Special member functions in C++

Kris van Rens // C++ on Sea 2023 // Folkestone, UK

Special Member Functions in C++

Kris van Rens

What’s ahead?

- Fundamentals
- Declaration rules
- Implementation guidelines
- Thoughts and ponderings

A little bit about me

🤓 👩‍👩‍👧‍👦 🎸 🏃 🧗

kris@vanrens.org

Premise and glossary

I use C++20 throughout, unless indicated otherwise

Glossary

SMF	==	Special Member Function
ctor	==	Constructor
dtor	==	Destructor

Fundamentals

Define ‘special’..

> *Some member functions are **special**: under certain circumstances they are defined by the compiler even if not defined by the user.*

cppreference.com § Non-static member functions

Declare vs define

Declaration

Declarations (re-)introduce names into the program. Each kind of entity is declared differently.

cppreference.com § Declarations

Definition

Definitions are declarations that fully define the entity introduced by the declaration.

cppreference.com § Definitions and ODR

Declarations

(introduce the name only)

```cpp
namespace Example
{
}

void func();

using Func = std::function<void()>;

enum class E;

template<typename T>
class Y;

static_assert(/* ..condition.. */);
```

Definitions

(introduce the name + definition)

```cpp
int i = 42;

float array[2] = {};

void func() { /* ... */  }

enum class E { Zero, One };

template<typename T> // A.K.A. 'temploid'.
class Y {
  auto size() {
    return sizeof(T);
  }
};
```

Magic!

Original code:

```cpp
class X {};

int main() {
  X x1;

  X x2{x1};
  x2 = x1;

  X x3{std::move(x1)};

  X x4;
  x4 = std::move(x2);
}
```

What the compiler 🧙 will define for X:

```cpp
class X {
public:
  inline constexpr X() noexcept = default;
  // inline ~X() noexcept = default;

  inline constexpr X(const X&) noexcept            = default;
  inline constexpr X& operator=(const X&) noexcept = default;

  inline constexpr X(X&&) noexcept            = default;
  inline constexpr X& operator=(X&&) noexcept = default;
};
```

Try it on CppInsights

When the magic is gone

Constant member:

```cpp
class X {
  const int value_;
};

int main() {
  X x; // Compiler error!
}
```

Reference member:

```cpp
class Y {
  uint8_t &data_;
};

int main() {
  Y y; // Compiler error!
}
```

Custom constructor:

```cpp
struct Z {
  Z(int value) { /* ... */ }
};

int main() {
  Z z; // Compiler error!
}
```

Examples are ill-formed, and constructors are deleted (+ other SMFs..?!)

Having the magic return

```cpp
class X {
  const int value_;

public:
  explicit X(int arg)
    : value_{arg} {
  }
};

int main() {
  X x1{42};             // Calls custom ctor.
  X x2{x1};             // Calls copy ctor.
  X x3{std::move(x2)}; // Calls move ctor.

  // Any assignment is still impossible..
}
```

```cpp
class Y {
  uint8_t &data_;

public:
  explicit Y(uint8_t& data)
    : data_{data} {
  }
};

int main() {
  uint8_t data = 42;

  Y y1{data};           // Calls custom ctor.
  Y y2{y1};             // Calls copy ctor.
  Y y3{std::move(y2)}; // Calls move ctor.

  // Any assignment is still impossible..
}
```

```cpp
struct Z {
  Z() = default;
  Z(int value) { /* ... */ }
};

int main() {
  Z z1;
  Z z2{42};
}
```

(all ops possible for Z)

What are the options?

- **Do nothing**,
- **User-declare the SMF**:
  - `SMF(..) { /* ..custom implementation.. */ }`
  - `SMF(..) = default;`
  - `SMF(..) = delete;`

What the magic is made of

`default`-defined SMFs:

> *The corresponding action (default construct/copy/move) will be performed on the non-`static` member variables (in order).*

`delete`d SMFs:

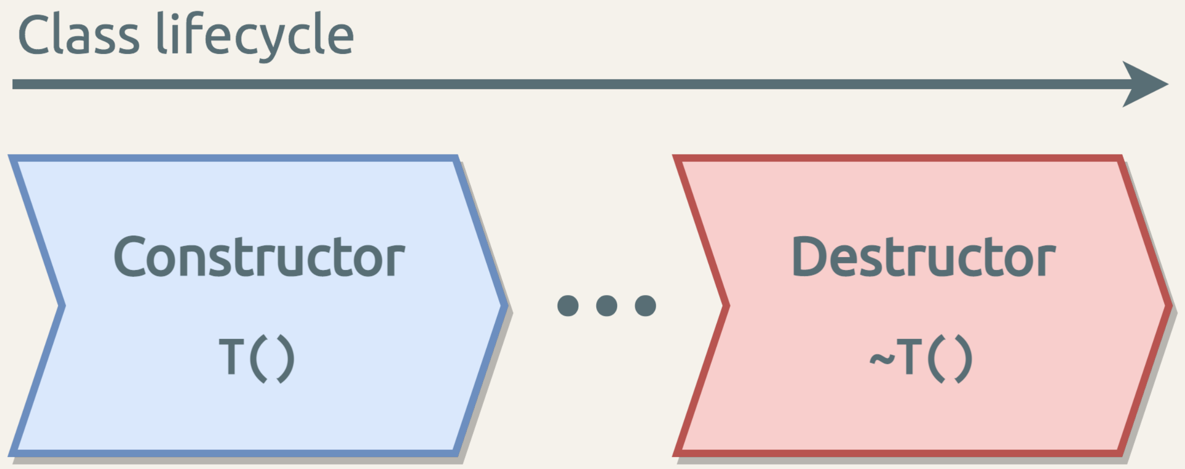
> *The corresponding SMF is declared but may not be used.*

(there are exceptions)

SMF prototypes

```cpp
class T {
public:
  T();                          // Default constructor.
  ~T();                         // Destructor.

  T(const T& other);            // Copy constructor.
  T& operator=(const T& other); // Copy assignment operator.

  T(T&& other);                 // Move constructor (C++11 and later).
  T& operator=(T&& other);      // Move assignment operator (C++11 and later).
};
```

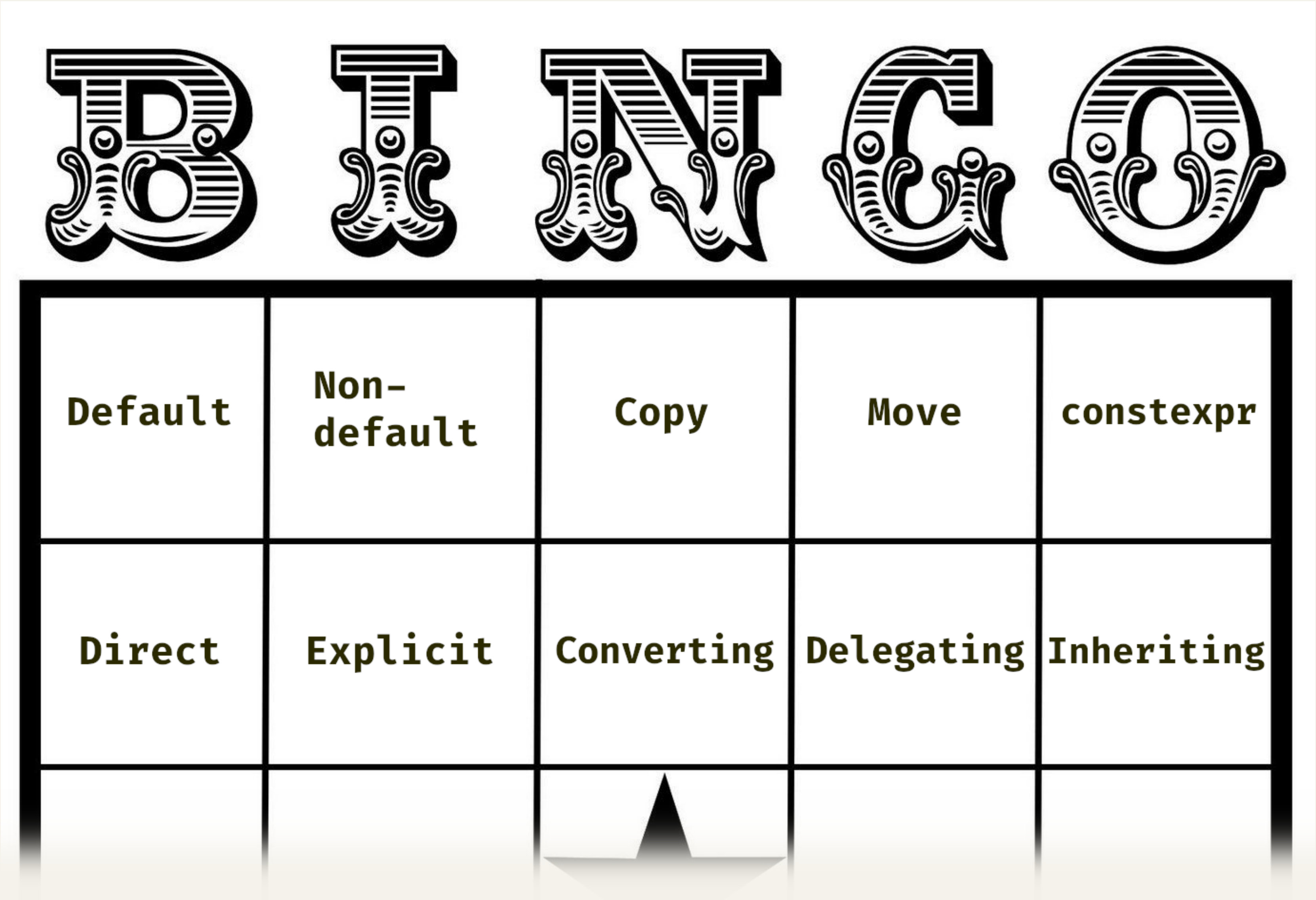
Mandatory memory material Ⓜ



SMF: Constructors

Called when initialization of an object takes place.

```cpp
class Zaphod { /* ... */ };

Zaphod a;               // Calls default ctor.
Zaphod b{};             // Calls default ctor.
auto   c = Zaphod{};    // Calls default ctor ('copy initialization').

Zaphod d{42};           // Calls 'some' non-default ctor taking an int.

Zaphod e{a};            // Calls copy ctor.

Zaphod f{std::move(a)}; // Calls move ctor.
```



B	I	N	G	O
Default	Non-default	Copy	Move	constexpr
Direct	Explicit	Converting	Delegating	Inheriting

Sorry, no bingo 😏

- **Default** ctor
- **Non-default** ctor(s)
- **Copy** ctor
- **Move** ctor

SMF: Constructors

Default ctor

May be called without arguments

```cpp
class Zaphod { /* ... */ };

Zaphod a;
Zaphod b();
Zaphod c{};
auto d = Zaphod{};
```

Non-default ctor

Any ctor that is not a default ctor

```cpp
class Zaphod { /* ... */ };

Zaphod a{42};
Zaphod b("Hi!", 3.1415f);
Zaphod c{a};             // Calls copy ctor.
Zaphod d{std::move(b)}; // Calls move ctor.
```

SMF: Constructors

A ctor with (all) default arguments **counts as a default ctor**

User-declared default ctor:

```cpp
class Point2D {
  float x_{};
  float y_{};

public:
  Point2D() = default; // Request a default implementation.

  Point2D(float x, float y)
    : x_{x}, y_{y} {
  }
};

Point2D p1{1.23f, 4.56f}; // OK.
Point2D p2{};             // OK, initializes to [0.0f, 0.0f].
```

Non-default ctor as a default ctor:

```cpp
class Point2D {
  float x_;
  float y_;

public:
  Point2D(float x = {}, float y = {}) // Default ctor.
    : x_{x}, y_{y} {
  }
};

Point2D p1{1.23f, 4.56f}; // OK.
Point2D p2{};             // OK, initializes to [0.0f, 0.0f].
Point2D p3{1.23f};        // OK, initializes to [1.23f, 0.0f].
```

SMF: Destructor ~T()

Called when object lifetime ends

```cpp
class Vogon { /* ... */ };

{
  Vogon a;
  Vogon b;

  {
    Vogon c;
  } // Dtor of 'c' is called.
}   // Dtor of 'b', then 'a' is called.

Vogon *d = new Vogon{};
delete d; // Dtor of 'd' is called.
```

SMF: Copy operations

Copy constructor	`T(const T&)`
Copy assignment operator	`T& operator=(const T&)`

```cpp
class Dolly { /* ... */ };

Dolly a;

Dolly b{a};  // Calls copy ctor.
Dolly c = a; // Calls copy ctor.

b = a;    // Calls copy assignment operator.
```

SMF: Move operations

Move constructor	`T(T&&)`
Move assignment operator	`T& operator=(T&&)`

```cpp
class Marvin { /* ... */ };

Marvin a;

Marvin b{std::move(a)};  // Calls move ctor.
Marvin c = std::move(b); // Calls move ctor.

b = std::move(c);        // Calls move assignment operator.
```

Noisy type

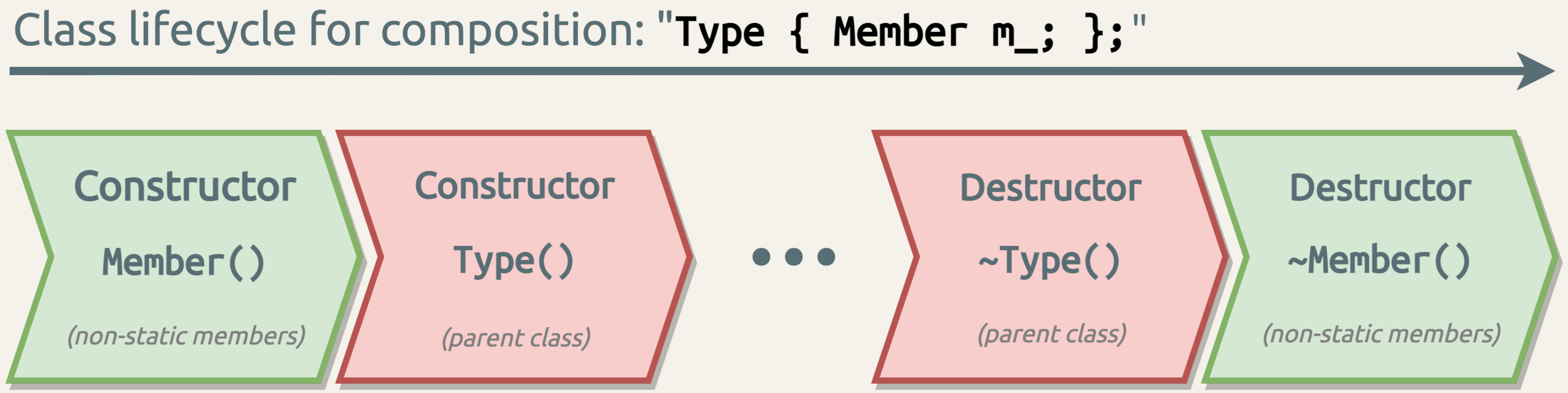
```cpp
#include <cstdio>

#define LOG puts(__PRETTY_FUNCTION__)

struct T {
  T()                        { LOG; }
  ~T()                       { LOG; }
  T(const T&)                { LOG; }
  T& operator=(const T&) { LOG; return *this; }
  T(T&&)                     { LOG; }
  T& operator=(T&&)      { LOG; return *this; }
};
```

SMFs and composition

```cpp
class Member { /* ... */ };

class Type {
  Member m_;
};

Type t;
```

Output:

```
Member::Member()
Type::Type()
Type::~Type()
Member::~Member()
```

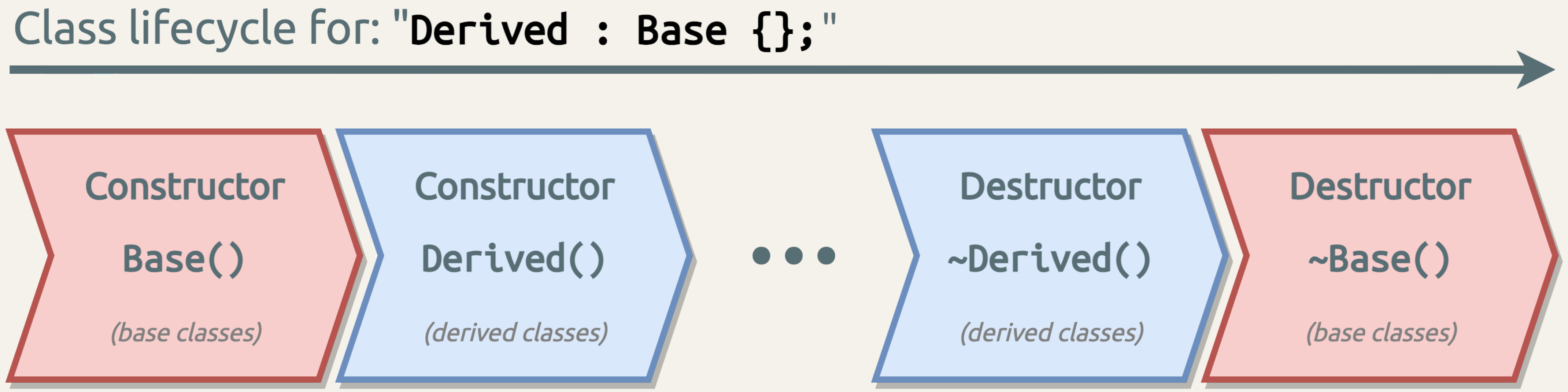


SMFs and inheritance

```cpp
class Base { /* ... */ };

class Derived : Base {};

Derived d;
```

Output:

```
Base::Base()
Derived::Derived()
Derived::~Derived()
Base::~Base()
```



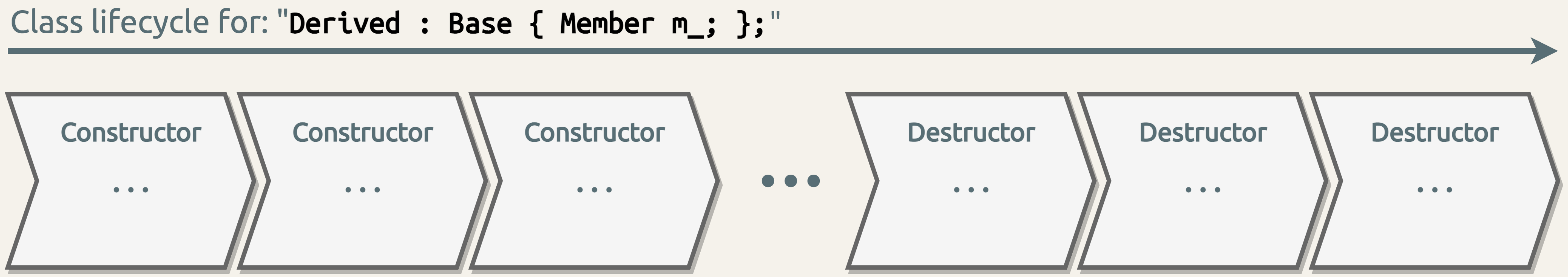
Class lifecycle for: "**Derived : Base {};**"

SMFs and inheritance

```cpp
class Base   { /* ... */ };
class Member { /* ... */ };

class Derived : Base {
  Member m_;
};

Derived d;
```

Output:

```
// ?
// ?
// ?
// ?
// ?
// ?
```



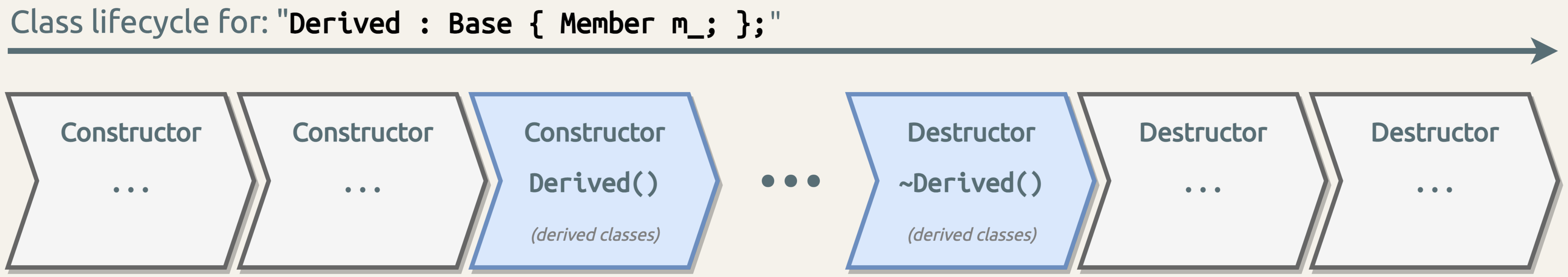
Class lifecycle for: "`Derived : Base { Member m_; };`"

SMFs and inheritance

```cpp
class Base   { /* ... */ };
class Member { /* ... */ };

class Derived : Base {
  Member m_;
};

Derived d;
```

Output:

```
// ?
// ?
Derived::Derived()
Derived::~Derived()
// ?
// ?
```



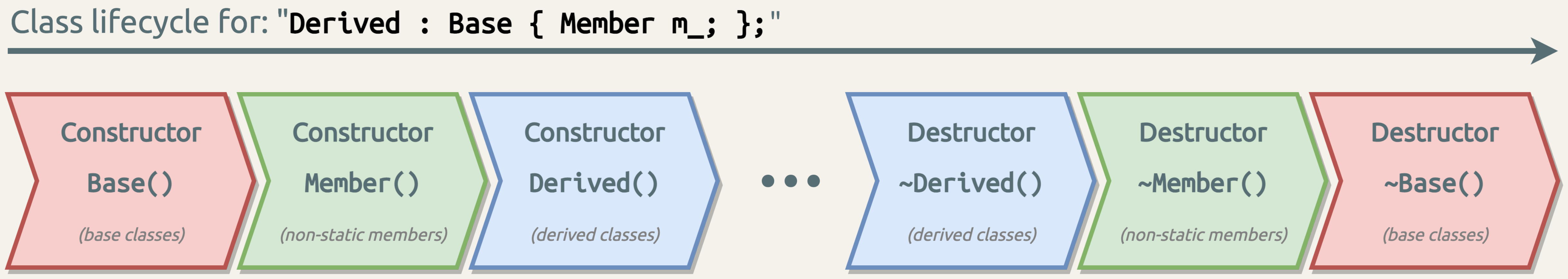
Class lifecycle for: "Derived : Base { Member m_; };"

SMFs and inheritance

```cpp
class Base   { /* ... */ };
class Member { /* ... */ };

class Derived : Base {
  Member m_;
};

Derived d;
```

Output:

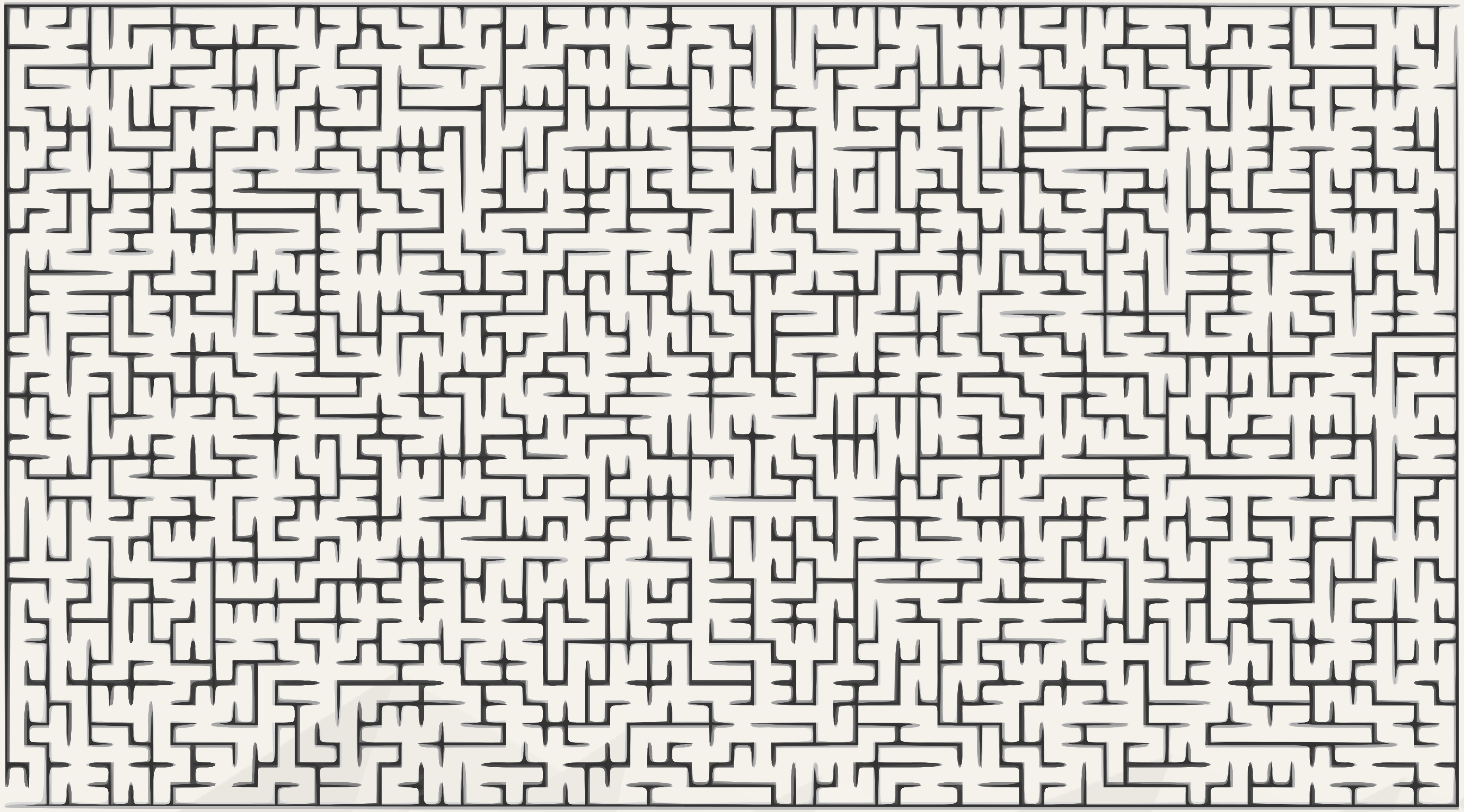
```
Base::Base()
Member::Member()
Derived::Derived()
Derived::~Derived()
Member::~Member()
Base::~Base()
```



1. Base classes (in order),
2. Non-static member variables (in order),
3. Derived class.

Declaration rules

When are SMFs generated?



Compiler implicitly declares

User declares

	Default constructor	Destructor	Copy constructor	Copy assignment	Move constructor	Move assignment
Nothing						
Any constructor						
Default constructor	User-declared					
Destructor		User-declared				
Copy constructor			User-declared			
Copy assignment				User-declared		
Move constructor					User-declared	
Move assignment						User-declared

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						
Default constructor	User-declared					
Destructor		User-declared				
Copy constructor			User-declared			
Copy assignment				User-declared		
Move constructor					User-declared	
Move assignment						User-declared

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					
Destructor		User-declared				
Copy constructor			User-declared			
Copy assignment				User-declared		
Move constructor					User-declared	
Move assignment						User-declared

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		User-declared				
Copy constructor			User-declared			
Copy assignment				User-declared		
Move constructor					User-declared	
Move assignment						User-declared

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			User-declared			
Copy assignment				User-declared		
Move constructor					User-declared	
Move assignment						User-declared

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					User-declared	
Move assignment						User-declared

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

“Not declared” vs “deleted”

Deleted members still participate in overload resolution,
“Not declared” members are just not there.

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

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

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

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

Declaration rules summary

Compiler “not declared” rules are easier to remember ❌

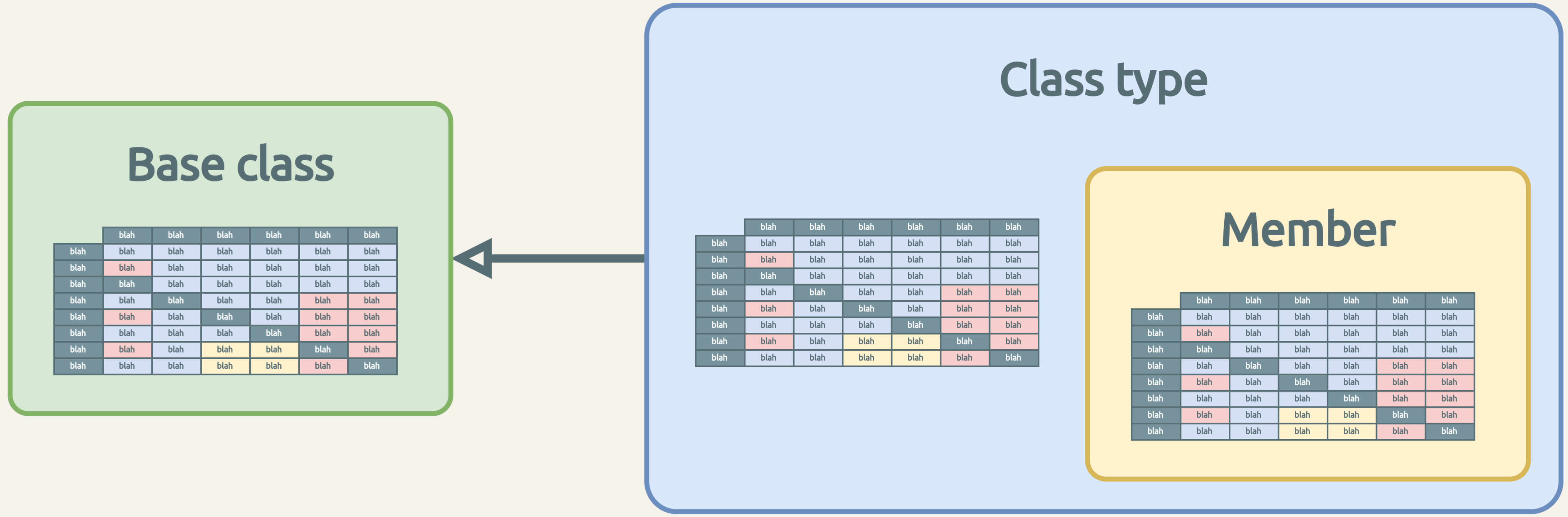
When user declares..	Result
Any other constructor	Default constructor **not declared**
Any copy/move/destructor	Move operations **not declared**
Any move operation	Copy operations **deleted**

Declaration rules summary

Compiler “not declared” rules are easier to remember ✗

When user declares..	Result
`T(*)`	`T()` **not declared**
`*(const T&)/*(T&&)/~T()`	`*(T&&)` **not declared**
`*(T&&)`	`*(const T&)` **deleted**

Declaration rules are transmissible



	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah

	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah

	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah
blah	blah	blah	blah	blah	blah	blah

Copy vs move and fallback

Consider this copy-only class:

```cpp
class Trillian {
public:
  Trillian() = default;

  Trillian(const Trillian& other)            = default;
  Trillian& operator=(const Trillian& other) = default;

  // Move operations are "not declared" as per the rules.
};
```

```cpp
Trillian x;

Trillian y{x};            // Will copy-construct.
Trillian z{std::move(x)}; // Will copy-construct..!?
```

🤔

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

		Compiler implicitly declares					
		Default constructor	Destructor	Copy constructor	Copy assignment	Move constructor	Move assignment
User declares	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 (deprecated)	Defaulted (deprecated)	Not declared (fallback enabled)	Not declared (fallback enabled)
	Copy constructor	Not declared	Defaulted	User-declared	Defaulted (deprecated)	Not declared (fallback enabled)	Not declared (fallback enabled)
	Copy assignment	Defaulted	Defaulted	Defaulted (deprecated)	User-declared	Not declared (fallback enabled)	Not declared (fallback enabled)
	Move constructor	Not declared	Defaulted	Deleted	Deleted	User-declared	Not declared (fallback disabled)
	Move assignment	Defaulted	Defaulted	Deleted	Deleted	Not declared (fallback disabled)	User-declared

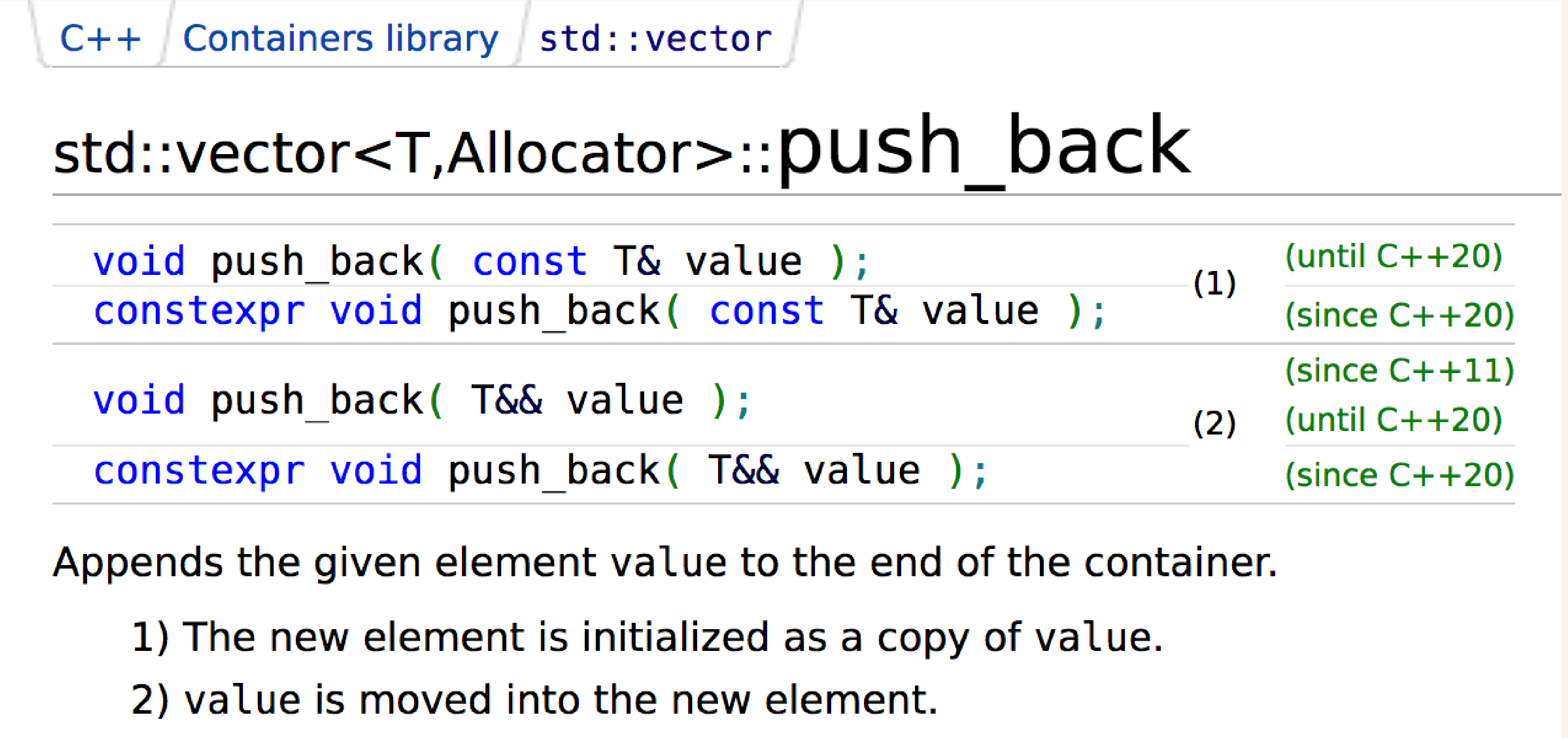
Compiler implicitly declares

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 (fallback enabled)	Not declared (fallback enabled)
Copy constructor	Not declared	Defaulted	User-declared	Defaulted	Not declared (fallback enabled)	Not declared (fallback enabled)
Copy assignment	Defaulted	Defaulted	Defaulted	User-declared	Not declared (fallback enabled)	Not declared (fallback enabled)
Move constructor	Not declared	Defaulted	Deleted	Deleted	User-declared	Not declared (fallback disabled)
Move assignment	Defaulted	Defaulted	Deleted	Deleted	Not declared (fallback disabled)	User-declared

Porting legacy code

```cpp
class Towel { // Implementation Somewhere Else (tm).
public:
  Towel();
  ~Towel();

  // Move operations are "not declared"..
  // ..because of the custom destructor!

  // ...
};

std::vector<Towel> v;

v.push_back(Towel{}); // Pre-C++11       : copy
                      // C++11 and later: copy
```



C++ / Containers library / std::vector

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

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

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

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

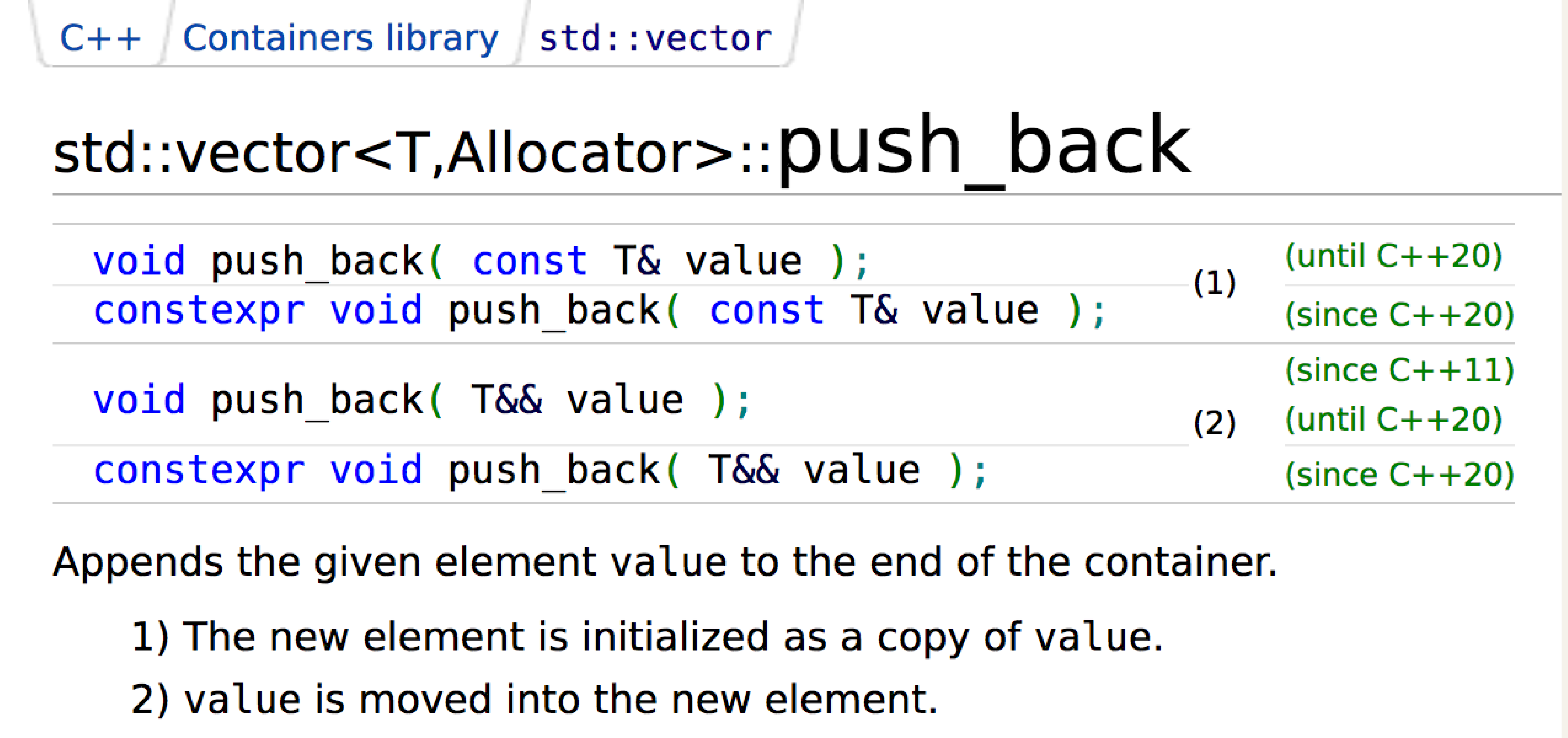
1) The new element is initialized as a copy of value.

2) value is moved into the new element.

C++11 and later will still copy – due to ~`Towel()` and copy fallback

Porting legacy code

```cpp
class Towel { // Implementation Somewhere Else (tm).
public:
  Towel();
  ~Towel();

  Towel(Towel&& other);
  Towel& operator=(Towel&& other);

  // ...
};

std::vector<Towel> v;

v.push_back(Towel{}); // C++11 and later: move
                      // If needed: restore copying as well.
```



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.

2) value is moved into the new element.

C++11 and later will now move using user-declared move operations

Rule of ...X?

- Rule of zero
- Rule of zero or all
- Rule of three
- Rule of four and a half
- Rule of five
- Rule of six
- Copy-only
- Move-only

🤯

Rule of zero

> *If you don’t have to implement SMFs…then don’t!*

```
class X {
public:
  // Let the compiler / library implementers / standards committee do all the work.
};
```

If custom SMFs are needed

Container/wrapper classes 🫙

Resource managing classes ⛲

Immobile classes 🏠

Use the rule of zero for everything else.

Container classes

Implement all SMFs, or: ‘rule of all’:

```cpp
class Container {
public:
  Container() noexcept;
  ~Container() noexcept;

  Container(const Container& other);
  Container& operator=(const Container& other) noexcept;

  Container(Container&& other);
  Container& operator=(Container&& other) noexcept;
};
```

Resource managing classes ⛲

Implement as move-only / uncopyable:

```cpp
class ResourceManager {
public:
  ResourceManager() noexcept;
  ~ResourceManager() noexcept;

  ResourceManager(ResourceManager&& other) noexcept;
  ResourceManager& operator=(ResourceManager&& other) noexcept;
};
```

The rules/compiler will take care of the rest!

Immobile classes 🏠

Implement as non-copyable, non-movable:

```cpp
class Immobile {
public:
  Immobile(const Immobile& other)            = delete;
  Immobile& operator=(const Immobile& other) = delete;
};
```

The rules/compiler will take care of the rest!

The rule of zero or all

> *As a general one-rule-only policy the rule of zero or all can be followed.*

Trigger for 'all': any copy/move ops or ~T()

Some last notes ☝

- Try to have a default constructor,
- Don’t split copy operations and move operations,
- Don’t build copy-only types, they are weird.

Implementation guidelines

What are the options again?

- Do nothing,
- User-declare the SMF:
  - `SMF(..) { /* ..custom implementation.. */ }`
  - `SMF(..) = default;`
  - `SMF(..) = delete;`

Constructor guidelines

General wisdom ☝️

- Use the member initialization list,
- Never call `virtual` functions from `T()`/`~T()`,
- Make sure your object is in a specified state after `T()`,
- Mark a constructor taking only one argument `explicit`.

Explicit constructor

> *Make any constructor that takes only one argument an* **`explicit`** *constructor.*

Unless you intend to write a converting constructor.

Explicit constructor

Perhaps not what was intended.. 🤔

```cpp
struct Ford {
  Ford(int value) {} // Converting constructor.
};

void func(Ford arg);

func(42); // Will implicitly convert '42' to Ford..
```

Explicit constructor

Fixed 🛠️

```cpp
struct Ford {
  explicit Ford(int value) {} // Explicit constructor.
};

void func(Ford arg);

//func(42);      // Compiler error!
func(Ford{42}); // OK, explicit and safe.
```

Member initializer list

Consider this example:

```cpp
class Arthur {
  std::string value_;

public:
  Arthur(std::string_view value) {
    value_ = value;
  }
};
```

1. Default-constructs `value_` first,
2. Calls assignment operator second.

Member initializer list

More efficient version:

```cpp
struct Arthur {
  std::string value_;

public:
  Arthur(std::string_view value)
    : value_{value} {
  }
};
```

Direct-initializes `value_`

Member variable initializations

What about this example?

```cpp
class Arthur {
  int value1_ = 0;
  int value2_ = 0;

public:
  Arthur()
    : value1_{0} {
  }
};

Arthur d;
```

Question 1: will `value1_` be initialized twice?

Question 2: how will `value2_` be initialized?

Member variable initializations

No constructor needed

```cpp
class Arthur {
  // Default member initializer will be used in generated ctors.
  int value1_ = 0;
  int value2_ = 0;
};

Arthur d;
```

Member initializer list

Strict initialization order is NOT enforced.. 😞

```cpp
class Arthur {
  int value1_;
  int value2_;

public:
  Arthur(int value1, int value2)
    : value2_{value2 + value1_},
      value1_{value1} {
  }
};
```

A warning message *may* be generated if configured..

Member initializer list

Use the member initializer list!

Turn on compiler warnings: `-Wreorder` */* `/w15038`

(and look at them..or better even, use `-Werror`/`/WX`)

Destructor guidelines

General wisdom ☝

- Never call `virtual` functions from `T()/~T()`,
- Make destructor `virtual` if applicable,
- Don’t let exceptions leave destructors.

virtual destructors

> *A base class must have a* ***virtual*** *destructor.*

```cpp
struct Base            { /* ... */ };
struct Derived : Base { /* ... */ };

{
  auto ptr = std::make_unique<Base>(Derived{});
} // Pointer 'ptr' will go out of scope here and destruct via Base-type.
```

virtual destructors

```cpp
struct Base {
  virtual ~Base();
};

struct Derived : Base {
  ~Derived() override; // 'override' specifier was added in C++11.
};

{
  auto ptr = std::make_unique<Base>(Derived{});
} // All is safe now.
```

Destructors and exceptions

> *Don’t throw exceptions from destructors!*

🤖 `terminate()` will be called!

Look into `std::uncaught_exception` if you want to detect active exceptions

Copy operations

General wisdom ☝️

- Always copy all parts of a class,
- If applicable and needed, call the base class,
- Handle self-assignment,
- Return a reference to `*this`,
- Provide (strong) exception-safety,
- If possible, mark as `noexcept`.

Move operations

General wisdom ☝️

- Always move all parts of a class,
- If applicable and needed, call the base class,
- Handle self-assignment,
- Return a reference to `*this`,
- Provide (strong) exception-safety,
- If possible, mark as `noexcept`,
- Leave a moved-from object in a valid state.

Moved-from object state

```cpp
class Dolphin {
  std::unique_ptr<Resource> resource_;
  bool                      initialized_{false};

  void deinit_resource(); // Deinitialize resource (if applicable).

public:
  Dolphin() = default;

  Dolphin(Dolphin&& other) noexcept
    : resource_{std::move(other.resource_)},
      initialized_{other.initialized_} {
  }

  Dolphin& operator=(Dolphin&& other) noexcept {
    // ..similar to move ctor, including a self-assignment check.
  }

  [[nodiscard]] bool init(); // Initialize resource.
};
```

```cpp
Dolphin a;

// Set up resource.
if (!a.init()) {
  // ..error handling..
}

// ..do stuff with 'a'..

auto b{std::move(a)};

// ..do stuff with 'b'..

// ..what about 'a' now?
```

Post-move state of `a` is invalid!

(it's still marked `initialized_ = true`..)

Moved-from object state

```cpp
class Dolphin {
  std::unique_ptr<Resource> resource_;
  bool                      initialized_{false};

  void deinit_resource(); // Deinitialize resource (if applicable).

public:
  Dolphin() = default;

  Dolphin(Dolphin&& other) noexcept
    : resource_{std::move(other.resource_)},
      initialized_{other.initialized_} {
    other.initialized_ = false;
  }

  Dolphin& operator=(Dolphin&& other) noexcept {
    // ..similar to move ctor, including a self-assignment check.
  }

  [[nodiscard]] bool init(); // Initialize resource.
};
```

```cpp
Dolphin a;

// Set up resource.
if (!a.init()) {
  // ..error handling..
}

// ..do stuff with 'a'..

auto b{std::move(a)};

// ..do stuff with 'b'..

// ..what about 'a' now?
```

Post-move state of `a` is OK now.

RAII

RAII == Resource Acquisition Is Initialization

```
std::unique_ptr<>/std::shared_ptr<>/std::lock_guard<>/...
```

Constructor/destructor pattern

```cpp
void work(std::stop_token s) {
  while (!s.stop_requested()) {
    busy_wait_for_work();

    try {
      std::lock_guard lock{data_mutex};

      // ..do dangerous stuff with 'data' that may throw..

    } catch (...) {
      // ..deal with exceptions..
    }
  }
}

int main() {
  std::jthread worker{work};

  // ...
}
```

Object **lock** will:

- Lock the mutex upon construction,
- Unlock the mutex upon destruction.

Constructor/destructor pattern

```cpp
class VaultAccessor {
  Vault &vault_; // Initialized in ctor.

public:
  VaultAccessor() {
    vault_.unlock();
  }

  ~VaultAccessor() {
    vault_.lock();
  }
};
```

```cpp
{
  VaultAccessor v; // Constructor unlocks the vault.

  // ..get data from the vault..
} // Destructor locks the vault.
```

```cpp
class TemporaryFile {
public:
  TemporaryFile() {
    // ..create temporary file, set name..
  }

  ~TemporaryFile() {
    // ..remove temporary file..
  }

  [[nodiscard]] const std::string &name() const;
};
```

```cpp
{
  TemporaryFile tmp; // Constructor creates temporary file.

  // ..use 'tmp' for scratch data..
} // File is cleaned up at destruction.
```

Unit-test your SMFs!

```cpp
X x;                 // Test default construction.
X x{...};            // Test alternative construction.
X y{x};              // Test copy construction.
X y{std::move(x)}; // Test move construction.
y = x;               // Test copy assignment.
y = std::move(x);  // Test move assignment.
```

Enable compiler sanitizers! E.g. ASAN, UB, ..

Run static analyzers!

Unit-test your SMFs!

Also test for compile-time guarantees

Trivial copy + move:

```cpp
#include <array>

using X = std::array<float, 8>;
```

```cpp
static_assert(std::is_trivially_destructible<X>{});
static_assert(std::is_trivially_default_constructible<X>{});
static_assert(std::is_trivially_copy_constructible<X>{});
static_assert(std::is_trivially_copy_assignable<X>{});
static_assert(std::is_trivially_move_constructible<X>{});
static_assert(std::is_trivially_move_assignable<X>{});
```

Non-trivial, **noexcept** copy + move:

```cpp
#include <memory>

using X = std::shared_ptr<int>;
```

```cpp
static_assert(std::is_nothrow_destructible<X>{});
static_assert(std::is_nothrow_default_constructible<X>{});
static_assert(std::is_nothrow_copy_constructible<X>{});
static_assert(std::is_nothrow_copy_assignable<X>{});
static_assert(std::is_nothrow_move_constructible<X>{});
static_assert(std::is_nothrow_move_assignable<X>{});
```

Unit-test your SMFs!

Also test for compile-time guarantees

Move-only type:

```cpp
#include <thread>

using X = std::jthread;
```

```cpp
static_assert( std::is_nothrow_destructible<X>{});
static_assert( std::is_nothrow_default_constructible<X>{});
static_assert(!std::is_copy_constructible<X>{});
static_assert(!std::is_copy_assignable<X>{});
static_assert( std::is_nothrow_move_constructible<X>{});
static_assert( std::is_nothrow_move_assignable<X>{});
```

Immobile type:

```cpp
#include <iostream>

using X = std::ostream;
```

```cpp
static_assert( std::is_nothrow_destructible<X>{});
static_assert(!std::is_default_constructible<X>{});
static_assert(!std::is_copy_constructible<X>{});
static_assert(!std::is_copy_assignable<X>{});
static_assert(!std::is_move_constructible<X>{});
static_assert(!std::is_move_assignable<X>{});
```

(`std::ostream` has a **protected** move ctor)

Thoughts and ponderings

Do we really need SMFs?

How do other languages deal with construction/destruction/copy/move?

```rust
pub mod example {
    #[derive(Clone)]
    pub struct Configuration {
        pub setting1: f32,
        pub setting2: String,
    }

    #[derive(Clone)]
    pub struct Algorithm {
        config: Configuration,
    }

    impl Algorithm {
        pub fn new(config: Configuration) -> Self {
            Algorithm { config }
        }

        pub fn run(self: &mut Self) {
            // ..do algorithmic stuff..
        }
    }
}
```

```rust
use example::{Algorithm, Configuration};

fn main() {
    let c = Configuration {
        setting1: 1.2,
        setting2: "resource.json".to_owned(),
    };

    let a = Algorithm::new(c);
    let b = a; // Will move 'a' into 'b'.
    let c = b.clone();
}
```

If a “destructor” of sorts is needed:

```rust
impl Drop for Algorithm {
    fn drop(&mut self) {
        // ...
    }
}
```

Factory method

```cpp
class Sensor {
  unsigned int id_ = {};

public:
  [[nodiscard]] static Sensor create(unsigned int id) {
    return Sensor{id};
  }

  [[nodiscard]] unsigned int id() const {
    return id_;
  }

private:
  explicit Sensor(unsigned int id)
    : id_{id} {
  }
};
```

```cpp
//Sensor x{42}; // Direct construction is impossible.

auto x = Sensor::create(42);

const auto id = x.id();
```

How to deal with factory failure?

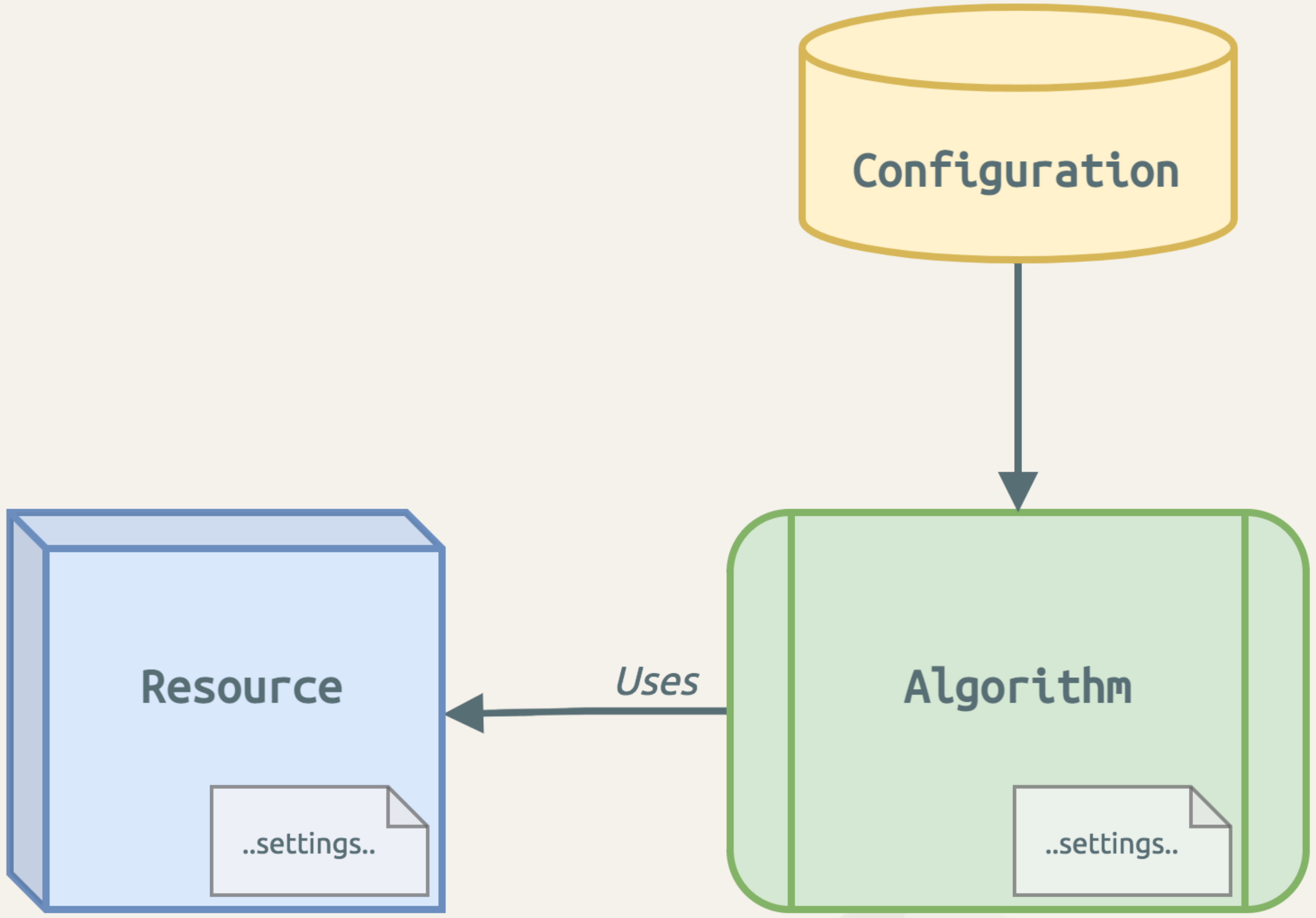
Factory method

```cpp
class Sensor {
  unsigned int id_ = {};

public:
  using CreateResult = std::expected<Sensor, std::string>;

  [[nodiscard]] static CreateResult create(unsigned int id) {
    if (id > 128) {
      return std::unexpected{"ID must be lower than 128"};
    }

    return Sensor{id};
  }

  [[nodiscard]] unsigned int id() const {
    return id_;
  }

private:
  explicit Sensor(unsigned int id)
    : id_{id} {
  }
};
```

```cpp
//Sensor x{42}; // Direct construction is impossible.

const auto print_id = [](Sensor t) {
  std::cout << std::format("ID = {}\n", t.id());
};

auto x = Sensor::create(42);

x.transform(print_id);
```

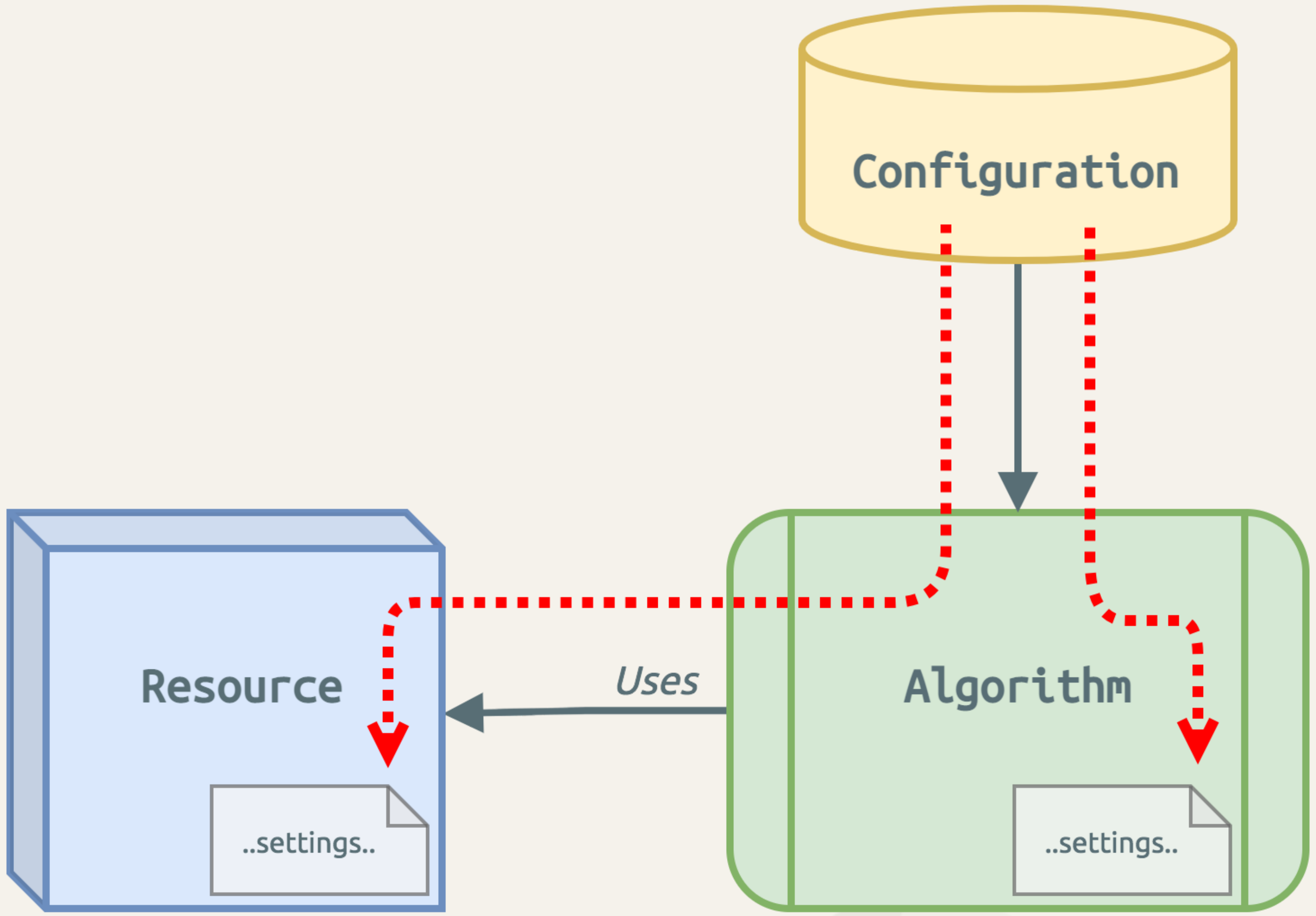
Uses `std::expected` from C++23

Class design considerations



```cpp
/// Algorithm configuration settings.
struct Configuration {
  float       setting1 = 1.2f;
  std::string setting2 = "resource.json";
};

// Move-only resource.
class Resource {
  std::string config_file_;

  // ...
};

// The algorithm class used for processing by the user.
class Algorithm {
  Configuration config_;
  Resource      resource_;

  // ...
};
```

Class design considerations



```cpp
/// Algorithm configuration settings.
struct Configuration {
  float       setting1 = 1.2f;
  std::string setting2 = "resource.json";
};

// Move-only resource.
class Resource {
  std::string config_file_;

  // ...
};

// The algorithm class used for processing by the user.
class Algorithm {
  Configuration config_;
  Resource      resource_;

  // ...
};
```

Version 1: hard to misuse

```cpp
/// Algorithm configuration settings.
struct Configuration {
  float       setting1 = 1.2f;
  std::string setting2 = "resource.json";

  [[nodiscard]] bool is_valid() const {
    // ..check configuration validity..
  }
};

/// Move-only resource.
class Resource {
  std::string config_file_;

public:
  Resource(std::string_view config_file)
    : config_file_{config_file} {
  }

  Resource(Resource&&)            = default;
  Resource& operator=(Resource&&) = default;

  // Copy operations are 'deleted' as per the rules.
};
```

```cpp
class Algorithm {
  Configuration config_;
  Resource      resource_;

public:
  Algorithm(const Configuration& config)
    : config_{config},
      resource_{config_.setting2} {
    if (!config_.is_valid()) {
      throw std::runtime_error{"Invalid configuration"};
    }
  }

  void run() { /* ..do algorithmic stuff.. */ }
};
```

```cpp
try {
  Configuration config;
  Algorithm algo{config};

  algo.run();
} catch (const std::exception& error) {
  std::cerr << "Error: " << error.what() << '\n';
}
```

Version 2: deferred initialization

```cpp
/// Algorithm configuration settings.
struct Configuration {
  float       setting1 = 1.2f;
  std::string setting2 = "resource.json";

  [[nodiscard]] bool is_valid() const;
};

/// Move-only resource.
class Resource {
public:
  Resource(std::string_view config_file);
};
```

```cpp
try {
  Configuration config;
  Algorithm algo{config};

  algo.init();
  algo.run();
} catch (const std::exception& error) {
  std::cerr << "Error: " << error.what() << '\n';
}
```

```cpp
class Algorithm {
  Configuration             config_;
  std::unique_ptr<Resource> resource_;

public:
  Algorithm(const Configuration& config)
    : config_{config} {
    if (!config_.is_valid()) {
      throw std::runtime_error{"Invalid configuration"};
    }
  }

  void init() {
    if (resource_) throw std::logic_error{"..."};

    resource_ = std::make_unique<Resource>(config_.setting2);
  }

  void run() {
    if (!resource_) throw std::logic_error{"..."};

    // ..do algorithmic stuff..
  }
};
```

Version 3: reinitialization support

```cpp
/// Algorithm configuration settings.
struct Configuration;

/// Move-only resource.
class Resource;
```

```cpp
try {
  Algorithm algo;

  Configuration config1;
  algo.init(config1);

  algo.run();

  Configuration config2;
  algo.init(config2);

  algo.run();
} catch (const std::exception& error) {
  std::cerr << "Error: " << error.what() << '\n';
}
```

```cpp
class Algorithm {
  Configuration                config_;
  std::unique_ptr<Resource> resource_;
  std::shared_mutex         resource_mutex_;

public:
  void init(const Configuration& config) {
    if (!config.is_valid()) {
      throw std::runtime_error{"Invalid configuration"};
    }

    config_ = config;

    std::unique_lock lock{resource_mutex_};
    resource_ = std::make_unique<Resource>(config_.setting2);
  }

  void run() {
    std::shared_lock lock{resource_mutex_};
    if (!resource_) throw std::logic_error{"..."};

    // ..do algorithmic stuff..
  }
};
```

Version 4: re-simplification

```cpp
/// Algorithm configuration settings.
struct Configuration;

/// Move-only resource.
class Resource;
```

```cpp
try {
  std::unique_ptr<Algorithm> algo;

  Configuration config1;
  algo = std::make_unique<Algorithm>(config1);

  algo->run();

  Configuration config2;
  algo = std::make_unique<Algorithm>(config2);

  algo->run();
} catch (const std::exception& error) {
  std::cerr << "Error: " << error.what() << '\n';
}
```

```cpp
class Algorithm {
  Configuration config_;
  Resource      resource_;

public:
  Algorithm(const Configuration& config)
    : config_{config},
      resource_{config_.setting2} {
    if (!config_.is_valid()) {
      throw std::runtime_error{"Invalid configuration"};
    }
  }

  void run() {
    // ..do algorithmic stuff..
  }
};
```

Back at version 1! 😃

Version 5: split stage types

```cpp
/// Algorithm configuration settings.
struct Configuration;

/// Move-only resource.
class Resource;
```

```cpp
try {
  std::unique_ptr<Algorithm> algo;

  Configuration config1;
  AlgorithmInit init1{config1};
  algo = std::make_unique<Algorithm>(std::move(init1));
  algo->run();

  Configuration config2;
  AlgorithmInit init2{config2};
  algo = std::make_unique<Algorithm>(std::move(init2));
  algo->run();
} catch (const std::exception& error) {
  std::cerr << "Error: " << error.what() << '\n';
}
```

Add more stages as needed.

```cpp
class AlgorithmInit {
  Configuration config_; // Perhaps store other stuff too.

public:
  AlgorithmInit(const Configuration& config)
    : config_{config} {
    if (!config_.is_valid()) {
      throw std::runtime_error{"Invalid configuration"};
    }
  }

  friend class Algorithm;
};

class Algorithm {
  AlgorithmInit init_;
  Resource      resource_;

public:
  Algorithm(AlgorithmInit&& init)
    : init_{std::move(init)},
      resource_{init_.config_.setting2} {
  }

  void run() { /* ..do algorithmic stuff.. */ }
};
```

End

Thank you 😃

And many thanks for excellent online resources from Howard Hinnant, Jonathan Müller and Matt Godbolt + CE team!

💬❓

🔗 github.com/krisvanrens

All emoji in this presentation are part of the Twemoji set, licensed under CC-BY 4.0.
All other images are mine, unless specified otherwise.

Extra slides

Defaulting or hiding an SMF

(pre-C++11)

- Request default implementation: `T(){}`
- ‘Hiding’ an SMF: declaration in `private` section

`constexpr` constructors

> *The constructors with a `constexpr` specifier make their type a LiteralType.*

Destructors and exceptions

Don’t throw exceptions from destructors!

- `terminate()` is called when another exception is already active,
- Since C++11, destructors are **`noexcept`** by default,
- `terminate()` is called *always* (C++11 and onwards),
- Also, a **`throw`** would break completion of the destructor.

Case study

Destructors and exceptions

Designing a RAII type with strong exception guarantee

std::uncaught_exceptions?

```cpp
struct Transaction {
  // ...
};
```

How to safely roll back a transaction?

```cpp
struct X {
  ~X() {
    try {
      Transaction t;
      // ...
    } catch (...) {}
  }
};

int main() {
  try {
    X x;
  } catch (...) {}
}
```

std::uncaught_exceptions?

```cpp
struct Transaction {
  ~Transaction() {
    if (/* ...uuhhh... */) {
      // Rollback transaction!
    } else {
      // No need to rollback -- all good.
    }
  }
};
```

```cpp
struct X {
  ~X() {
    try {
      Transaction t;
      // ...
    } catch (...) {}
  }
};

int main() {
  try {
    X x;
  } catch (...) {}
}
```

std::uncaught_exceptions?

```cpp
struct Transaction {
  Transaction()
    : num_exceptions{std::uncaught_exceptions()} {}

  ~Transaction() {
    if (std::uncaught_exceptions() != num_exceptions) {
      puts("Rollback transaction!");
    } else {
      puts("No need to rollback -- all good.");
    }
  }

  int num_exceptions = {};
};
```

```cpp
struct X {
  ~X() {
    try {
      Transaction t;
      // ...
    } catch (...) {}
  }
};

int main() {
  try {
    X x;
  } catch (...) {}
}
```

SMFs and inheritance

Multiple inheritance and multiple non-static class members:

```cpp
struct Derived
  : Base1,      // Base class initialization in left-to-right order 1..N
    Base2,
    // ...
    BaseN
{
  Member m1_; // Initialization of non-static members in declaration order 1..M
  Member m2_;
  // ...
  Member mM_;
};
```

Multiple destructors

C++20 adds **prospective destructors** for allowing *concepts*

> *A type that can be a **trivial type** (depending on the `requires` clause) or not, can have multiple destructors on which overload resolution is performed.*

(also holds for copy constructors)

ISO C++ draft § class.dtor

Deleting SMFs

Don’t be overzealous at **delete**’ing members, it may break your code:

```cpp
class X {
public:
  // Default ctor must be manually added as per the rules.
  X() = default;

  X(const X&)            = default;
  X& operator=(const X&) = default;
};
```

```cpp
class Y {
public:
  // Default ctor must be manually added as per the rules.
  Y() = default;

  Y(const Y&)            = default;
  Y& operator=(const Y&) = default;
  Y(Y&&)                 = delete;
  Y& operator=(Y&&)      = delete;
};
```

```cpp
X factory() {
  return X{}; // Will always fall back to copy.
}

X x = factory();
```

```cpp
Y factory() {
  return Y{}; // Will break under C++14.
}

Y y = factory();
```

Empty base optimization

```cpp
#include <cassert>

struct Base {};

struct Derived : Base {
  int value_;
};

int main() {
  // Every object is guaranteed to have a unique address.
  assert(sizeof(Base) >= 1);

  assert(sizeof(Derived) == sizeof(int)); // EBO.
}
```

Why make `delete`’ed functions `public`?

- Because accessibility is checked before definition.
- The wrong error can be triggered in the compilation.

Example: deleted and private

```cpp
1 class X {
2 private:
3   X() = delete;
4 };
5
6 int main() {
7   X x;
8 }
```

```
<source>: In function 'int main()':
<source>:3:3: error: 'X::X()' is private
   X() = delete;
   ^
<source>:7:5: error: within this context
   X x;
     ^
```

Example: deleted and public

```cpp
1  class X {
2  public:
3    X() = delete;
4  };
5
6  int main() {
7    X x;
8  }
```

```
<source>: In function 'int main()':
<source>:7:5: error: use of deleted function 'X::X()'
   X x;
     ^
<source>:3:3: note: declared here
   X() = delete;
   ^
```

The copy and swap idiom

Use this idiom for implementing exception-safe copy/move operations.

Copy and swap (1)

```cpp
class X {
public:
  X() { /* ..initialize r_.. */ }

  // ???

private:
  Resource r_;
}
```

```cpp
int main() {
  X x;

  X y{x};
  y = x;
}
```

Copy and swap (2)

```cpp
class X {
public:
  X() { /* ..initialize r_.. */ }

  X(const X& other) {
    r_ = other.r_;
  }

  // ???

private:
  Resource r_;
}
```

```cpp
int main() {
  X x;

  X y{x};
  y = x;
}
```

Copy and swap (3)

```cpp
class X {
public:
  X() { /* ..initialize resource.. */ }

  X(const X& other) { /* ... */ }

  X& operator=(X other) { // Take 'other' by value.
    swap(*this, other);
    return *this;
  }

  friend void swap(X& x1, X& x2);

private:
  Resource r_;
};
```

```cpp
void swap(X& x1, X& x2) {
  std::swap(x1.r_, x2.r_);
}
```

```cpp
int main() {
  X x;

  X y{x};
  y = x;
}
```

Copy and swap (4)

```cpp
class X {
public:
  X() { /* ..initialize resource.. */ }

  X(const X& other)     { /* ... */ }
  X& operator=(X other) { /* ... */ }

  X(X&& other) // 'delegating' constructor..
    : X() {    // ..delegates to 'X()'.
    swap(*this, other);
  }

  friend void swap(X& x1, X& x2);

private:
  Resource r_;
};
```

```cpp
void swap(X& x1, X& x2) {
  std::swap(x1.r_, x2.r_);
}
```

```cpp
int main() {
  X x;

  X y{x};
  y = x;

  X z{std::move(y)};
  z = std::move(x);
}
```

Copy and swap (5)

```cpp
class X {
public:
  X() { /* ..initialize resource.. */ }

  X(const X& other) {
    r_ = other.r_;
  }

  X& operator=(X other) { // Take 'other' by value.
    swap(*this, other);
    return *this;
  }

  X(X&& other)
    : X() {
    swap(*this, other);
  }

  friend void swap(X& x1, X& x2);

private:
  Resource r_;
};
```

```cpp
void swap(X& x1, X& x2) {
  std::swap(x1.r_, x2.r_);
}
```

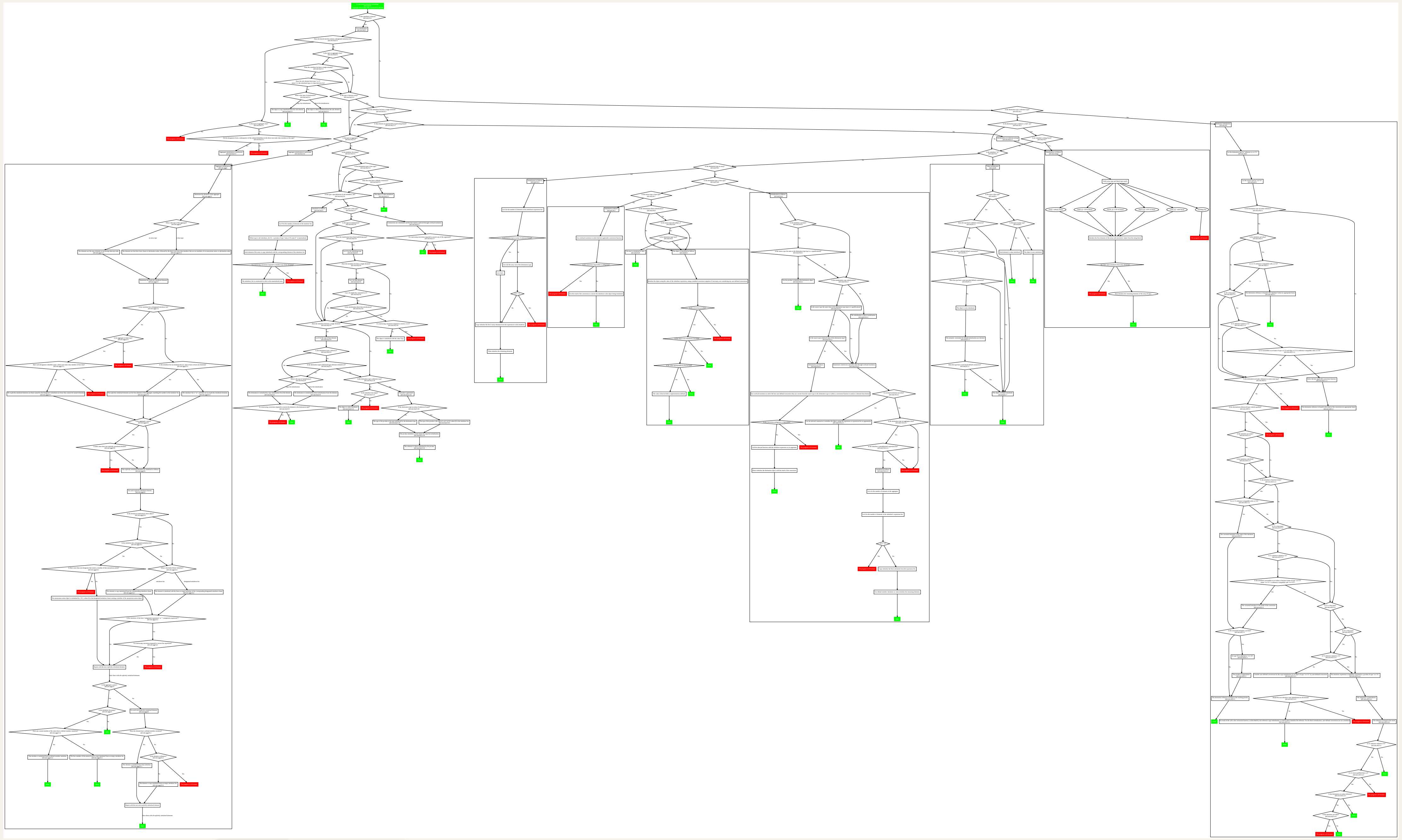
```cpp
int main() {
  X x;

  X y{x};
  y = x;

  X z{std::move(y)};
  z = std::move(x);
}
```

Uncopyable type

```cpp
class Uncopyable {
protected:
  Uncopyable()  = default;
  ~Uncopyable() = default;

public:
  Uncopyable(Uncopyable&&)            = default;
  Uncopyable& operator=(Uncopyable&&) = default;

  // Copy operations are deleted as per the rules.
};
```

```cpp
class X : private Uncopyable {
  // ...
};

int main() {
  X a;

  //X b{a};           // Will fail to compile.
  X b{std::move(a)}; // Will move-construct.

  X c;
  //c = b;          // Will fail to compile.
  c = std::move(b); // Will move-assign.
}
```

Initialization in C++20



From: https://github.com/randomnetcat/cpp_initialization