
  uint64_t priceToBuyInDollars,  // buy if cheaper than priceToBuyInDollars
  uint64_t sumToSpendInDollars,  // amount of money to spend buying stocks
  uint64_t priceToSellInDollars, // sell if more expensive than priceToSellInDollars
  uint32_t stocksToSell          // number of stocks to sell
);

void setupOrder(std::string_view, uint64_t, uint64_t, uint64_t, uint32_t);
```

Let’s refactor some more!

```
void setupOrder(
  std::string_view name,
  uint64_t priceToBuyInDollars,  // buy if
  uint64_t sumToSpendInDollars,  // amount
  uint64_t priceToSellInDollars, // sell if
  uint32_t stocksToSell          // number
);

void setupOrder(std::string_view, uint64_t

setupOrder("SNDP", 10, 1000000, 20, 1000);
```

https://prettybigpickles.com/

Let’s refactor some more!

```
void setupOrder(
  std::string_view name,
  uint64_t priceToBuyInDollars,  // buy if cheaper than priceToBuyInDollars
  uint64_t sumToSpendInDollars,  // amount of money to spend buying stocks
  uint64_t priceToSellInDollars, // sell if more expensive than priceToSellInDollars
  uint32_t stocksToSell          // number of stocks to sell
);

void setupOrder(std::string_view, uint64_t, uint64_t, uint64_t, uint32_t);

                  ?      ?      ?      ?
setupOrder("SNDP", 10, 1000000, 20, 1000);
```

Let’s refactor some more!

```
struct Buy {
  uint64_t priceToBuyInDollars;
  uint64_t sumToSpendInDollars;
};

struct Sell {
  uint64_t priceToSellInDollars;
  uint32_t stocksToSell;
};

void setupOrder(std::string_view name,
                const Sell&,
                const Buy&);
```

Let’s refactor some more!

```
setupOrder("SNDP",   10, 1000000  ,   20, 1000  );
```

itempool.com/cpp-ape/live

Let’s refactor some more!

Item 5

```
setupOrder("SNDP", { 10, 1000000 }, { 20, 1000 });
```

Let’s refactor some more!

```
                                // buy $, $ to spend,  sell $, N stocks
    setupOrder("SNDP",                10, 1000000,     20, 1000  );
```

```
void setupOrder(std::string_view, const Sell&,     const Buy&  );
    setupOrder("SNDP",            { 10, 1000000 }, { 20, 1000 });
```

Let’s refactor some more!

```
                                  // buy $, $ to spend,  sell $, N stocks
    setupOrder("SNDP",                  10, 1000000,     20, 1000  );
```

```
void setupOrder(std::string_view, const Sell&,     const Buy&  );
    setupOrder("SNDP",              { 10, 1000000 }, { 20, 1000 });
```

Strong types

Strong types

```
struct Buy {
  explicit Buy(uint64_t priceToBuyInDollars, uint64_t sumToSpendInDollars) :
    priceToBuyInDollars(priceToBuyInDollars),
    sumToSpendInDollars(sumToSpendInDollars)
  {}

  uint64_t priceToBuyInDollars;
  uint64_t sumToSpendInDollars;
};
```

Strong types

```
struct Sell {
  explicit Sell(uint64_t priceToSellInDollars, uint32_t stocksToSell) :
    priceToSellInDollars(priceToSellInDollars),
    stocksToSell(stocksToSell)
  {}

  uint64_t priceToSellInDollars;
  uint32_t stocksToSell;
};
```

Strong types

```
void setupOrder(std::string_view name, const Sell&, const Buy&);


setupOrder("SNDP", { 10, 1000000 }, { 20, 1000 }); // doesn’t work

setupOrder("SNDP", Buy{ 10, 1000000 }, Sell{ 20, 1000 }); // nope

setupOrder("SNDP", Sell{ 20, 1000 }, Buy{ 10, 1000000 }); // ✔
```

Strong types

```
void setupOrder(std::string_view name, const Sell&, const Buy&);


setupOrder("SNDP", { 10, 1000000 }, { 20, 1000 }); // doesn’t work

setupOrder("SNDP", Buy{ 10, 1000000 }, Sell{ 20, 1000 }); // nope

setupOrder("SNDP", Sell{ 20, 1000 }, Buy{ 10, 1000000 }); // ✔
```

Strong types

```
void setupOrder(std::string_view name, const Sell&, const Buy&);


setupOrder("SNDP", { 10, 1000000 }, { 20, 1000 }); // doesn’t work

setupOrder("SNDP", Buy{ 10, 1000000 }, Sell{ 20, 1000 }); // nope

setupOrder("SNDP", Sell{ 20, 1000 }, Buy{ 10, 1000000 }); // ✔
```

Strong types

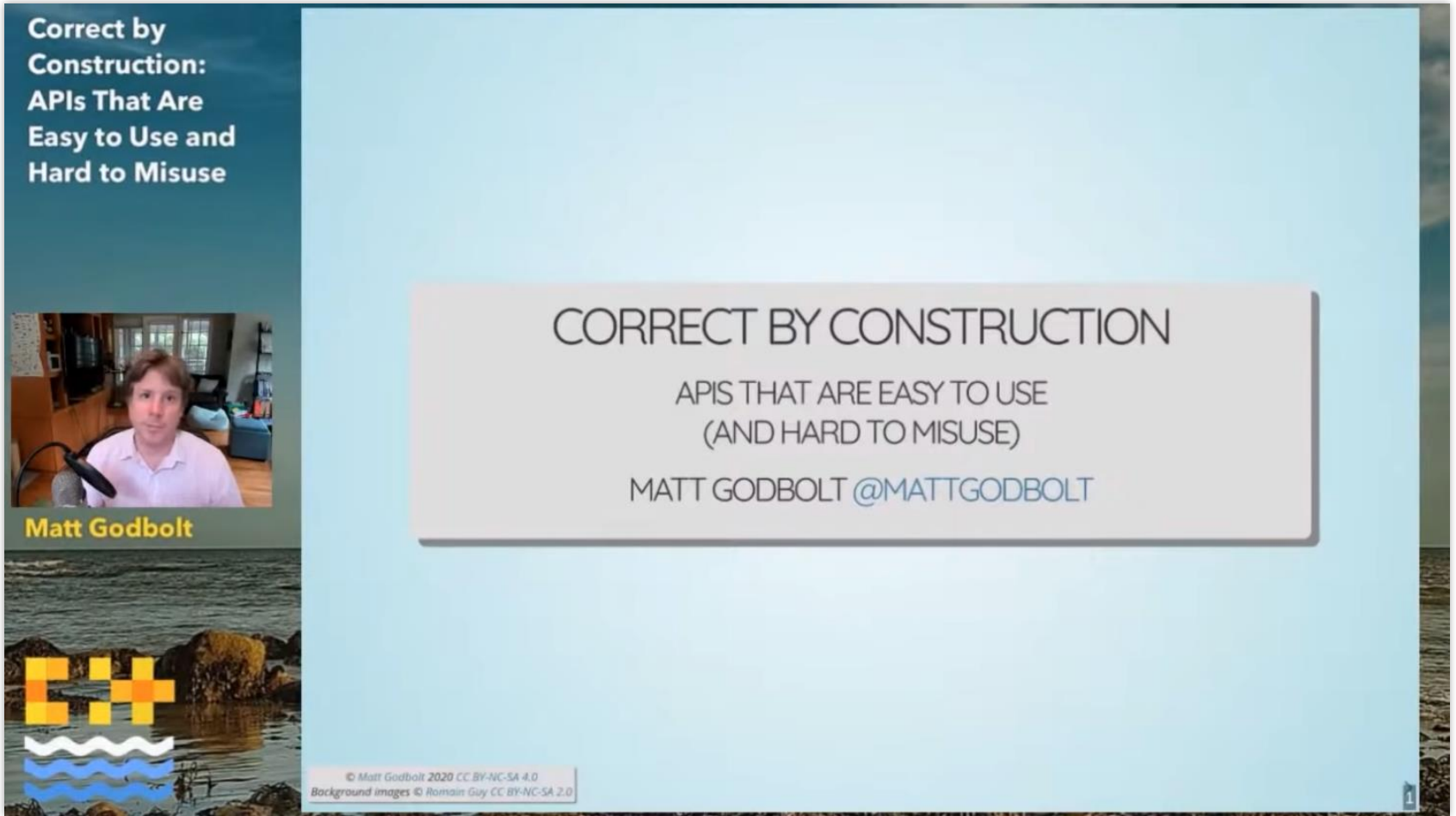
```
setupOrder("SNDP", Sell{ 20, 1000 }, Buy{ 10, 1000000 });
```

```
setupOrder("SNDP", Sell{ 20_$, 1000 }, Buy{ 10_$, 1000000_$ });
```

Correct by construction:
APIs That Are Easy to Use and Hard to Misuse

by Matt Godbolt

https://youtu.be/nLSm3Haxz0I



Consider making ctors and dtors `constexpr`


```
struct Foo {
  int bar;
  double baz;
};
```

```
constexpr auto foo = Foo{ 42, 1.618033988749895 };

constexpr Foo getFoo();

constexpr auto foo = getFoo();

constinit auto foo = getFoo();  // C++20
```

Consider making ctors and dtors `constexpr`


```
struct Foo {
  int bar;
  double baz;
};
```

```
constexpr auto foo = Foo{ 42, 1.618033988749895 };

constexpr Foo getFoo();

constexpr auto foo = getFoo();

constinit auto foo = getFoo();  // C++20
```

Consider making ctors and dtors `constexpr`


```
struct Foo {
  int bar;
  double baz;
};
```

```
constexpr auto foo = Foo{ 42, 1.618033988749895 };

constexpr Foo getFoo();

constexpr auto foo = getFoo();

constinit auto foo = getFoo();  // C++20
```

Consider making ctors and dtors `constexpr`


```
struct Foo {
  int bar;
  double baz;
};
```

```
constexpr auto foo = Foo{ 42, 1.618033988749895 };

constexpr Foo getFoo();

constexpr auto foo = getFoo();

constinit auto foo = getFoo();  // C++20
```

Consider making ctors and dtors `constexpr`

```
struct Fidget {
  constexpr Fidget();
  explicit constexpr Fidget(std::string_view featureList);
  explicit constexpr Fidget(double speed);

private:
  int coolness = 0;
  //...
};

constexpr auto f1 = Fidget{};
constexpr auto f2 = Fidget{ "shiny" };
constexpr auto f3 = Fidget{ 9000 };
```

Consider making ctors and dtors constexpr

```
struct Fidget {
  constexpr Fidget();
  explicit constexpr Fidget(std::string_view featureList);
  explicit constexpr Fidget(double speed);
  trivial destructor
private:
  int coolness = 0;
  //...
};

constexpr auto f1 = Fidget{};
constexpr auto f2 = Fidget{ "shiny" };
constexpr auto f3 = Fidget{ 9000 };
```

Consider making ctors and dtors `constexpr`

literal type

```
struct Fidget {
  constexpr Fidget();
  explicit constexpr Fidget(std::string_view featureList);
  explicit constexpr Fidget(double speed);
```

trivial destructor

```
private:
  int coolness = 0;
  //...
};

constexpr auto f1 = Fidget{};
constexpr auto f2 = Fidget{ "shiny" };
constexpr auto f3 = Fidget{ 9000 };
```

Consider making ctors and dtors `constexpr`

```
struct Fidget {
  constexpr Fidget();
  explicit constexpr Fidget(std::string_view featureList);
  explicit constexpr Fidget(double speed);
  ~Fidget() = default;
private:
  int coolness = 0;
  //...
};

constexpr auto f1 = Fidget{};
constexpr auto f2 = Fidget{ "shiny" };
constexpr auto f3 = Fidget{ 9000 };
```

literal type

trivial destructor

Consider making ctors and dtors `constexpr`

```
struct Fidget {
  constexpr Fidget();
  constexpr ~Fidget();
  explicit constexpr Fidget(std::string_view featureList);
  explicit constexpr Fidget(double speed);

private:
  int coolness = 0;
  //...
};

constexpr auto f1 = Fidget{};
constexpr auto f2 = Fidget{ "shiny" };
constexpr auto f3 = Fidget{ 9000 };
```

Consider making ctors and dtors `constexpr`

literal type in **C++20**

```
struct Fidget {
  constexpr Fidget();
  constexpr ~Fidget();
  explicit constexpr Fidget(std::string_view featureList);
  explicit constexpr Fidget(double speed);

private:
  int coolness = 0;
  //...
};

constexpr auto f1 = Fidget{};
constexpr auto f2 = Fidget{ "shiny" };
constexpr auto f3 = Fidget{ 9000 };
```

Consider making ctors and dtors `constexpr`

literal type in **C++20**

```
struct Fidget {
  constexpr Fidget();
  constexpr ~Fidget();
  explicit constexpr Fidget(std::string_view featureList);
  explicit constexpr Fidget(double speed);

private:
  int coolness = 0;
  //...
};

constexpr auto f1 = Fidget{};
constexpr auto f2 = Fidget{ "shiny" };
constexpr auto f3 = Fidget{ 9000 };
```

itempool.com/cpp-ape/live

Test yourself!

Item 6

```
struct Fidget {
  //...
  explicit constexpr Fidget(double speed) {
    if (speed < 0.)
      throw Error{ "speed must be non-negative" };
    //...
  }
  //...
};

int main() {
  constexpr auto fidget = Fidget{ -1. };
  std::cout << "created a fidget\n";
}
```

Consider making ctors and dtors `constexpr`

```
struct String {
  constexpr String() = default;
  constexpr ~String() {
    delete[] storage;
  }
  constexpr String(const char *s) {
    size = std::char_traits<char>::length(s);
    storage = new char[size];
    std::copy(s, s + size, storage);
  }

private:
  char *storage = nullptr;
  size_t size = 0;
};
```

Consider making ctors and dtors `constexpr`

```
struct String {
  constexpr String() = default;
  constexpr ~String() {
    delete[] storage;
  }
  constexpr String(const char *s) {
    size = std::char_traits<char>::length(s);
    storage = new char[size];
    std::copy(s, s + size, storage);
  }

private:
  char *storage = nullptr;
  size_t size = 0;
};
```

Consider making ctors and dtors `constexpr`

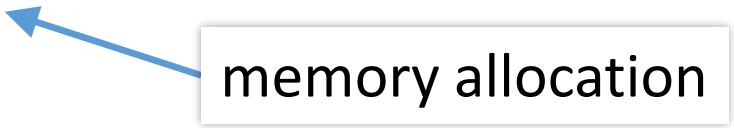
```
constexpr int process(const char *s) {
  String string{ s };
  //...
  return result;
}


constexpr int processingResult = process("text to process!");
```

Consider making ctors and dtors `constexpr`

```
constexpr int process(const char *s) {
  String string{ s };
  //...
  return result;
} string.~String()


constexpr int processingResult = process("text to process!");
```

Consider making ctors and dtors `constexpr`

```
constexpr int process(const char *s) {
  String string{ s };
  //...
  return result;
} string.~String()
```




```
constexpr int processingResult = process("text to process!");
```

Consider making ctors and dtors `constexpr`

```
constexpr int process(const char *s) {
  String string{ s };
  //...
  return result;
} string.~String()
```

memory allocation

matched memory deallocation

```
constexpr int processingResult = process("text to process!");
```

Consider making ctors and dtors `constexpr`

```
constexpr int process(const char *s) {
  String string{ s };
  //...
  return result;
} string.~String()
```

memory allocation

matched memory deallocation

```
constexpr int processingResult = process("text to process!");
```

Consider making ctors and dtors `constexpr`

These containers are **literal types**:

- `basic_string` `// C++20`
- `vector` `// C++20`
- `array`

These views are **literal types**:

- `basic_string_view`
- `span` `// C++20`

Non-literal types

- containers:
  `deque`, `list`, `forward_list`,
  sets & maps,
  unordered sets & maps
- adaptors:
  `queue`, `priority_queue`,
  `stack`

Consider making ctors and dtors `constexpr`

```
void foo() { // not constexpr
  //...
  constexpr auto string = std::string{ "☹" };  // does not work
  //...
}
```

```
constexpr auto string = std::string{ "😭" };  // does not work
```

Destructors are `noexcept` by default

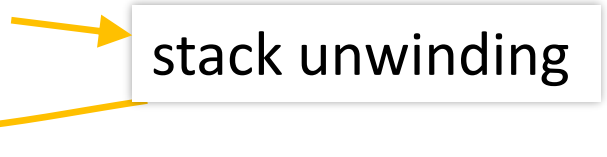
```
struct Bar {
  Bar();
  ~Bar();
};

void foo() {
  Bar b;
  throw "oops!";
}
```

Destructors are `noexcept` by default

```
struct Bar {
  Bar();
  ~Bar();
};

void foo() {
  Bar b;
  throw "oops!";
}
```

Destructors are `noexcept` by default

```
struct Bar {
  Bar();
  ~Bar() noexcept
};

void foo() {
  Bar b;
  throw "oops!";
} b.~Bar()
```



Destructors are `noexcept` by default

```
struct Bar {
  Bar();
  ~Bar() noexcept
};

struct Foo : Bar {
  ~Foo() noexcept
};

struct Baz {
  ~Baz() noexcept

  Bar b;
};
```

Destructors are `noexcept` by default

```
struct Bar {
  Bar();
  ~Bar() noexcept(false)
};

struct Foo : Bar {
  ~Foo();
};

struct Baz {
  ~Baz();

  Bar b;
};
```

Destructors are `noexcept` by default

```
struct Bar {
  Bar();
  ~Bar() noexcept(false)
};

struct Foo : Bar {
  ~Foo() noexcept(false)
};

struct Baz {
  ~Baz();

  Bar b;
};
```

Destructors are `noexcept` by default

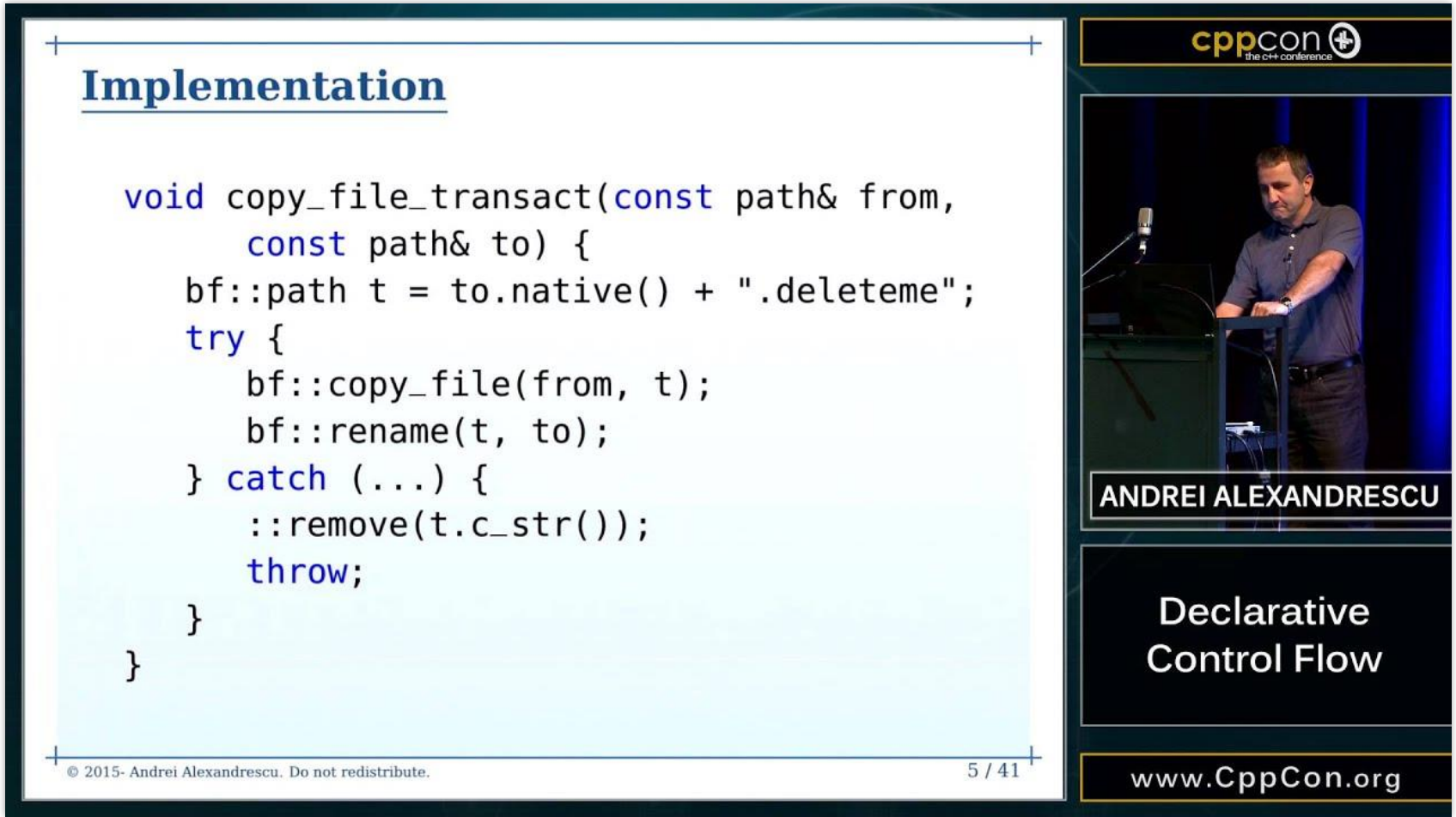
```
struct Bar {
  Bar();
  ~Bar() noexcept(false)
};

struct Foo : Bar {
  ~Foo() noexcept(false)
};

struct Baz {
  ~Baz() noexcept(false)

  Bar b;
};
```

C++ Core Guidelines:
C.37: Make destructors noexcept

Destructors are `noexcept` by default

Andrei Alexandrescu "Declarative Control Flow"

https://youtu.be/WjTrfoiB0MQ



Destructors should be `virtual` for polymorphic types

```
struct ConnectionBase {
  ConnectionBase();
  virtual ~ConnectionBase();
  void setEnabled(bool);
  bool isEnabled() const;

private:
  //...
};
```

Destructors should be `virtual` for polymorphic types

```
struct Connection : ConnectionBase { /*...*/ };
struct Connection1Arg : ConnectionBase { /*...*/ };

struct Subscription {
  template<typename F>
  std::shared_ptr<ConnectionBase> subscribe(F &&f) {
    auto connection = std::make_shared<Connection>(/*...*/);
    //...
    return connection;
  }
  void notify();

private:
  //...
};
```

Destructors should be `virtual` for polymorphic types

```
struct Connection : ConnectionBase { /*...*/ };
struct Connection1Arg : ConnectionBase { /*...*/ };

struct Subscription {
  template<typename F>
  std::shared_ptr<ConnectionBase> subscribe(F &&f) {
    auto connection = std::make_shared<Connection>(/*...*/);
    //...
    return connection;
```

```
Subscription subscription;

auto connection = subscription.subscribe([] { std::cout << "got event\n"; });

subscription.notify();

connection->setEnabled(false);
```

Destructors should be `virtual` for polymorphic types

itempool.com/cpp-ape/live

Item 7

```
struct ConnectionBase {
  //...
  virtual ~ConnectionBase();
  //...
};

struct Connection : ConnectionBase { /*...*/ };

struct Subscription {
  template<typename F>
  std::shared_ptr<ConnectionBase> subscribe(F &&f) {
    auto connection = std::make_shared<Connection>(/*...*/);
    //...
    return connection;
  }
  //...
};
```

Destructors should be `virtual` for polymorphic types

```
struct ConnectionBase {
  ConnectionBase();
  virtual ~ConnectionBase();
  void setEnabled(bool);
  bool isEnabled() const;

private:
  //...
};

auto connection = subscription.subscribe(/*...*/);

connection->setEnabled(false);
```

not `virtual`, can not `override` in derived types

Destructors should be `virtual` for polymorphic types

```
struct ConnectionBase {
  ConnectionBase();
  virtual ~ConnectionBase();
  void setEnabled(bool);
  bool isEnabled() const;

private:
  //...
};

auto connection = subscription.subscribe(/*...*/);

connection->setEnabled(false);
```

not `virtual`, can not `override` in derived types

Destructors should be `virtual` for polymorphic types

Very simple rule:

- polymorphic types (i.e. types with virtual functions) should have virtual destructor

```
struct Interface {
  virtual ~Interface() = default;
  virtual void abstractMethod() = 0;
  virtual void methodWithBaseImplementation() {}
};

struct Instance : Interface {
  void abstractMethod() override {}
};
```

- if the type is not polymorphic, destructor should be non-virtual

C++20: constrained destructors

```
template<typename T> struct Optional {
  Optional() = default;
  Optional(T value) : value{ value }, hasValue{ true } {}
  ~Optional() {
    if (hasValue)
      value.~T();
  }


private:
  union {
    int empty;
    T value;
  };
  bool hasValue = false;
};
```

C++20: constrained destructors

```
template<typename T> struct Optional {
  Optional() = default;
  Optional(T value) : value{ value }, hasValue{ true } {}
  ~Optional() requires(!std::is_trivially_destructible_v<T>) {
    if (hasValue)
      value.~T();
  }
  ~Optional() = default;  // trivial dtor

private:
  union {
    int empty;
    T value;
  };
  bool hasValue = false;
};
```

two *prospective* destructors


Thanks for watching!
many interesting
much thanks
wow
very audience
so knowledge

Thanks for watching!

itempool.com/cpp-ape/live

Constructors and destructors: A few things you might want to know

Pavel Novikov

@cpp_ape

R&D Align Technology

Thanks to Howard Hinnant, Stephan T. Lavavej and Jennifer Yao for feedback and for answering my questions!

Slides: https://git.io/JMBpo

References

- C++ Core Guidelines https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- Howard E. Hinnant. How I Declare My `class` And Why https://howardhinnant.github.io/classdecl.html
- Working Draft, Standard for Programming Language C++ https://eel.is/c++draft/class.copy.elision
- The Sandwich With A Pretty Big Pickle In It Corporation https://prettybigpickles.com/
- Matt Godbolt. Correct by construction: APIs That Are Easy to Use and Hard to Misuse https://youtu.be/nLSm3Haxz0I
- Andrei Alexandrescu. Declarative Control Flow https://youtu.be/WjTrfoiB0MQ
- C++ Extensions for Library Fundamentals, Version 3 https://en.cppreference.com/w/cpp/experimental/lib_extensions_3

Bonus slides

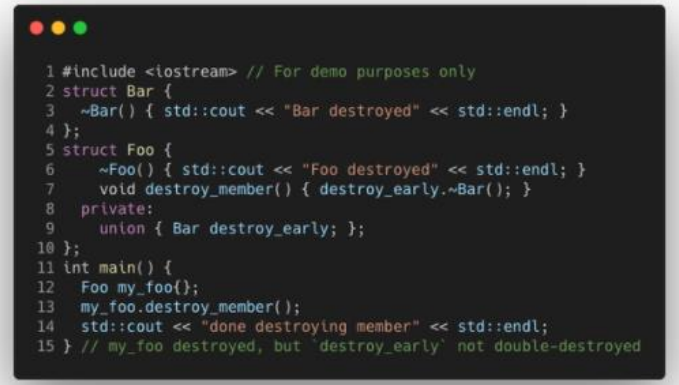
Don’t try this at home!

```
struct Foo {
  Foo();
  ~Foo();
};

struct Bar {
  ~Bar() {}
  void releaseResources() {
    lifetimeManagedManually.~Foo();
  }

private:
  union {
    Foo lifetimeManagedManually{};
  };
};

Bar bar;
bar.releaseResources();
```

Daisy Hollman
@The_Whole_Daisy

Cute C++ trick of the day: you can disable the destructor of a data member using an anonymous union: godbolt.org/z/K1ozxY

(It's the kind of thing that rarely comes up, but it's a nifty tool to have in your back pocket for those odd situations where it does.)



Daisy Hollman
@The_Whole_Daisy

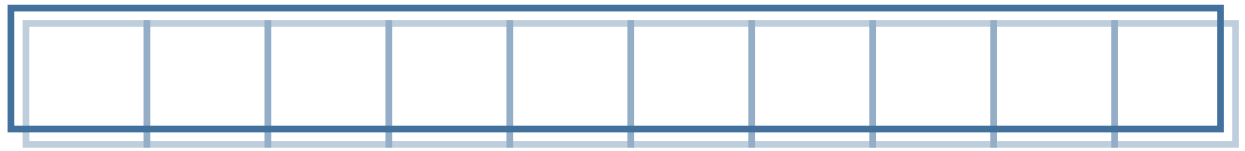
Replying to @DSPonFPGA

It comes up most often in asynchronous code, particularly with things like coroutines. I learned this trick originally from @lewissbaker, who uses it pretty extensively in the Unifex executor library:

facebookexperimental/libunifex
Unified Executors. Contribute to facebookexperimental/libunifex development by creating an ...
github.com

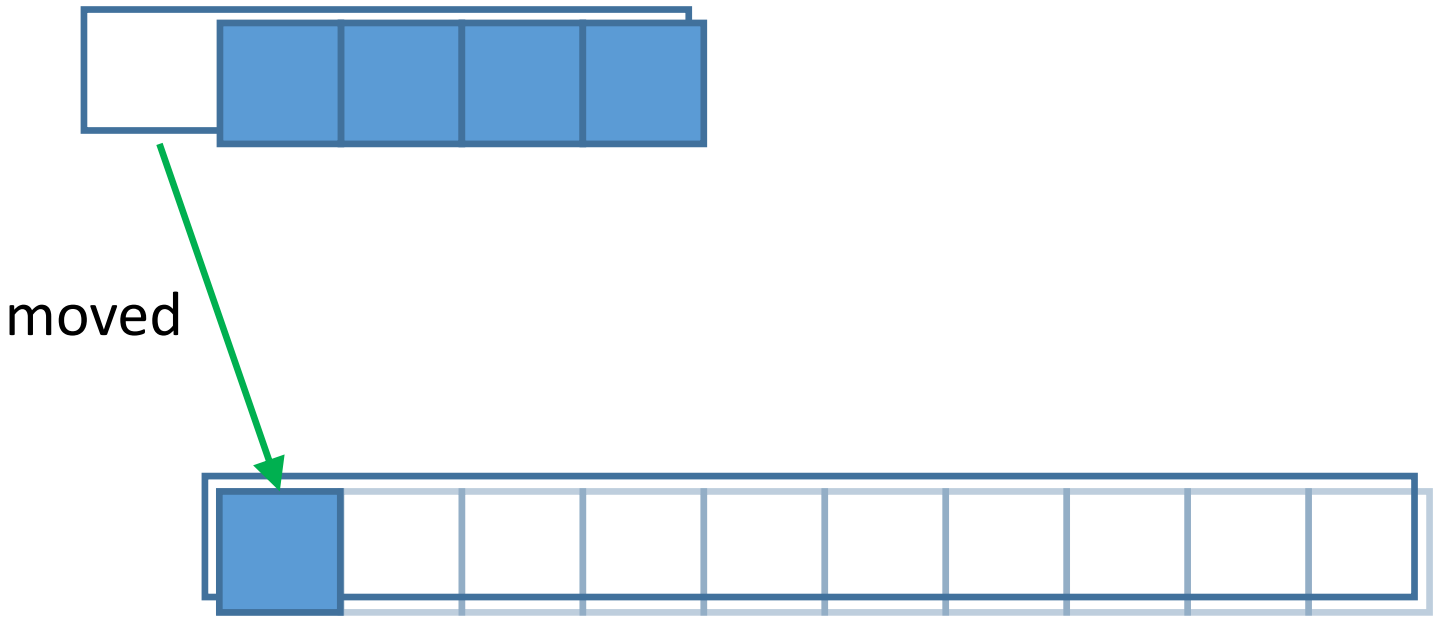
Exception guarantees

```
std::vector<T> items;
T item;
items.push_back(item);
```



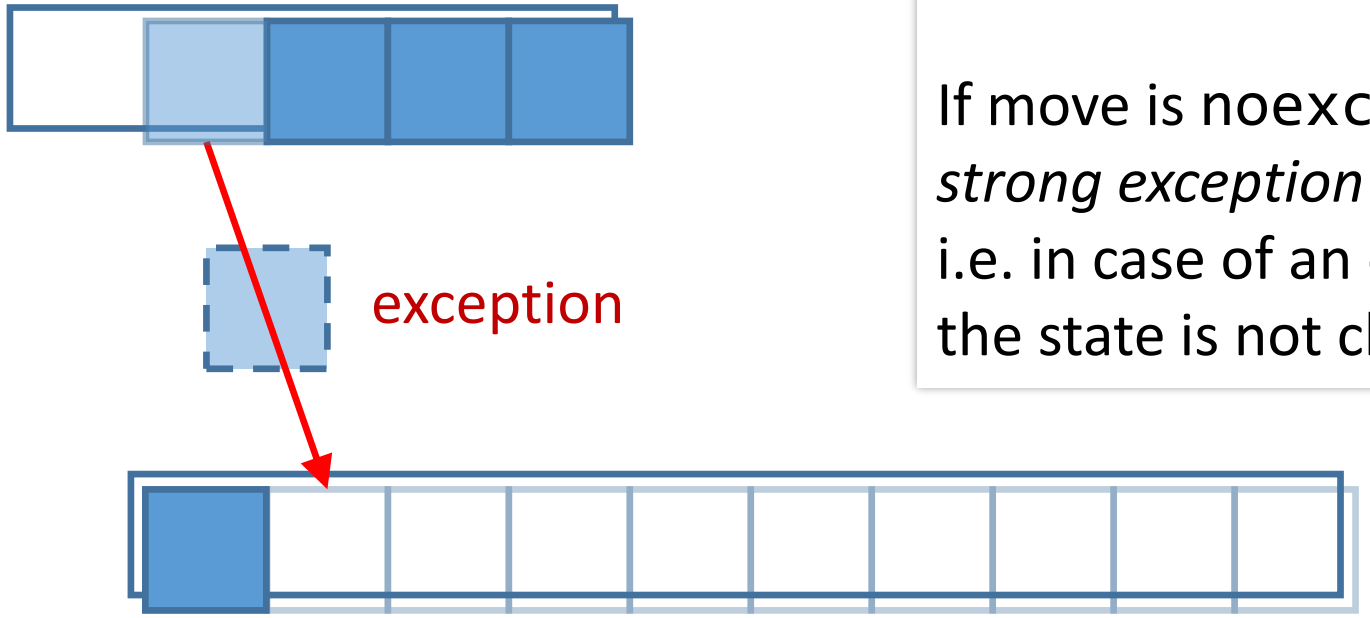
Exception guarantees

```
std::vector<T> items;
T item;
items.push_back(item);
```



Exception guarantees

```
std::vector<T> items;
T item;
items.push_back(item);
```



If during move an exception is thrown,
the state becomes inconsistent (“unspecified”),
hence weaker exception guarantee.

If move is `noexcept`, the operation has
strong exception guarantee,
i.e. in case of an exception (e.g. `bad_alloc`)
the state is not changed.

Strong types

```
struct Dollars {
  explicit Dollars(uint64_t amount) : amount{ amount } {}

  uint64_t amount;
};

auto operator""_$(uint64_t amount) {
  return Dollars{ amount };
}
```

Strong types

```
struct Buy {
  explicit Buy(Dollars priceToBuy, Dollars sumToSpend) :
    priceToBuy(priceToBuy),
    sumToSpend(sumToSpend)
  {}

  Dollars priceToBuy;
  Dollars sumToSpend;
};

struct Sell {
  explicit Sell(Dollars priceToSell, uint32_t stocksToSell) :
    priceToSell(priceToSell),
    stocksToSell(stocksToSell)
  {}

  Dollars priceToSell;
  uint32_t stocksToSell;
};
```

Strong types

```
void setupOrder(std::string_view name, const Sell&, const Buy&);


setupOrder("SNDP", Sell{ 20_$, 1000 }, Buy{ 10_$, 1000000_$ });
```

User-declared vs. user-provided constructors

abseil.io/tips/146

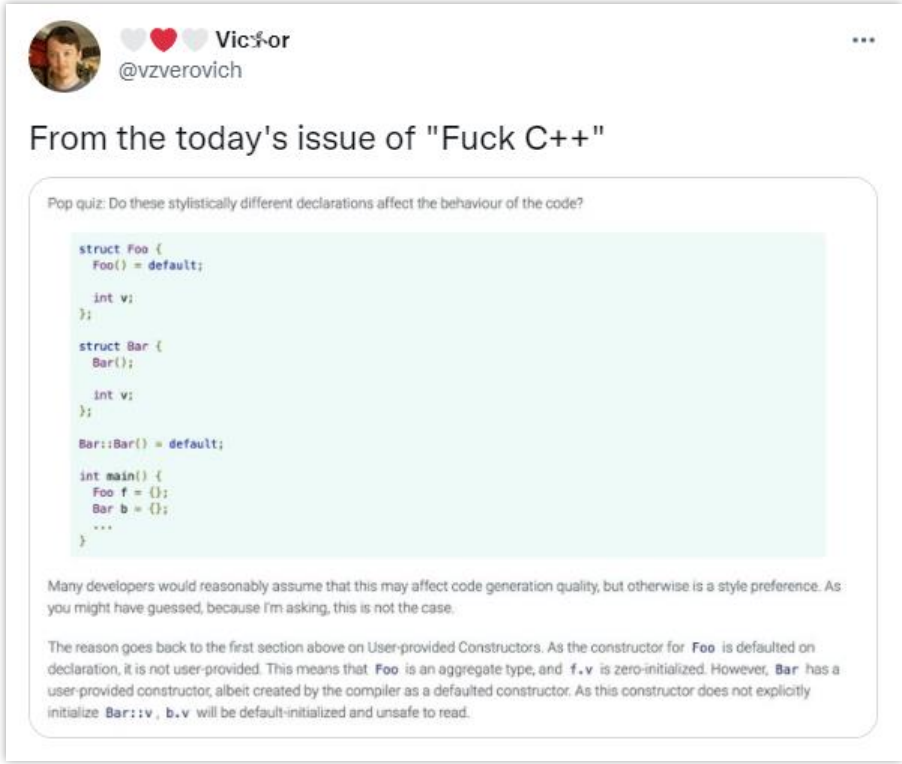
```
struct Foo {
  Foo() = default;
  int v;
};

struct Bar {
  Bar();
  int v;
};
Bar::Bar() = default;

int main() {
  Foo f = {};
  Bar b = {};
}
```

user-declared, not user-provided, hence trivial constructor

user-provided, non-trivial constructor, `Bar::v` is not explicitly initialized

`f.v` is *zero*-initialized

`b.v` is *default*-initialized to indeterminate value



Types of polymorphism

	Set of types	
Set of operations	known	unknown
known		`virtual`
unknown		

Types of polymorphism

	Set of types	
Set of operations	known	unknown
known		virtual
unknown		

```
struct Interface {
  virtual ~Interface() = default;
  virtual void abstractMethod() = 0;
  virtual void methodWithBaseImplementation();
};

struct Instance : Interface {
  void abstractMethod() override;
};

std::unique_ptr<Interface> instance = std::make_unique<Instance>();
```

Types of polymorphism

	Set of types	
Set of operations	known	unknown
known		`virtual`
unknown	`variant`	

Types of polymorphism

Set of operations	known
known	
unknown	variant

Set of types

```
struct Ellipse {/*...*/};
struct Polygon {/*...*/};

std::variant<Ellipse, Polygon> shape{ Ellipse{} };

auto visitor = [](const auto &s) {
  using T = std::remove_reference_t<decltype(s)>;
  if constexpr (std::is_same_v<T, Ellipse>) {
    // do ellipse stuff
  }
  else {
    // do polygon stuff
  }
};
std::visit(visitor, shape);
```

Types of polymorphism

	Set of types	
Set of operations	known	unknown
known	no polymorphism	`virtual`
unknown	`variant`	

Types of polymorphism

	Set of types	
Set of operations	known	unknown
known	no polymorphism	`virtual`
unknown	`variant`	dynamic typing

Observing exceptions in constructors

```
struct Base {
  Base(int);
};

struct Foo : Base {
  Foo(std::string_view s) try : Base(42), member{ s } {
    // rest of init
  }
  catch (...) {
    // handle error
  }//implicit 'throw;' here

private:
  std::string member;
};
```

Constructing virtual base type in a hierarchy with multiple virtual inheritance

intentionally left blank