Back To Basics
Value Semantics

KLAUS IGLBERGER

Cppcon
The C++ Conference

2022
September 12th-16th

C++ Trainer/Consultant

Author of the blaze C++ math library

(Co-)Organizer of the Munich C++ user group

Chair of the CppCon B2B track

Email: klaus.iglberger@gmx.de

Klaus Iglberger

UNIVERSITY OF CAMBRIDGE

UNIVERSITY OF CAMBRIDGE

Title:	**Communication and control in distributed computer systems.**		
Author:	Stroustrup, B.	ISNI:	0000 0001 3488 5095
Awarding Body:	University of Cambridge		
Current Institution:	University of Cambridge		
Date of Award:	1979		
Availability of Full Text:	Full text unavailable from EThOS. Please contact the current institution's library for further details.		
Abstract:			
No abstract available			
Supervisor:	Not available	**Sponsor:**	Not available
Qualification Name:	Thesis (Ph.D.)	**Qualification Level:**	Doctoral
EThOS ID:	uk.bl.ethos.474113	**DOI:**	Not available

Picture taken from W

simula

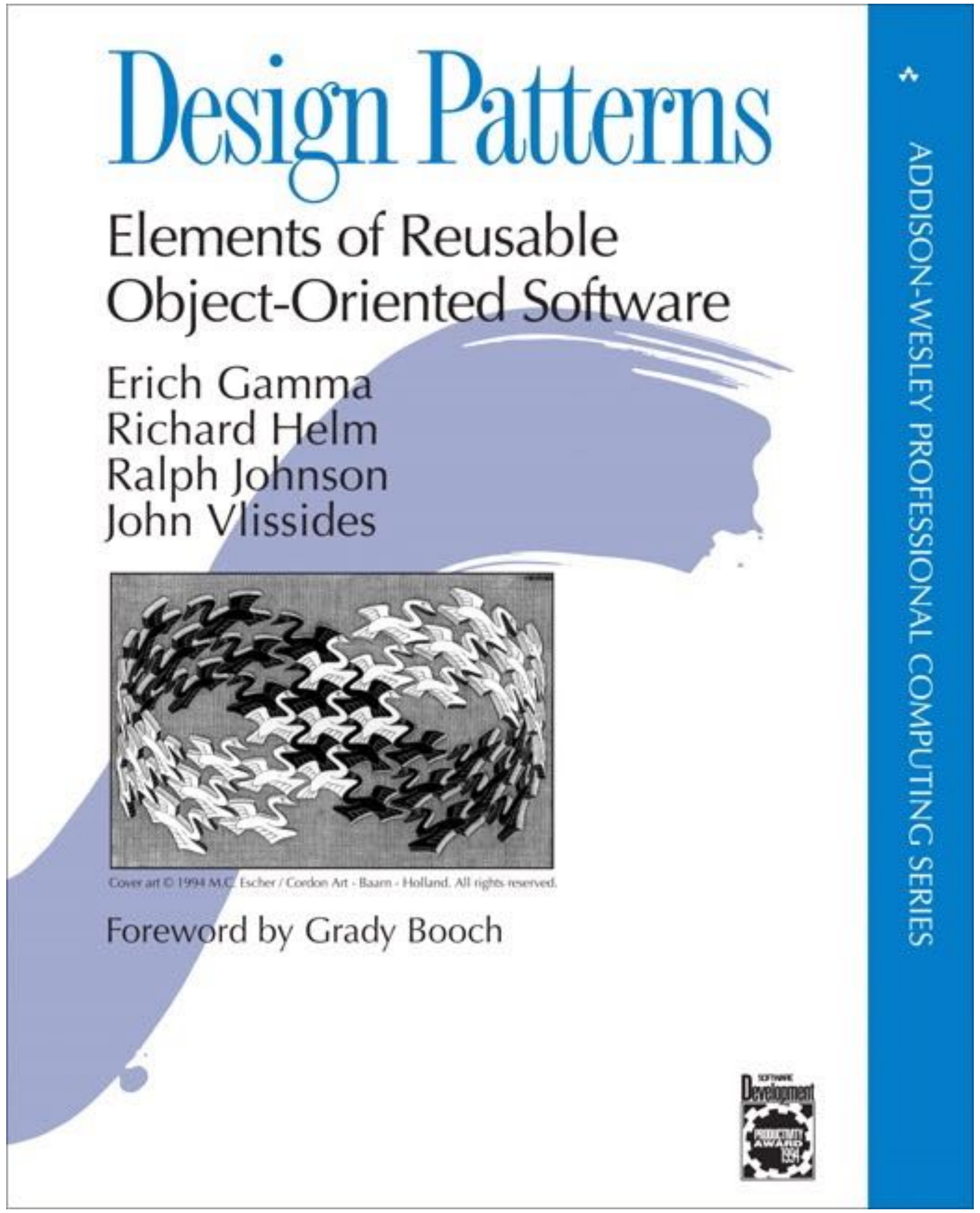

hilser
Ole-Johan Dahl
Kristen Nygaard

simula

```
Begin
   Class Glyph;
      Virtual: Procedure print Is Procedure print;;
   Begin
   End;

   Glyph Class Char (c);
      Character c;
   Begin
      Procedure print;
        OutChar(c);
   End;

   Glyph Class Line (elements);
      Ref (Glyph) Array elements;
   Begin
      Procedure print;
      Begin
         Integer i;
         For i:= 1 Step 1 Until UpperBound (elements, 1) Do
            elements (i).print;
         OutImage;
      End;
   End;

   Ref (Glyph) rg;
   Ref (Glyph) Array rgs (1 : 4);

   ! Main program;
   rgs (1):- New Char ('A');
   rgs (2):- New Char ('b');
   rgs (3):- New Char ('b');
   rgs (4):- New Char ('a');
   rg:- New Line (rgs);
   rg.print;
End;
```

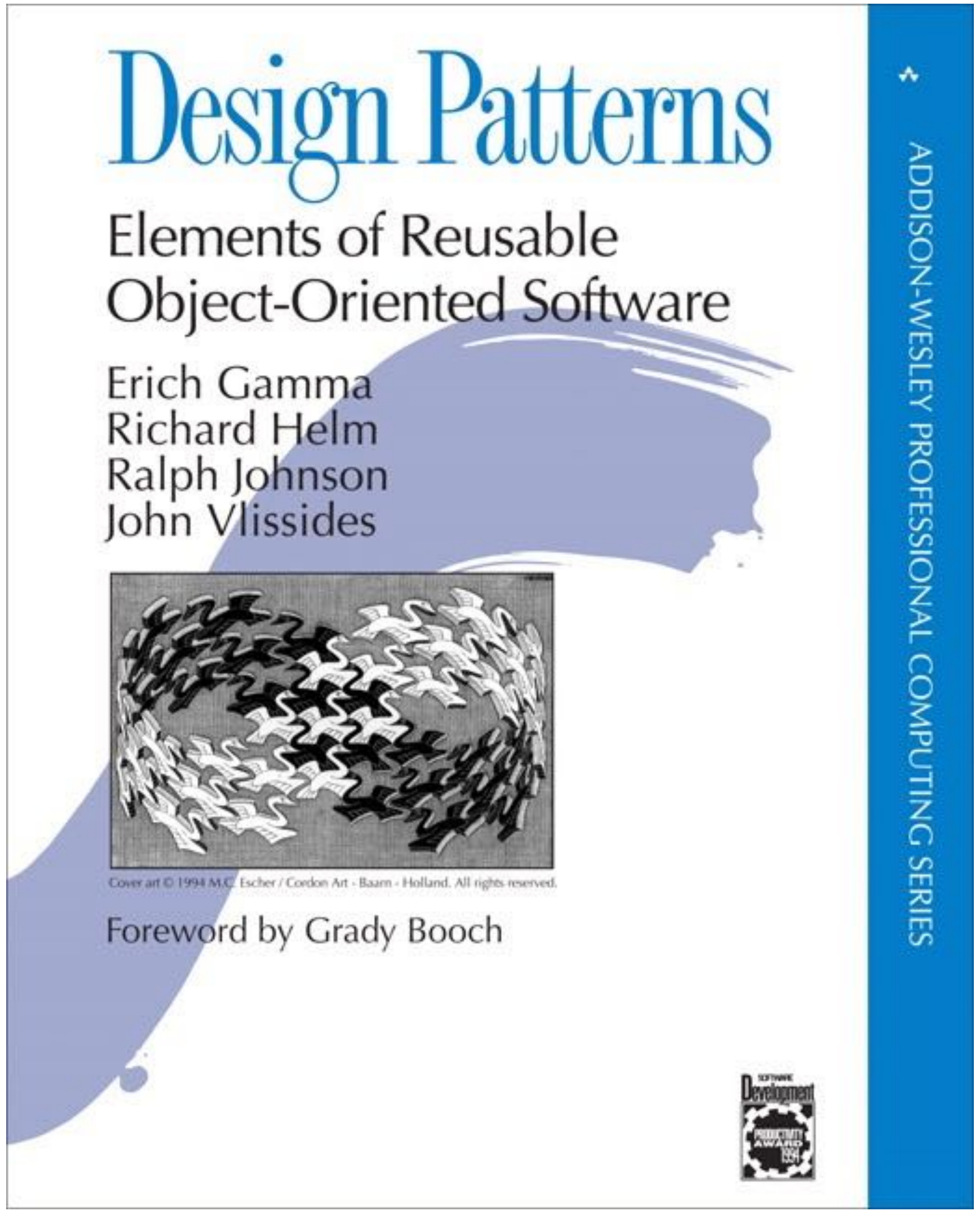

THE
C
PROGRAMMING
LANGUAGE
simula
C++
Bell Laboratories

The Source of Classic OO Design Patterns

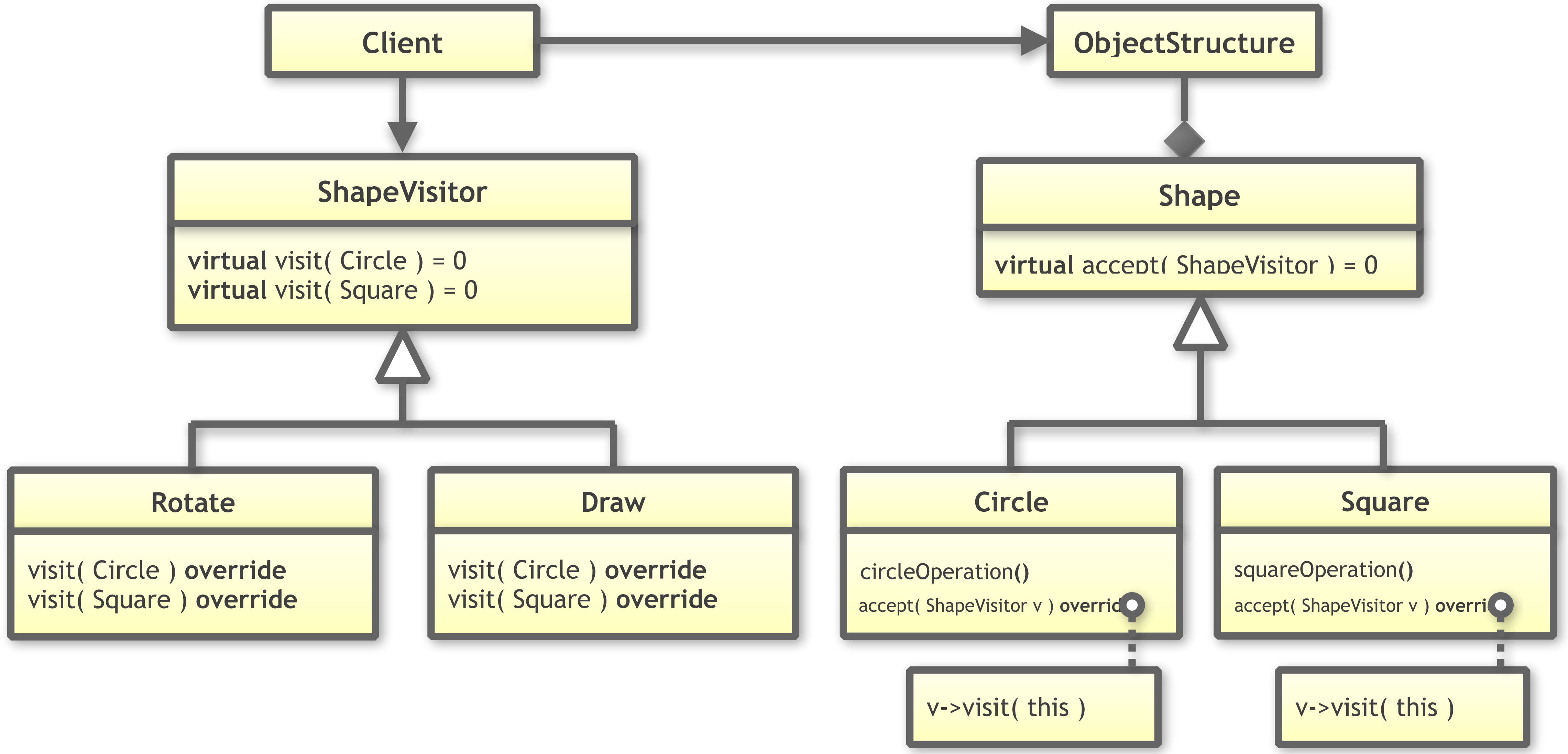


- The Gang of Four (GoF) book
- Published in 1994
- Source of 23 of the most commonly used design patterns
- Almost all design patterns are based on inheritance

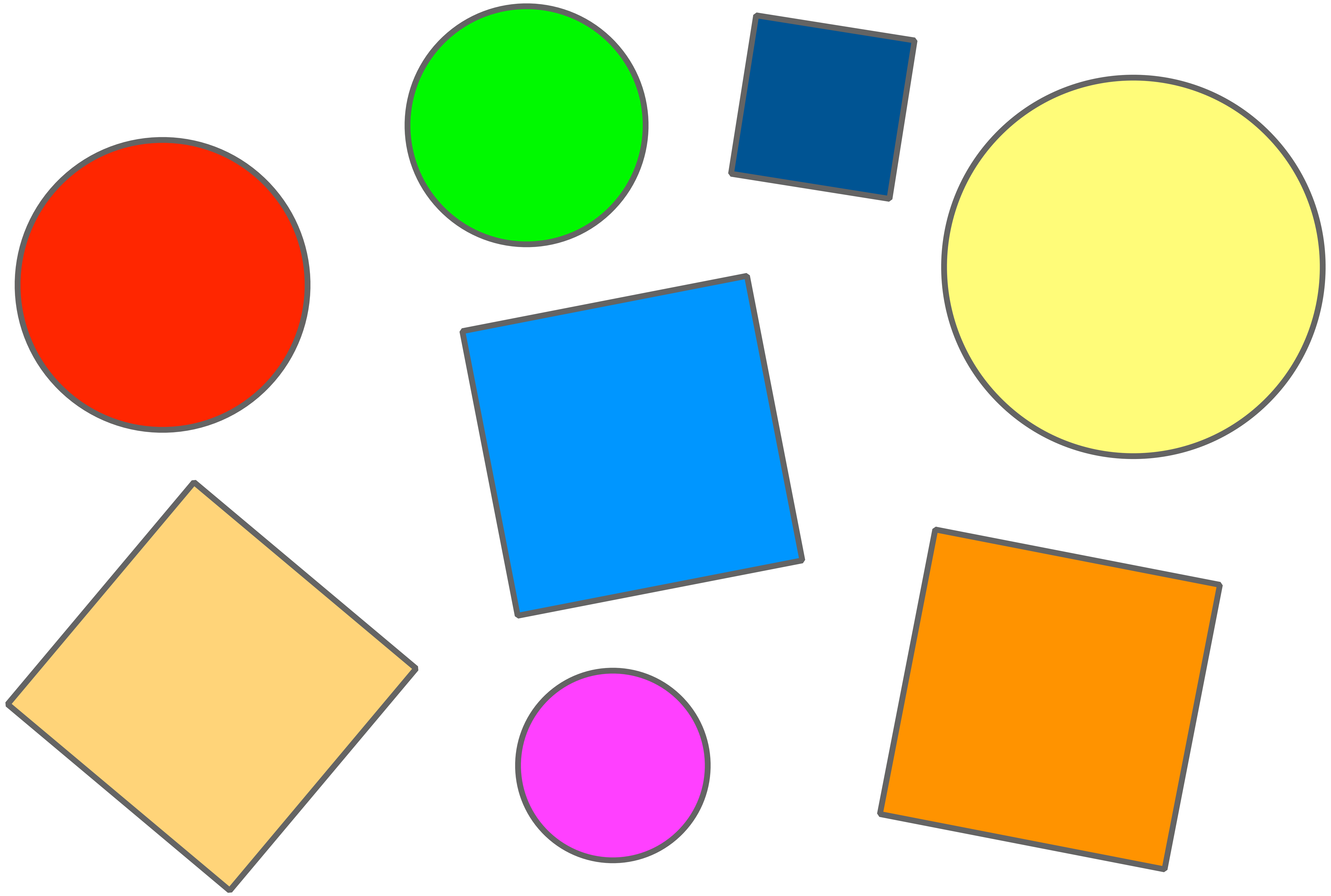
The Source of Classic OO Design Patterns



One of these patterns is the Visitor design pattern



Our Toy Problem: Drawing Shapes



A Classic Visitor Implementation

```cpp
class Circle;
class Square;

class ShapeVisitor
{
 public:
   virtual ~ShapeVisitor() = default;

   virtual void visit( Circle const& ) const = 0;
   virtual void visit( Square const& ) const = 0;
};


class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void accept( ShapeVisitor const& ) = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}
```

A Classic Visitor Implementation

```
class Circle;
class Square;

class ShapeVisitor
{
 public:
   virtual ~ShapeVisitor() = default;

   virtual void visit( Circle const& ) const = 0;
   virtual void visit( Square const& ) const = 0;
};


class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void accept( ShapeVisitor const& ) = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}
```

A Classic Visitor Implementation

```
class Circle;
class Square;

class ShapeVisitor
{
 public:
   virtual ~ShapeVisitor() = default;

   virtual void visit( Circle const& ) const = 0;
   virtual void visit( Square const& ) const = 0;
};


class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void accept( ShapeVisitor const& ) = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}
```

The two inheritance hierarchies

A Classic Visitor Implementation

```
class Circle;
class Square;

class ShapeVisitor
{
 public:
   virtual ~ShapeVisitor() = default;

   virtual void visit( Circle const& ) const = 0;
   virtual void visit( Square const& ) const = 0;
};


class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void accept( ShapeVisitor const& ) = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}
```

The consequence: many virtual functions

A Classic Visitor Implementation

```
   virtual ~Shape() = default;

   virtual void accept( ShapeVisitor const& ) = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void accept( ShapeVisitor const& ) override;

   // ...

 private:
   double radius;
   // ... Remaining data members
};


class Square : public Shape
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}
```

A Classic Visitor Implementation

```
 private:
   double radius;
   // ... Remaining data members
};


class Square : public Shape
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

   void accept( ShapeVisitor const& ) override;

   // ...

 private:
   double side;
   // ... Remaining data members
};


class Draw : public ShapeVisitor
{
 public:
   void visit( Circle const& ) const override;
   void visit( Square const& ) const override;
};
```

All derived classes need to implement the accept() function

A Classic Visitor Implementation

```
 private:
   double side;
   // ... Remaining data members
};


class Draw : public ShapeVisitor
{
 public:
   void visit( Circle const& ) const override;
   void visit( Square const& ) const override;
};


void drawAllShapes( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      s->accept( Draw{} )
   }
}


int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( std::make_unique<Circle>( 2.0 ) );
   shapes.emplace_back( std::make_unique<Square>( 1.5 ) );
```

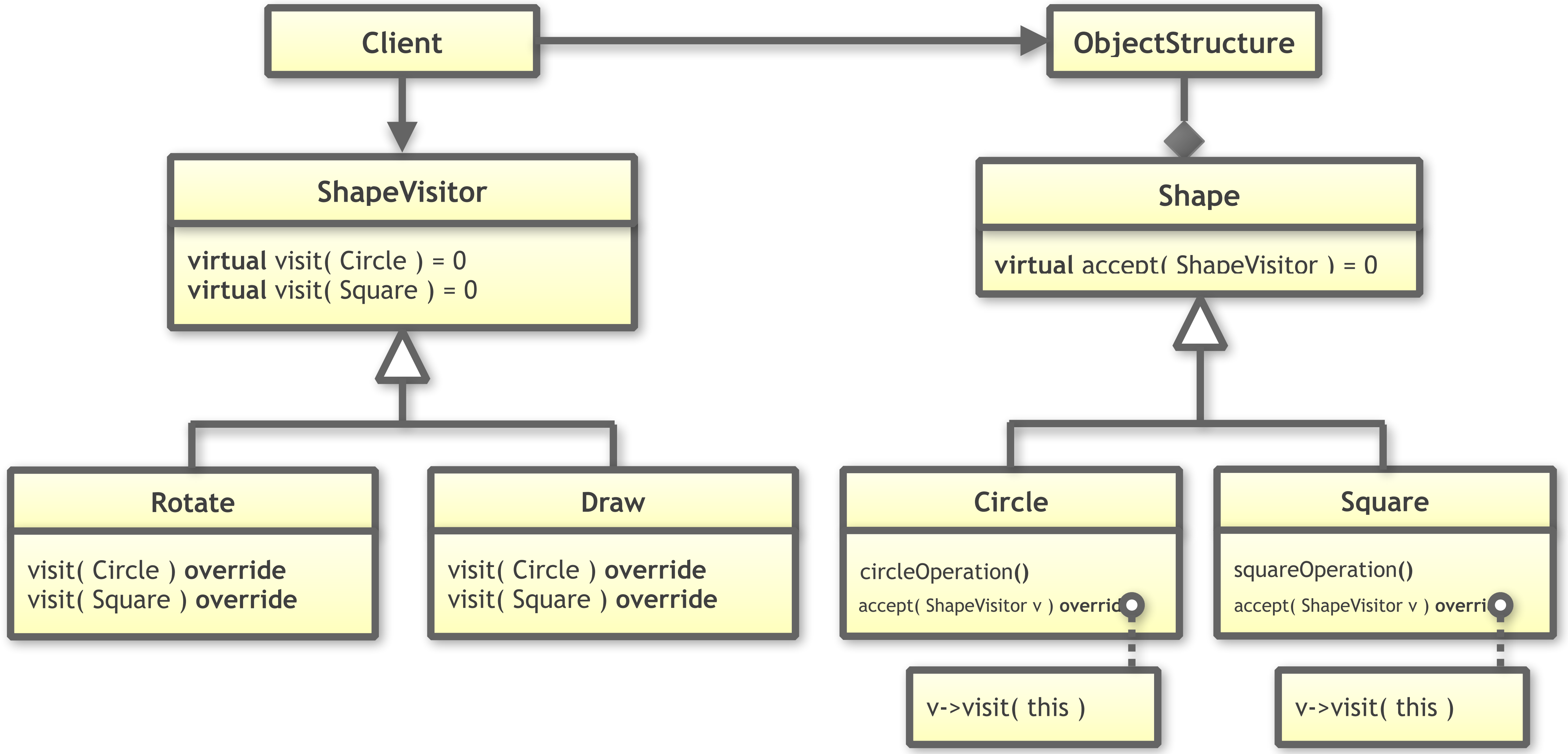
A Classic Visitor Implementation

```
void drawAllShapes( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      s->accept( Draw{} )
   }
}


int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( std::make_unique<Circle>( 2.0 ) );
   shapes.emplace_back( std::make_unique<Square>( 1.5 ) );
   shapes.emplace_back( std::make_unique<Circle>( 4.2 ) );

   // Drawing all shapes
   drawAllShapes( shapes );
}
```

Another consequence: pointers

And yet another consequence: manual allocations

Evaluation of the Classic Visitor Style



This style of programming has many disadvantages:

- We have a **two inheritance hierarchies** (intrusive)
- Performance is reduced due to **two virtual function calls** per operation
- Performance is affected due to **many pointers** (indirections)
- Promotes **dynamic memory allocation**
- Performance is reduced due to **many small, manual allocations**
- We need to **manage lifetimes explicitly** (`std::unique_ptr`)
- Danger of **lifetime-related bugs**

But there is a better solution …

… a value semantics solution …

```cpp
using Shape = std::variant<Circle,Square>;
```

A Value Semantics Solution

```cpp
class Circle
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double radius;
   // ... Remaining data members
};


class Square
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
```

A Value Semantics Solution

```
class Circle
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double radius;
   // ... Remaining data members
};
```

No base class required (non-intrusive)!

Circle is not used via pointers, but as a value

No accumulation of dependencies
via member functions!

```
class Square
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double side;
```

A Value Semantics Solution

```cpp
 private:
   double radius;
   // ... Remaining data members
};


class Square
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
};


class Draw
{
 public:
   void operator()( Circle const& ) const;
   void operator()( Square const& ) const;
};


using Shape = std::variant<Circle,Square>;
```

A Value Semantics Solution

```
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
};


class Draw
{
 public:
   void operator()( Circle const& ) const;
   void operator()( Square const& ) const;
};


using Shape = std::variant<Circle,Square>;

void drawAllShapes( std::vector<Shape> const& shapes )
{
   for( auto const& s : shapes )
   {
      std::visit( Draw{}, s );
   }
}


int main()
{
   using Shapes = std::vector<Shape>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( Circle{ 2.0 } );
```

No base class required!

Draw is not used via pointer, but as a value

Operations can be non-intrusively be added (OCP)

A Value Semantics Solution

```
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
};


class Draw
{
 public:
   void operator()( Circle const& ) const;
   void operator()( Square const& ) const;
};


using Shape = std::variant<Circle,Square>;

void drawAllShapes( std::vector<Shape> const& shapes )
{
   for( auto const& s : shapes )
   {
      std::visit( Draw{}, s );
   }
}


int main()
{
   using Shapes = std::vector<Shape>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( Circle{ 2.0 } );
```

A shape is a value, representing either a circle or a square

A Value Semantics Solution

```
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
};


class Draw
{
 public:
   void operator()( Circle const& ) const;
   void operator()( Square const& ) const;
};


using Shape = std::variant<Circle,Square>;

void drawAllShapes( std::vector<Shape> const& shapes )
{
   for( auto const& s : shapes )
   {
      std::visit( Draw{}, s );
   }
}


int main()
{
   using Shapes = std::vector<Shape>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( Circle{ 2.0 } );
```

The function expects
a vector of values

A Value Semantics Solution

```cpp
void drawAllShapes( std::vector<Shape> const& shapes )
{
   for( auto const& s : shapes )
   {
      std::visit( Draw{}, s );
   }
}


int main()
{
   using Shapes = std::vector<Shape>;

   // Creating some shapes
   Shapes shapes;
   shapes.emplace_back( Circle{ 2.0 } );
   shapes.emplace_back( Square{ 1.5 } );
   shapes.emplace_back( Circle{ 4.2 } );

   // Drawing all shapes
   drawAllShapes( shapes );
}
```

No pointers, no allocations, but values …

… and only values, making the code soooo much simpler.

Evaluation of the Modern Visitor Style

This style of programming has many advantages:

- There is **no inheritance** hierarchy
- The code is **so much simpler** (KISS)
- There are **no virtual functions**
- There are **no pointers** or indirections
- There is **no manual dynamic memory** allocation
- There is **no need to manage lifetime**
- There is **no lifetime-related issue** (no need for smart pointers)
- The **performance** is better

Performance Comparison

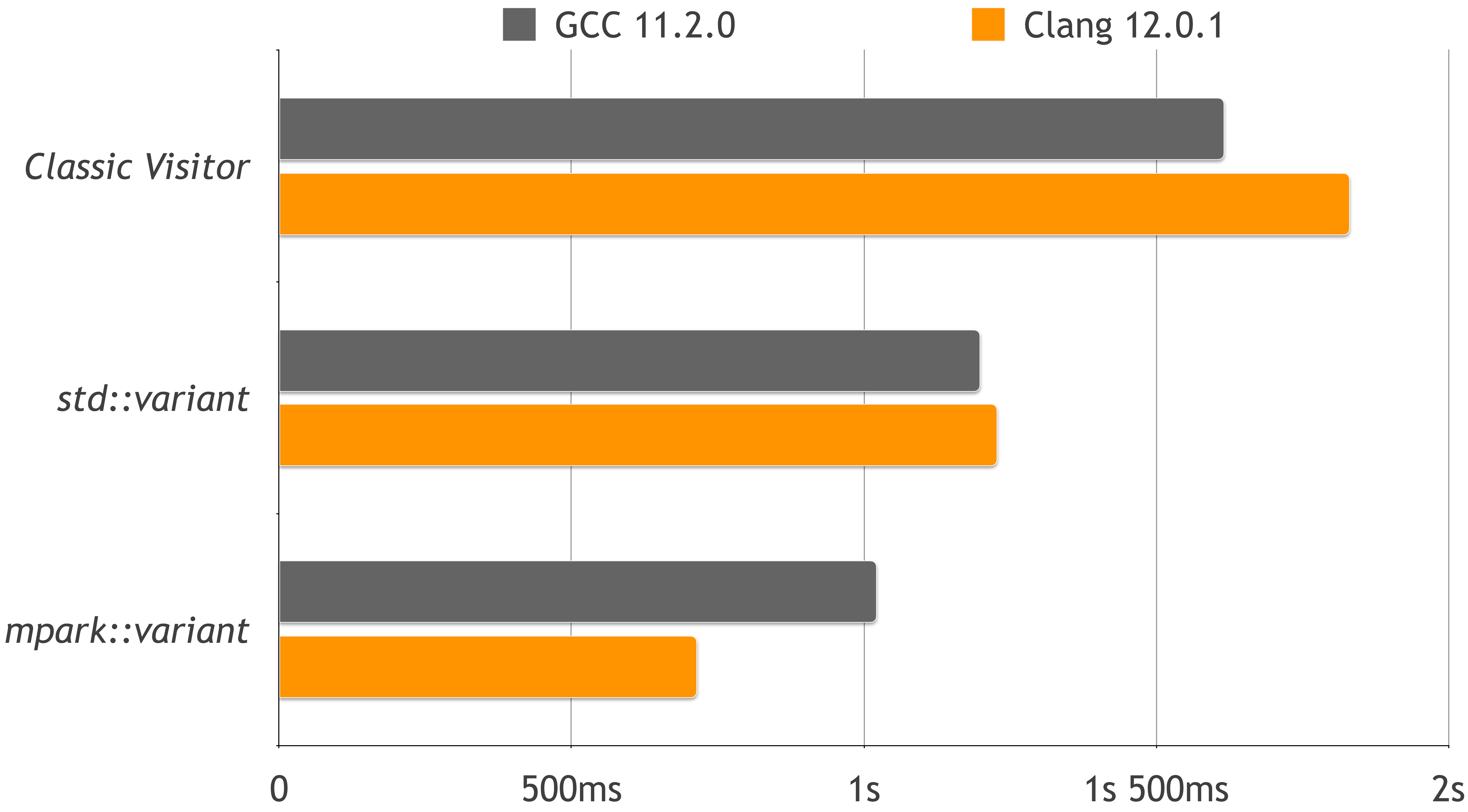
Performance … *sigh*

Do you promise to not take the following results too seriously and as qualitative results only?

Performance Comparison

- Using four different kinds of shape: circles, squares, ellipses and rectangles
- Using 10000 randomly generated shapes
- Performing 25000 `translate()` operations each
- Benchmarks with GCC-11.2.0 and Clang-12.0.1
- 8-core Intel Core i7 with 3.8 Ghz, 64 GB of main memory

Performance Comparison



You have just experienced the advantages of

Value Semantics

Value Semantics …

- … will make your code (much) easier to understand (less code).
- … will make your code (much) easier to write.
- … will make your code more correct (as you avoid many common bugs).
- … will (potentially) make your code faster.

… is preferable in comparison to reference semantics.

Here is another (smaller) example …

Reference Semantics: std::span

```cpp
#include <vector>
#include <span>

void print( std::vector<int> const& vec );


int main()
{
   std::vector<int> v{ 1, 2, 3, 4 };

   std::vector<int> const w{ v };
   std::span<int> const s{ v };   -> equivalent to a int* const

   w[2] = 99;  // Compilation error!  -> Value semantics
   s[2] = 99;  // Works!              -> Reference semantics

   // Prints  1 2 99 4
   print( v );

   return EXIT_SUCCESS;
}
```

Reference Semantics: std::span

```
#include <vector>
#include <span>

void print( std::vector<int> const& vec );


int main()
{
   std::vector<int> v{ 1, 2, 3, 4 };

   std::vector<int> const w{ v };
   std::span<int> const s{ v };   -> std::span<int const> const

   w[2] = 99;  // Compilation error!  -> Value semantics
   s[2] = 99;  // Works!              -> Reference semantics

   // Prints  1 2 99 4
   print( v );

   return EXIT_SUCCESS;
}
```

Reference Semantics: std::span

```
#include <vector>
#include <span>

void print( std::vector<int> const& vec );
void print( std::span<int> s );

int main()
{
   std::vector<int> v{ 1, 2, 3, 4 };


   std::span<int> const s{ v };

   v = { 5, 6, 7, 8, 9 }; -> Causes an internal reallocation 😕
   s[2] = 99;  // Works!  -> Triggers undefined behaviour (UB)!

   // Maybe prints  1 2 99 4
   print( s );

   return EXIT_SUCCESS;
}
```

It is reasonable to have a std::span as function argument

It is dangerous to keep a std::span around for longer (also as a data member)

And another example …

Reference Semantics: Reference Parameter

```
#include <algorithm>
#include <vector>

void print( std::span<int const> s );

int main()
{
   std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

   print( vec );     // Prints ( 1 -3 27 42 4 -8 22 42 37 4 18 9 )

   // Determining the maximum element in the range 'vec'
   auto const pos = std::max_element( begin(vec), end(vec) );

   // Removing all maximum elements
   vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );

   print( vec );     // Prints ( 1 -3 27 4 -8 22 42 37 18 9 )

   return EXIT_SUCCESS;
}
```

Reference Semantics: Reference Parameter

```
#include <algorithm>
#include <vector>

void print( std::span<int const> s );

int main()
{
```

```
template< class ForwardIt, class T >
constexpr ForwardIt remove( ForwardIt first, ForwardIt last, T const& value );
```

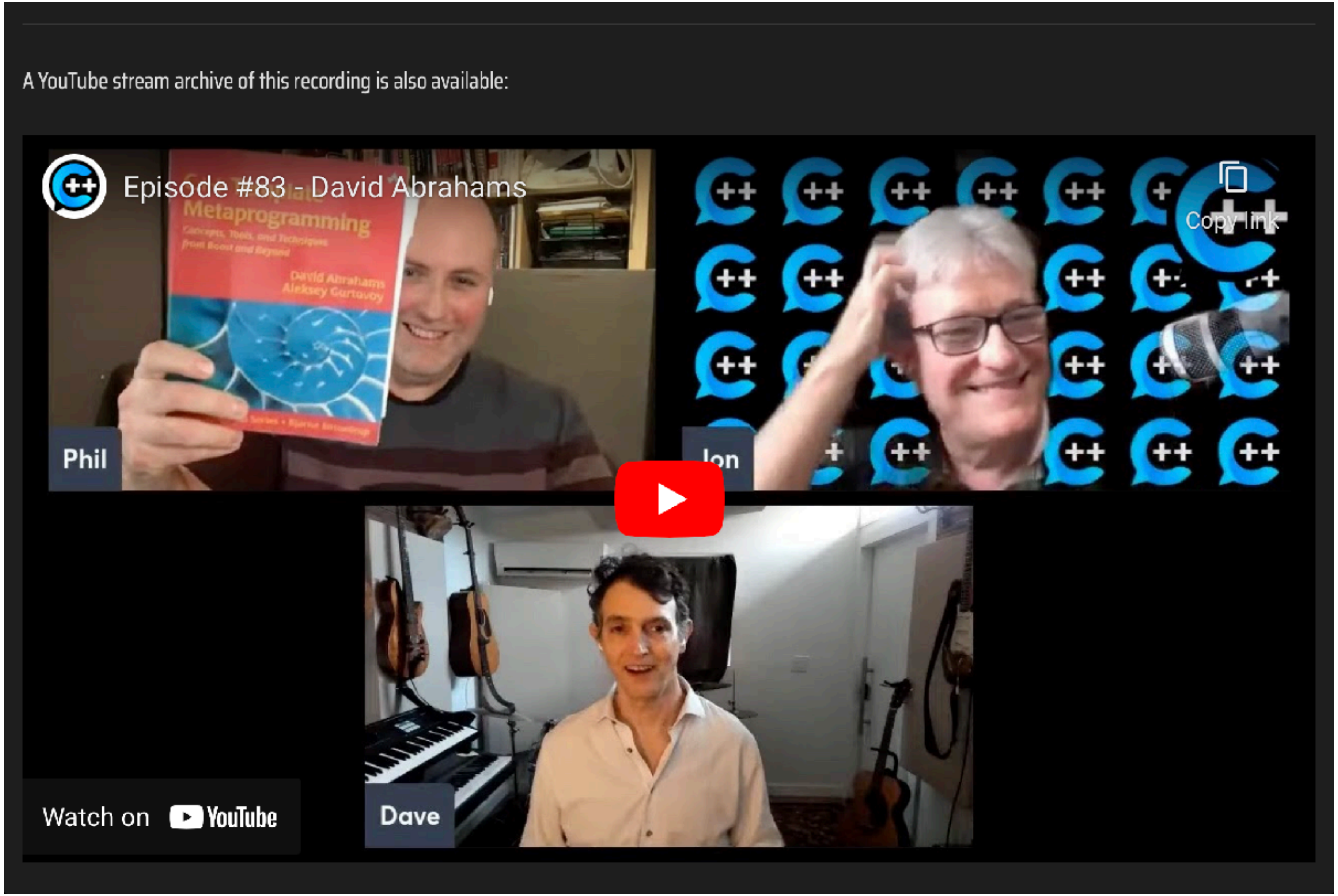
```
   // Determining the maximum element in the range 'vec'
   auto const pos = std::max_element( begin(vec), end(vec) );

   // Removing all maximum elements
   vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );

   print( vec );     // Prints ( 1 -3 27 4 -8 22 42 37 18 9 )

   return EXIT_SUCCESS;
}
```

Reference Semantics: Reference Parameter

```cpp
#include <algorithm>
#include <vector>

void print( std::span<int const> s );

int main()
{
   std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

   print( vec );     // Prints ( 1 -3 27 42 4 -8 22 42 37 4 18 9 )

   // Determining the maximum element in the range 'vec'
   auto const pos = std::max_element( begin(vec), end(vec) );

   // Removing all maximum elements
   vec.erase( std::remove( begin(vec), end(vec), *pos ), end(vec) );

   print( vec );     // Prints ( 1 -3 27 4 -8 22 42 37 18 9 )

   return EXIT_SUCCESS;
}
```

Reference Semantics: Reference Parameter

```
#include <algorithm>
#include <vector>

void print( std::span<int const> s );

int main()
{
   std::vector<int> vec{ 1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9 };

   print( vec );     // Prints ( 1 -3 27 42 4 -8 22 42 37 4 18 9 )

   // Determining the maximum element in the range 'vec'
   auto const pos = std::max_element( begin(vec), end(vec) );

   // Removing all maximum elements
   std::erase( vec, end(vec), *pos );

   print( vec );     // Prints ( 1 -3 27 4 -8 22 42 37 18 9 )

   return EXIT_SUCCESS;
}
```



“C++ takes value semantics seriously!”

(Dave Abrahams)

Examples from the Standard Library

There are further examples for value semantics from the Standard Library:

- The design of the STL (C++98)
- `std::optional` (C++17)
- `std::expected` (C++23)
- `std::function` (C++11)
- `std::any` (C++17)

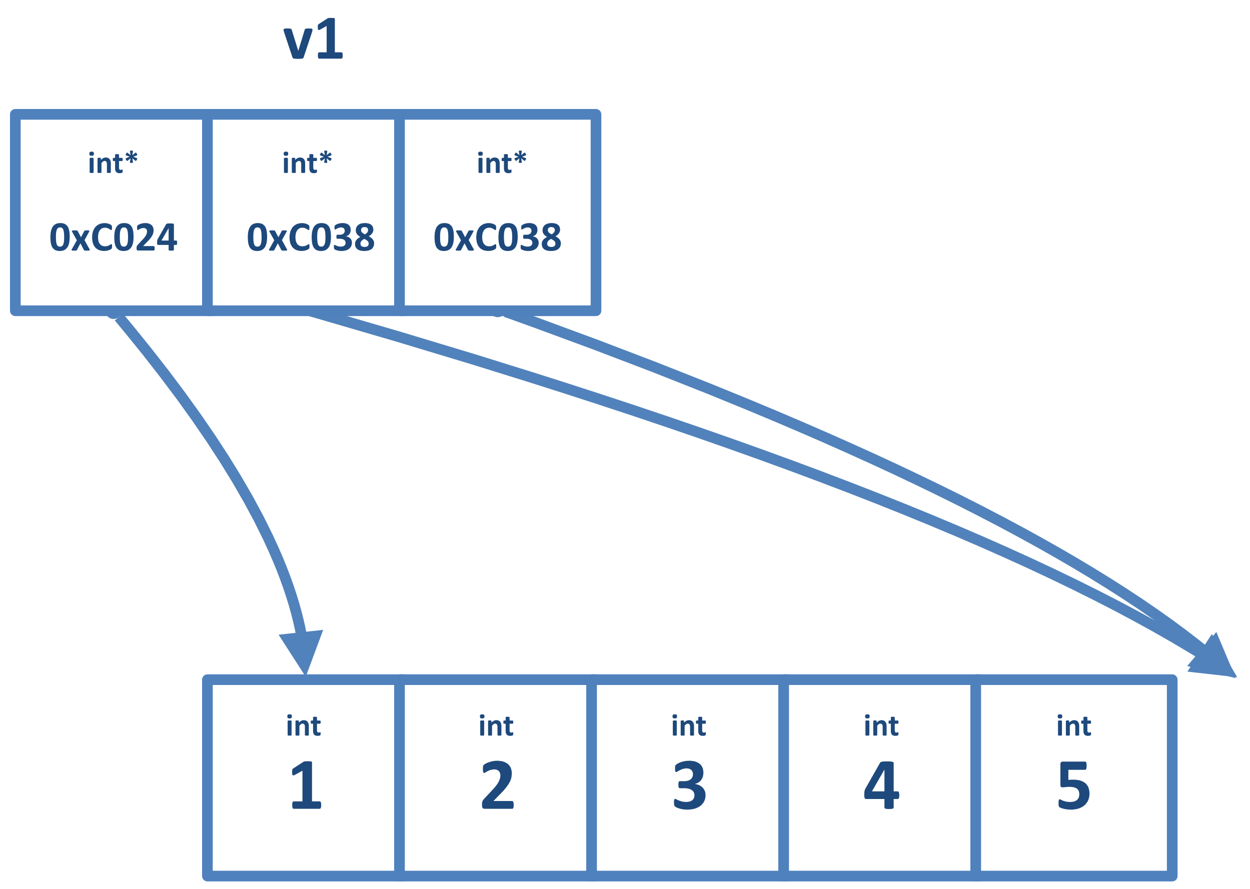
Example: The Design of the STL

In the design of the STL, …

- … containers are values
  - copy implies a deep copy

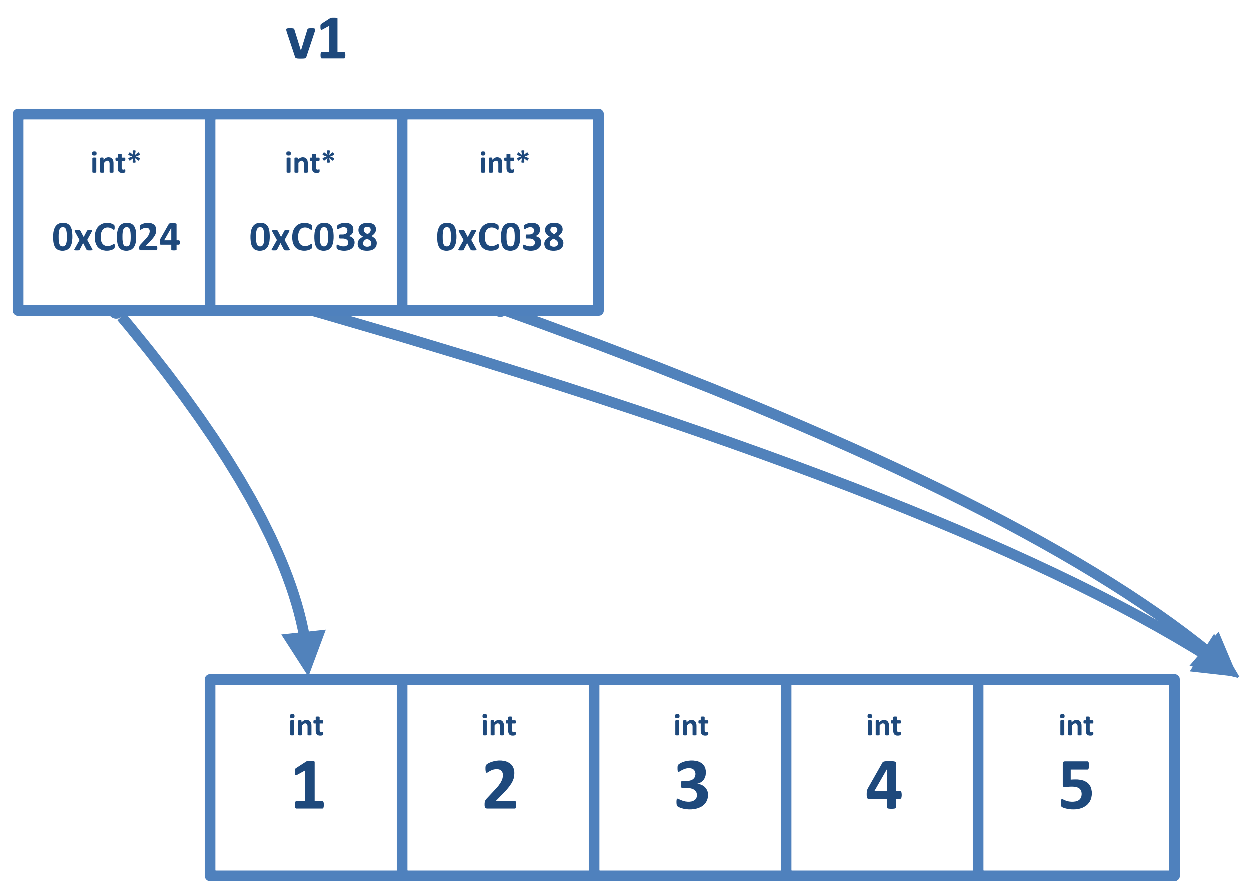
Example: The Design of the STL

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
```



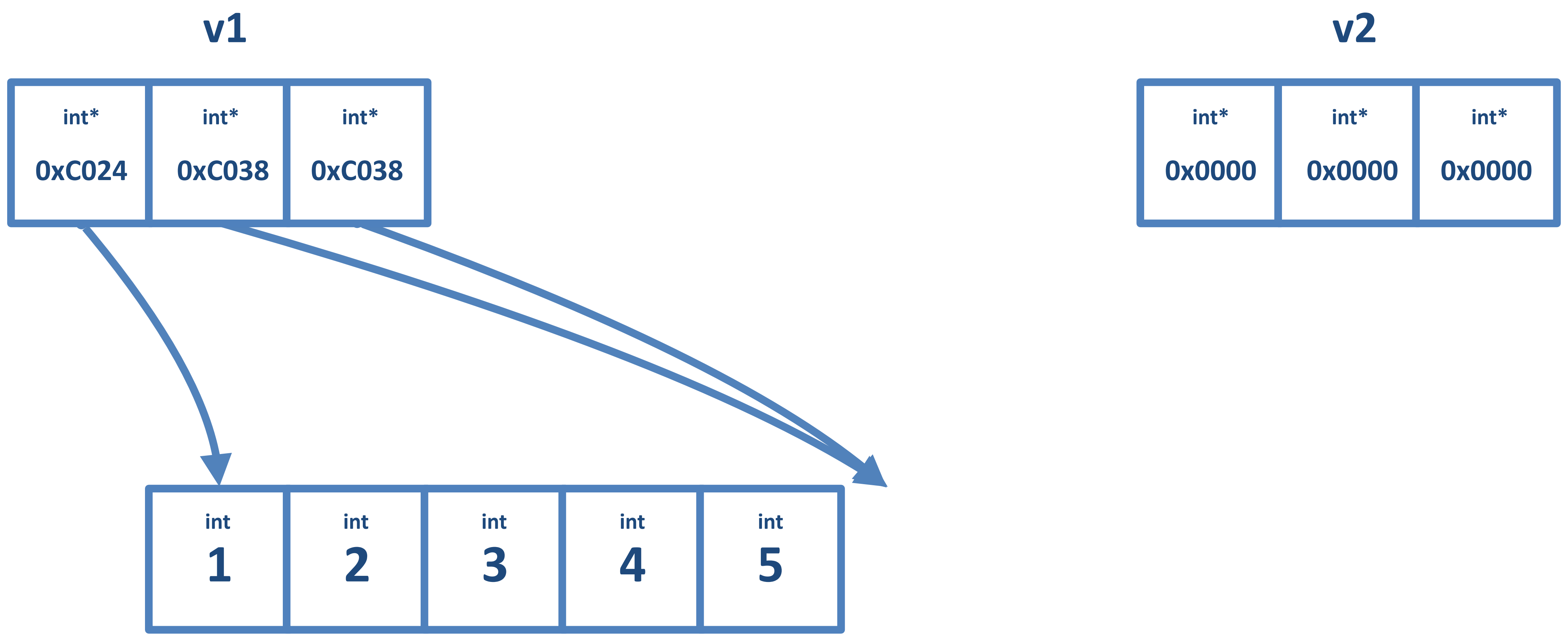
Example: The Design of the STL

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};
```



Example: The Design of the STL

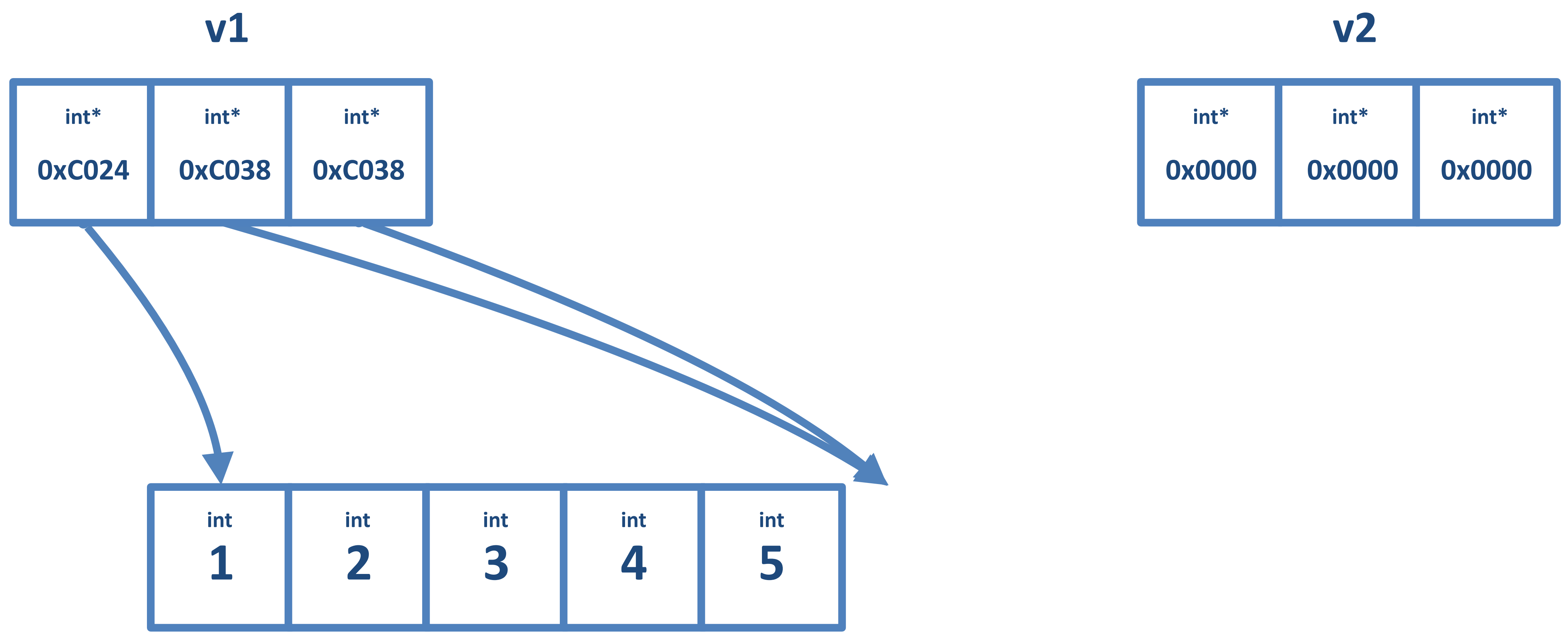
```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};
```



Example: The Design of the STL

C++98

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};

v2 = v1;
```

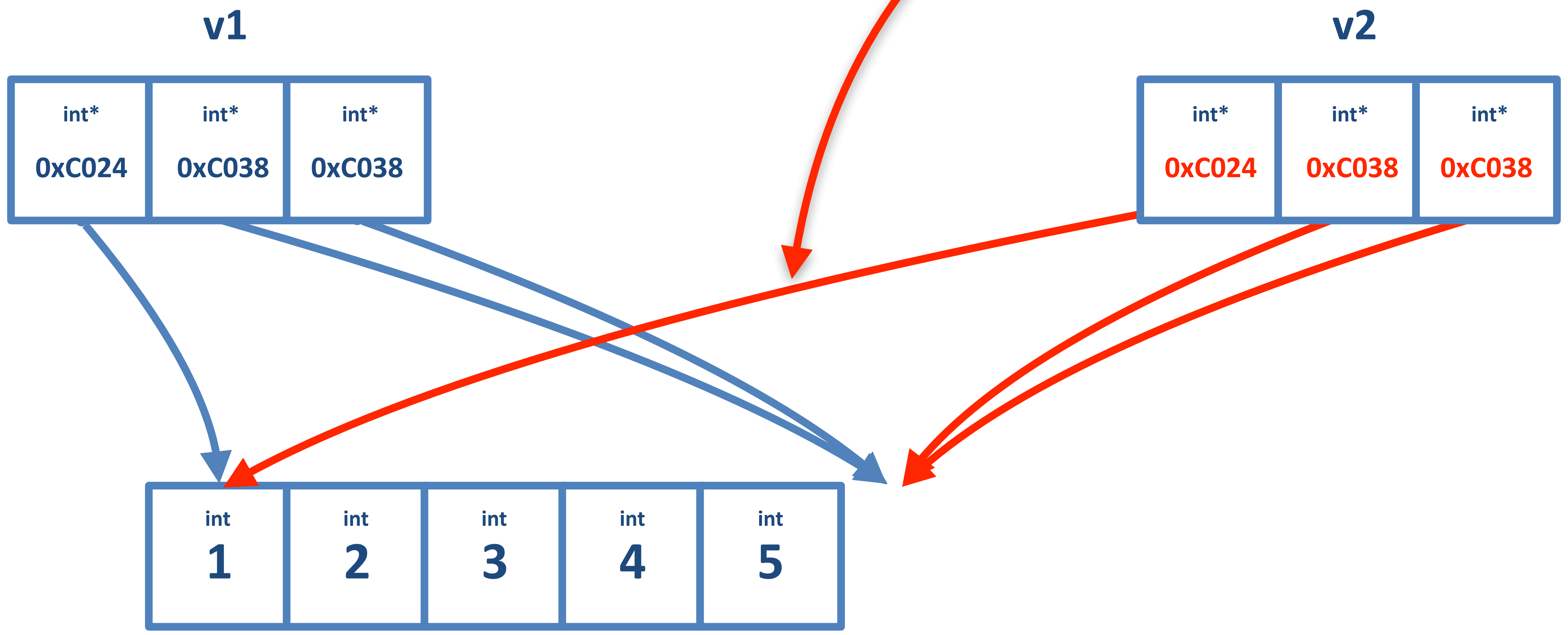


Example: The Design of the STL

C++98

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};

v2 = v1;
```

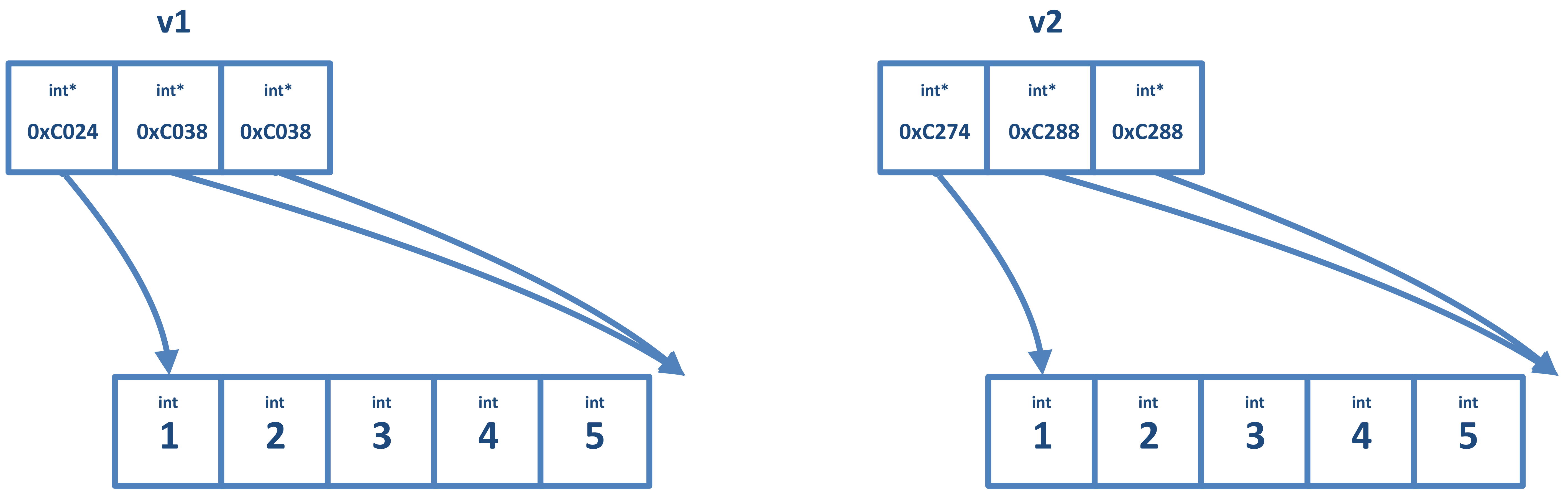
A shallow copy would be a bad idea: sharing makes it hard to reason about the code and the implementation sooooo much more difficult!



Example: The Design of the STL

```cpp
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};

v2 = v1;
```

The STL performs a deep copy instead: there is no sharing and both vectors have their own content



Example: The Design of the STL

In the design of the STL, …

- … containers are values
  - copy implies a deep copy
  - const means const

```
std::vector<int> v{ 1, 2, 3, 4 };

std::vector<int> const w{ v };
```

Example: The Design of the STL

In the design of the STL, …

- … containers are values
  - copy implies a deep copy
  - const means const
- … algorithms take arguments by value

```
template< typename InputIt, typename OutputIt, typename UnaryPredicate >
constexpr OutputIt copy_if( InputIt first, InputIt last,
                            OutputIt d_first,
                            UnaryPredicate pred );
```

Example: std::optional

```
// Return default int on parse error
int to_int( std::string_view s );

// Throw on parse error
int to_int( std::string_view s );

// Return false on parse error
bool to_int( std::string_view s, int& );

// Return null on parse error
std::unique_ptr<int> to_int( std::string_view s );
```

Example: std::optional

```
std::optional<int> to_int( std::string_view s );
```

- There is no question of **ownership**
- There is no question of **semantics**
- There is **no exception overhead**
- It is **efficient** (return value optimisation (RVO) and move semantics)
- It is simple!

The main disadvantage: You cannot communicate the reason for a failure

Example: std::expected

```
std::expected<int,std::string> to_int( std::string_view s );
```

- There is no question of **ownership**
- There is no question of **semantics**
- There is **no exception overhead**
- It is **efficient** (return value optimisation (RVO) and move semantics)
- It is simple!

Example: std::function

Would you provide an abstraction for callable by means of an inheritance hierarchy?

```
class Command
{
 public:
   virtual void operator()( int ) const = 0;
   // ...
};

class PrintCommand   : public Command { /*...*/ };
class SearchCommand  : public Command { /*...*/ };
class ExecuteCommand : public Command { /*...*/ };


void f( Command* command );
```

Example: std::function

No, you wouldn’t. You would use std::function instead!

```
class PrintCommand { /*...*/ };
class SearchCommand { /*...*/ };
class ExecuteCommand { /*...*/ };


void f( std::function<void(int)> command );
```

std::function instead of inheritance:

- no inheritance hierarchies
- non-intrusive
- no pointers
- no manual dynamic allocation
- no manual life-time management
- less code to write

Summary

The take-away from this talk:

Prefer value semantics over reference semantics

Value semantics …

- … will make your code (much) easier to understand (less code).
- … will make your code (much) easier to write.
- … will make your code more correct (as you avoid many common bugs).
- … will (potentially) make your code faster.

Back To Basics
Value Semantics

KLAUS IGLBERGER

Cppcon
The C++ Conference

2022
September 12th-16th