Breaking Dependencies:

Type Erasure - The Implementation Details

KLAUS IGLBERGER

C++ Trainer/Consultant

Author of the blaze C++ math library

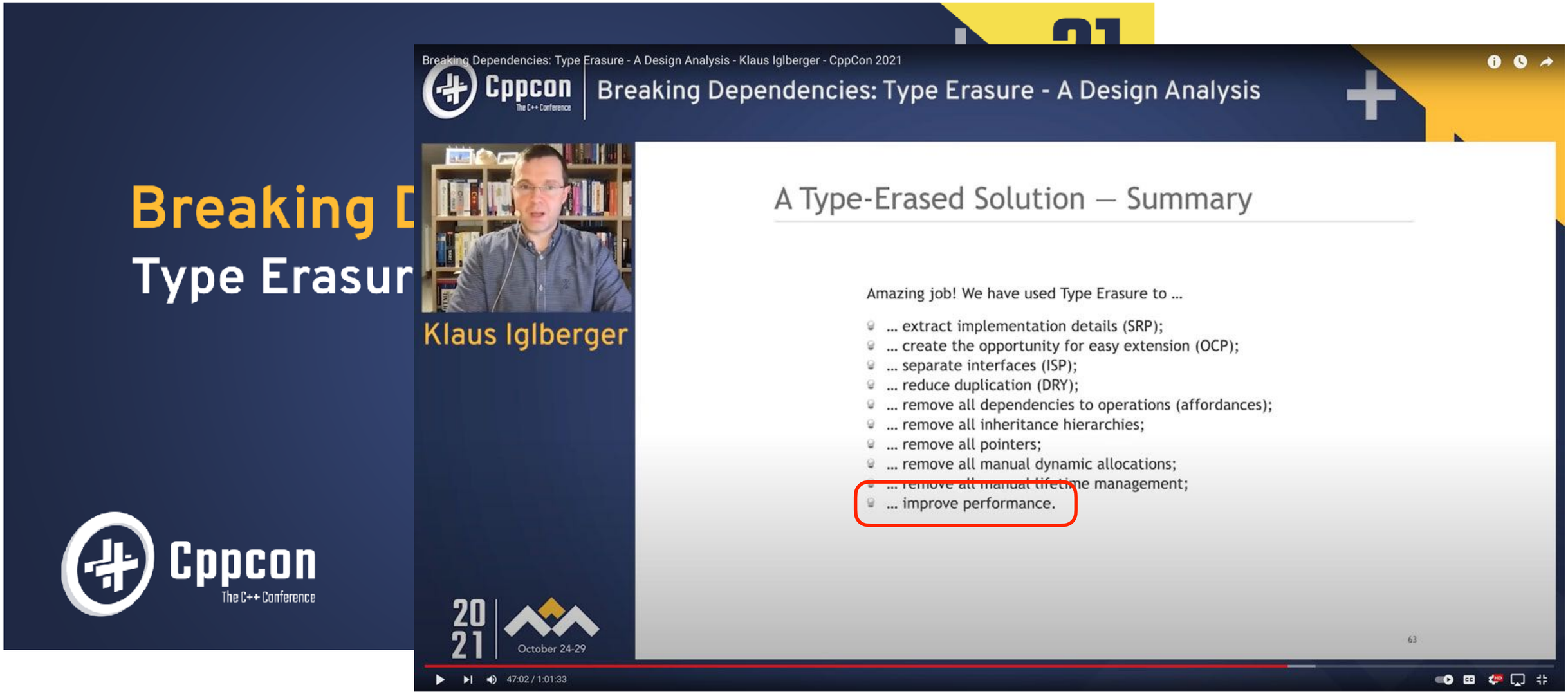
(Co-)Organizer of the Munich C++ user group

Chair of the CppCon B2B track

Email: klaus.iglberger@gmx.de

Klaus Iglberger

Type Erasure - A Design Analysis



Content

- The Motivation for Type Erasure
- A Basic Type Erasure Implementation
- Type Erasure with Small Buffer Optimization (SBO)
- Type Erasure with Manual Virtual Dispatch

Content

- The Motivation for Type Erasure
- A Basic Type Erasure Implementation
- Type Erasure with Small Buffer Optimization (SBO)
- Type Erasure with Manual Virtual Dispatch

A Motivating Example

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 );
```

A Motivating Example

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

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

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

Type Erasure instead of inheritance:

- no inheritance hierarchies
- non-intrusive
- less dependencies
- less pointers
- no manual dynamic allocation
- no manual life-time management
- value semantics
- less code to write
- potentially better performance

A Motivating Example

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

```
class Shape
{
 public:
   virtual void draw( /*...*/ ) const = 0;
   // ...
};

class Circle   : public Shape { /*...*/ };
class Square   : public Shape { /*...*/ };
class Triangle : public Shape { /*...*/ };


void f( Shape* shape );
```

A Motivating Example

No, you wouldn’t. You would use Type Erasure instead!

```
class Circle { /*...*/ };
class Square { /*...*/ };
class Triangle { /*...*/ };


void f( Shape command );
```

Type Erasure instead of inheritance:

- no inheritance hierarchies
- non-intrusive
- less dependencies
- less pointers
- no manual dynamic allocation
- no manual life-time management
- value semantics
- less code to write
- potentially better performance

”Inheritance is Rarely the Answer.”
(Andrew Hunt, David Thomas, The Pragmatic Programmer)

”The Answer might be Type Erasure (at least much more often).”
(myself)

Content

- The Motivation for Type Erasure
- A Basic Type Erasure Implementation
- Type Erasure with Small Buffer Optimization (SBO)
- Type Erasure with Manual Virtual Dispatch

Type Erasure — Terminology

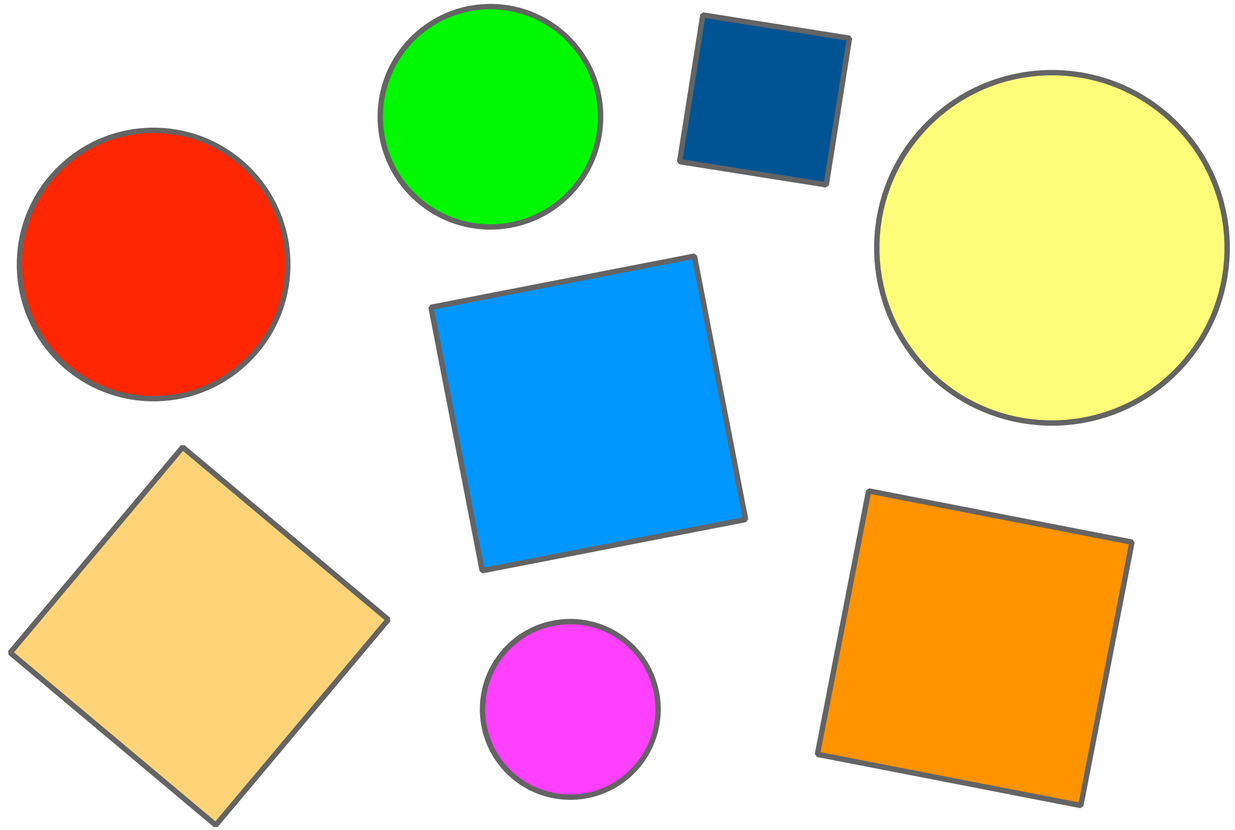
Type Erasure is **not** …

- … a **`void*`**;
- … a **pointer-to-base**;
- … a **`std::variant`**.

Type Erasure is …

- … a **templated constructor** plus …
- … a completely **non-virtual interface;**
- … **External Polymorphism + Bridge + Prototype.**

Our Toy Problem: Shapes



”I’m tired of this example, but I don’t know any better one.”

(Lukas Bergdoll, MUC++ organizer)

A Type-Erased 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
};
```

```
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 Type-Erased 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
};
```

```
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 Type-Erased Solution

```
 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
};
```

Circles and squares …

- **… don’t need a base class;**
- **… don’t know about each other;**
- **… don’t know anything about their operations (affordances).**

```
struct ShapeConcept
{
   virtual ~ShapeConcept() = default;
```

A Type-Erased Solution

```
struct ShapeConcept
{
   virtual ~ShapeConcept() = default;



   // ...
};

template< typename ShapeT >
struct ShapeModel : public ShapeConcept
{
   ShapeModel( ShapeT shape )
      : shape_{ std::move(shape) }
   {}

   // ...

   ShapeT shape_;
};
```

A Type-Erased Solution

```
struct ShapeConcept
{
   virtual ~ShapeConcept() = default;

   virtual void do_serialize( /*...*/ ) const = 0;
   virtual void do_draw( /*...*/ ) const = 0;
   // ...
};

template< typename ShapeT >
struct ShapeModel : public ShapeConcept
{
   ShapeModel( ShapeT shape )
      : shape_{ std::move(shape) }
   {}

   // ...

   ShapeT shape_;
};
```

A Type-Erased Solution

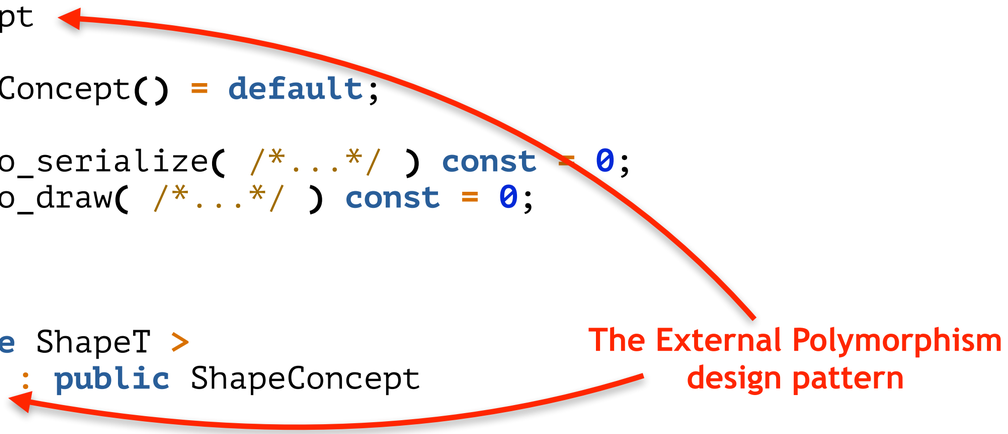
```
struct ShapeConcept
{
   virtual ~ShapeConcept() = default;

   virtual void do_serialize( /*...*/ ) const = 0;
   virtual void do_draw( /*...*/ ) const = 0;
   // ...
};

template< typename ShapeT >
struct ShapeModel : public ShapeConcept
{
   ShapeModel( ShapeT shape )
      : shape_{ std::move(shape) }
   {}

   // ...

   void do_serialize( /*...*/ ) const override
   {
      serialize( shape_, /*...*/ );
   }

   void do_draw( /*...*/ ) const override
   {
      draw( shape_, /*...*/ );
   }

   ShapeT shape_;
};
```

Note the do_ prefix, which avoids naming conflicts

The implementation of the virtual functions in the `ShapeModel` class defines the affordances required by the type `T`

A Type-Erased Solution

```
struct ShapeConcept
{
   virtual ~ShapeConcept() = default;

   virtual void do_serialize( /*...*/ ) const = 0;
   virtual void do_draw( /*...*/ ) const = 0;
   // ...
};

template< typename ShapeT >
struct ShapeModel : public ShapeConcept
{
   ShapeModel( ShapeT shape )
      : shape_{ std::move(shape) }
   {}

   // ...

   void do_serialize( /*...*/ ) const override
   {
      serialize( shape_, /*...*/ );
   }

   void do_draw( /*...*/ ) const override
   {
      draw( shape_, /*...*/ );
   }

   ShapeT shape_;
};
```



The External Polymorphism Design Pattern

External Polymorphism (3rd Pattern Languages of Programming Conference, September 4-6, 1996)

External Polymorphism

An Object Structural Pattern for Transparently Extending C++ Concrete Data Types

Chris Cleeland
chris@envision.com
Envision Solutions, St. Louis, MO 63141

Douglas C. Schmidt and Timothy H. Harrison
schmidt@cs.wustl.edu and harrison@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, Missouri, 63130

This paper appeared in the Proceedings of the 3rd Pattern Languages of Programming Conference, Allerton Park, Illinois, September 4–6, 1996.

1 Intent

Allow C++ classes unrelated by inheritance and/or having no virtual methods to be treated polymorphically. These unrelated classes can be treated in a common manner by software that uses them.

2 Motivation

Working with C++ classes from different sources can be difficult. Often an application may wish to "project" common

1. *Space efficiency* – The solution must not constrain the storage layout of existing objects. In particular, classes that have no virtual methods (*i.e.*, concrete data types) must not be forced to add a virtual table pointer.

2. *Polymorphism* – All library objects must be accessed in a uniform, transparent manner. In particular, if new classes are included into the system, we won't want to change existing code.

Consider the following example using classes from the ACE network programming framework [3]:

```
1. SOCK_Acceptor acceptor; // Global storage
2.
3. int main (void) {
4.     SOCK_Stream stream; // Automatic storage
```

The External Polymorphism Design Pattern

The External Polymorphism Design Pattern …

- … allows any shape_ to be treated polymorphically;
- … extracts implementation details (SRP);
- … removes dependencies to operations (affordances);
- … creates the opportunity for easy extension (OCP).

A Type-Erased Solution

```
struct ShapeConcept
{
   virtual ~ShapeConcept() = default;

   virtual void do_serialize( /*...*/ ) const = 0;
   virtual void do_draw( /*...*/ ) const = 0;
   // ...
};

template< typename ShapeT >
struct ShapeModel : public ShapeConcept
{
   ShapeModel( ShapeT shape )
      : shape_{ std::move(shape) }
   {}

   // ...

   void do_serialize( /*...*/ ) const override
   {
      serialize( shape_, /*...*/ );
   }

   void do_draw( /*...*/ ) const override
   {
      draw( shape_, /*...*/ );
   }

   ShapeT shape_;
};
```

A Type-Erased Solution

```
        serialize( shape_, /*...*/ );
    }

    void do_draw( /*...*/ ) const override
    {
        draw( shape_, /*...*/ );
    }

    ShapeT shape_;
};


void serialize( Circle const&, /*...*/ );
void draw( Circle const&, /*...*/ );

void serialize( Square const&, /*...*/ );
void draw( Square const&, /*...*/ );


void drawAllShapes( std::vector<std::unique_ptr<ShapeConcept>> const& shapes )
{
    for( auto const& shape : shapes )
    {
        shape->draw();
    }
}


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

    // Creating some shapes
```

These functions resolve the requirements posed by the External Polymorphism design pattern.

There can be many implementation, spread over many header/source files (e.g. for OpenGL, Metal, Vulcan, …).

A Type-Erased Solution

```
   ShapeT shape_;
};


void serialize( Circle const&, /*...*/ );
void draw( Circle const&, /*...*/ );

void serialize( Square const&, /*...*/ );
void draw( Square const&, /*...*/ );


void drawAllShapes( std::vector<std::unique_ptr<ShapeConcept>> const& shapes )
{
   for( auto const& shape : shapes )
   {
      shape->draw();
   }
}


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

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

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

A Type-Erased Solution

```
void serialize( Square const&, /*...*/ );
void draw( Square const&, /*...*/ );


void drawAllShapes( std::vector<std::unique_ptr<ShapeConcept>> const& shapes )
{
   for( auto const& shape : shapes )
   {
      shape->draw();
   }
}


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

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

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

A Type-Erased Solution

```
struct ShapeConcept
{
   virtual ~ShapeConcept() = default;

   virtual void do_serialize( /*...*/ ) const = 0;
   virtual void do_draw( /*...*/ ) const = 0;
   // ...
};

template< typename ShapeT >
struct ShapeModel : public ShapeConcept
{
   ShapeModel( ShapeT shape )
      : shape_{ std::move(shape) }
   {}

   // ...

   void do_serialize( /*...*/ ) const override
   {
      serialize( shape_, /*...*/ );
   }

   void do_draw( /*...*/ ) const override
   {
      draw( shape_, /*...*/ );
   }

   ShapeT shape_;
};
```

A Type-Erased Solution

```
class Shape
{
 private:
   struct ShapeConcept
   {
      virtual ~ShapeConcept() = default;

      virtual void do_serialize( /*...*/ ) const = 0;
      virtual void do_draw( /*...*/ ) const = 0;
      // ...
   };

   template< typename ShapeT >
   struct ShapeModel : public ShapeConcept
   {
      ShapeModel( ShapeT shape )
         : shape_{ std::move(shape) }
      {}

      // ...

      void do_serialize( /*...*/ ) const override
      {
         serialize( shape_, /*...*/ );
      }

      void do_draw( /*...*/ ) const override
      {
         draw( shape_, /*...*/ );
      }

      ShapeT shape_;
   };
```

A Type-Erased Solution

```
      }

      ShapeT shape_;
   };
```

The Bridge design pattern

A templated constructor, creating a bridge

```
   std::unique_ptr<ShapeConcept> pimpl;

public:
   template< typename ShapeT >
   Shape( ShapeT shape )
      : pimpl{ std::make_unique<ShapeModel<ShapeT>>( std::move(shape) ) }
   {}

   // Copy operations
   Shape( Shape const& other );
   Shape& operator=( Shape const& other );

   // Move operations
   Shape( Shape&& other );
   Shape& operator=( Shape&& other );

   // ...
};
```

A Type-Erased Solution

```
      }

      ShapeT shape_;
   };
   friend void serialize( Shape const& shape, /*...*/ )
   {
      shape.pimpl->do_serialize( /*...*/ );
   }

   friend void draw( Shape const& shape, /*...*/ )
   {
      shape.pimpl->do_draw( /*...*/ );
   }

   std::unique_ptr<ShapeConcept> pimpl;

public:
   template< typename ShapeT >
   Shape( ShapeT shape )
      : pimpl{ std::make_unique<ShapeModel<ShapeT>>( std::move(shape) ) }
   {}

   // Copy operations
   Shape( Shape const& other );
   Shape& operator=( Shape const& other );

   // Move operations
   Shape( Shape&& other );
   Shape& operator=( Shape&& other );

   // ...
};
```

Despite being defined inside the class definition, these `friend` functions are free functions and injected into the surrounding namespace.

A Type-Erased Solution

```
      }

      ShapeT shape_;
   };
   friend void serialize( Shape const& shape, /*...*/ )
   {
      shape.pimpl->do_serialize( /*...*/ );
   }

   friend void draw( Shape const& shape, /*...*/ )
   {
      shape.pimpl->do_draw( /*...*/ );
   }

   std::unique_ptr<ShapeConcept> pimpl;

public:
   template< typename ShapeT >
   Shape( ShapeT shape )
      : pimpl{ std::make_unique<ShapeModel<ShapeT>>( std::move(shape) ) }
   {}

   // Copy operations
   Shape( Shape const& other );
   Shape& operator=( Shape const& other );

   // Move operations
   Shape( Shape&& other );
   Shape& operator=( Shape&& other );

   // ...
};
```

… but does only hold a pointer to base

A shape should be copyable …

A Type-Erased Solution

```
class Shape
{
 private:
   struct ShapeConcept
   {
      virtual ~ShapeConcept() = default;

      virtual void do_serialize( /*...*/ ) const = 0;
      virtual void do_draw( /*...*/ ) const = 0;
      // ...
   };

   template< typename ShapeT >
   struct ShapeModel : public ShapeConcept
   {
      ShapeModel( ShapeT shape )
         : shape_{ std::move(shape) }
      {}

      // ...

      void do_serialize( /*...*/ ) const override
      {
         serialize( shape_, /*...*/ );
      }

      void do_draw( /*...*/ ) const override
      {
         draw( shape_, /*...*/ );
      }

      ShapeT shape_;
```

A Type-Erased Solution

```
class Shape
{
 private:
   struct ShapeConcept
   {
      virtual ~ShapeConcept() = default;

      virtual void do_serialize( /*...*/ ) const = 0;
      virtual void do_draw( /*...*/ ) const = 0;
      virtual std::unique_ptr<ShapeConcept> clone() const = 0;
   };

   template< typename ShapeT >
   struct ShapeModel : public ShapeConcept
   {
      ShapeModel( ShapeT shape )
         : shape_{ std::move(shape) }
      {}

      std::unique_ptr<ShapeConcept> clone() const override
      {
         return std::make_unique<ShapeModel>(*this);
      }

      void do_serialize( /*...*/ ) const override
      {
         serialize( shape_, /*...*/ );
      }

      void do_draw( /*...*/ ) const override
      {
         draw( shape_, /*...*/ );
```

The Prototype design pattern

Note the use of the copy constructor of the ShapeModel. This will always do the right thing, even if other code changes.

A Type-Erased Solution

```
    std::unique_ptr<ShapeConcept> pimpl;

  public:
    template< typename ShapeT >
    Shape( ShapeT shape )
       : pimpl{ std::make_unique<ShapeModel<ShapeT>>( std::move(shape) ) }
    {}

    // Copy operations
    Shape( Shape const& other )
       : pimpl( other.pimpl->clone() )
    {}

    Shape& operator=( Shape const& other )
    {
       // Copy-and-swap idiom
       Shape tmp( other );
       std::swap( pimpl, tmp.pimpl );
       return *this;
    }

    // Move operations
    Shape( Shape&& other );
    Shape& operator=( Shape&& other );




    // ...
};
```

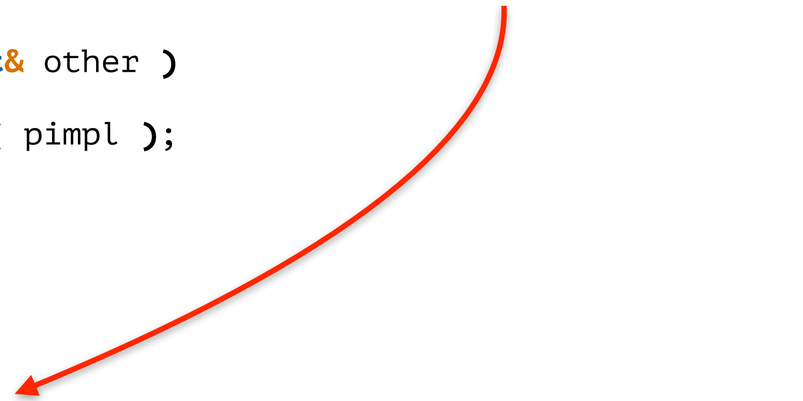
A Type-Erased Solution

```
   std::unique_ptr<ShapeConcept> pimpl;

public:
   template< typename ShapeT >
   Shape( ShapeT shape )
      : pimpl{ std::make_unique<ShapeModel<ShapeT>>( std::move(shape) ) }
   {}

   // Copy operations
   Shape( Shape const& other )
      : pimpl( other.pimpl->clone() )
   {}

   Shape& operator=( Shape const& other )
   {
      other.pimpl->clone().swap( pimpl );
      return *this;
   }



   // Move operations
   Shape( Shape&& other );
   Shape& operator=( Shape&& other );




   // ...
};
```

A Type-Erased Solution

```
   std::unique_ptr<ShapeConcept> pimpl;

public:
   template< typename ShapeT >
   Shape( ShapeT shape )
      : pimpl{ std::make_unique<ShapeModel<ShapeT>>( std::move(shape) ) }
   {}

   // Copy operations
   Shape( Shape const& other )
      : pimpl( other.pimpl->clone() )
   {}

   Shape& operator=( Shape const& other )
   {
      other.pimpl->clone().swap( pimpl );
      return *this;
   }



   // Move operations
   Shape( Shape&& other );
   Shape& operator=( Shape&& other );




   // ...
};
```

The move operations:



A Type-Erased Solution

```
   std::unique_ptr<ShapeConcept> pimpl;

public:
   template< typename ShapeT >
   Shape( ShapeT shape )
      : pimpl{ std::make_unique<ShapeModel<ShapeT>>( std::move(shape) ) }
   {}

   // Copy operations
   Shape( Shape const& other )
      : pimpl( other.pimpl->clone() )
   {}

   Shape& operator=( Shape const& other )
   {
      other.pimpl->clone().swap( pimpl );
      return *this;
   }



   // Move operations
   Shape( Shape&& other ) = default;
   Shape& operator=( Shape&& other ) = default;




   // ...
};
```

The move operations:

Option 1: Moved-from shapes are semantically equivalent to a nullptr

Consequence: It makes sense to have a default constructor

Example: `std::function`

A Type-Erased Solution

```
   std::unique_ptr<ShapeConcept> pimpl;

public:
   template< typename ShapeT >
   Shape( ShapeT shape )
      : pimpl{ std::make_unique<ShapeModel<ShapeT>>( std::move(shape) ) }
   {}

   // Copy operations
   Shape( Shape const& other )
      : pimpl( other.pimpl->clone() )
   {}

   Shape& operator=( Shape const& other )
   {
      other.pimpl->clone().swap( pimpl );
      return *this;
   }



   // Move operations
   //Shape( Shape&& other );
   //Shape& operator=( Shape&& other );




   // ...
};
```

The move operations:

Option 1: Moved-from shapes are semantically equivalent to a nullptr

Option 2: Move remains undefined, copy serves as a fallback

Consequence: This means that the move operations are NOT noexcept!

A Type-Erased Solution

```
   std::unique_ptr<ShapeConcept> pimpl;

public:
   template< typename ShapeT >
   Shape( ShapeT shape )
      : pimpl{ std::make_unique<ShapeModel<ShapeT>>( std::move(shape) ) }
   {}

   // Copy operations
   Shape( Shape const& other )
      : pimpl( other.pimpl->clone() )
   {}

   Shape& operator=( Shape const& other )
   {
      other.pimpl->clone().swap( pimpl );
      return *this;
   }



   // Move operations
   //Shape( Shape&& other );
   Shape& operator=( Shape&& other ) noexcept
   {
      pimpl.swap( other.pimpl );
      return *this;
   }

   // ...
};
```

The move operations:

Option 1: Moved-from shapes are semantically equivalent to a nullptr

Option 2: Move remains undefined, copy serves as a fallback

Option 3: The move constructor is undefined, the move assignment operator is implemented in terms of swap

There is not THE ONE solution, but it depends on the semantics you want to express.

A Type-Erased Solution

```
   // ...
};


void serialize( Circle const&, /*...*/ );
void draw( Circle const&, /*...*/ );

void serialize( Square const&, /*...*/ );
void draw( Square const&, /*...*/ );


void drawAllShapes( std::vector<Shape> const& shapes )
{
   for( auto const& shape : shapes )
   {
      draw( shape );
   }
}


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 );
}
```

A Type-Erased Solution

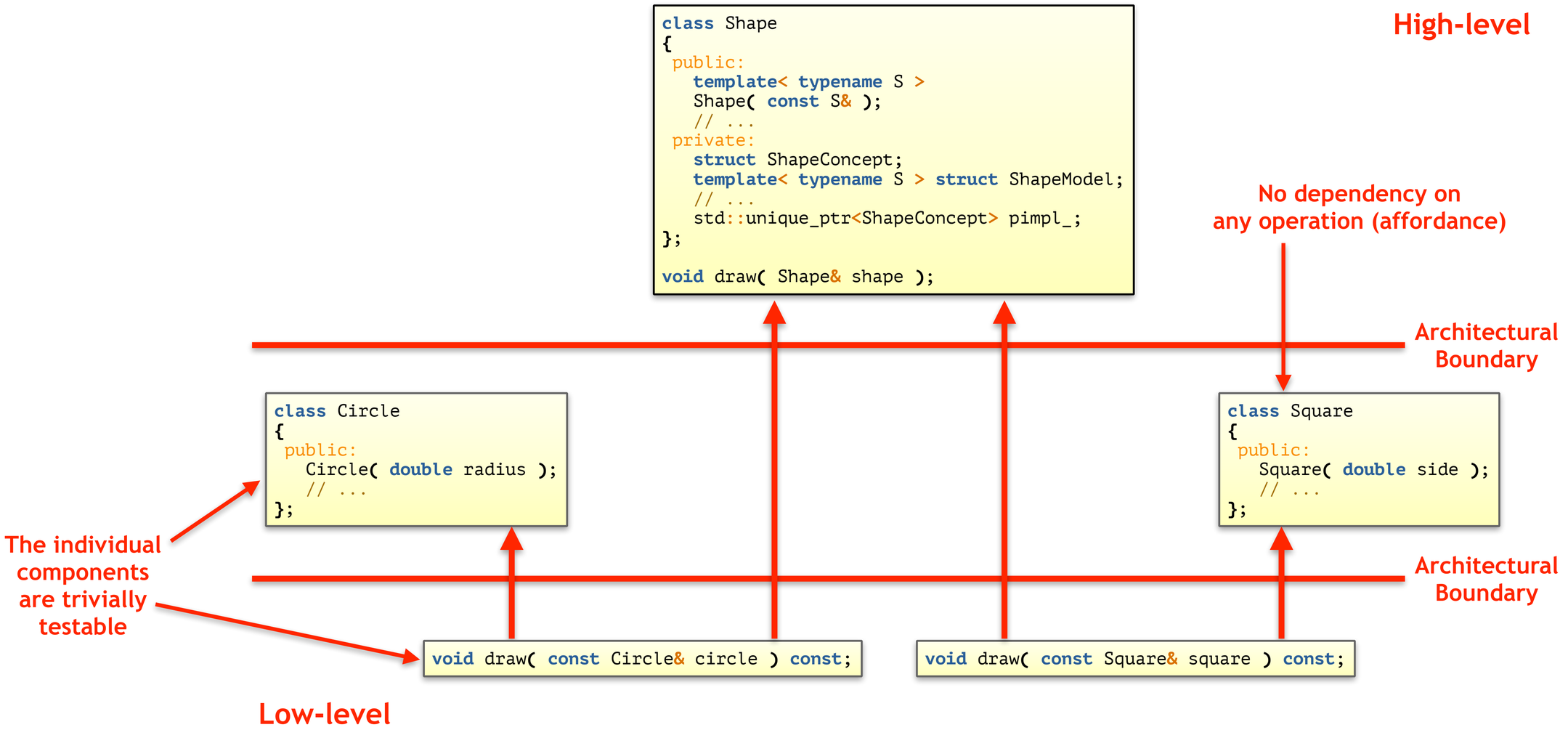
```
void serialize( Square const&, /*...*/ );
void draw( Square const&, /*...*/ );


void drawAllShapes( std::vector<Shape> const& shapes )
{
   for( auto const& shape : shapes )
   {
      draw( shape );
   }
}


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 manual dynamic allocation
- No manual life-time management
- value semantics
- Very simple code (KISS)
- Beautiful C++!

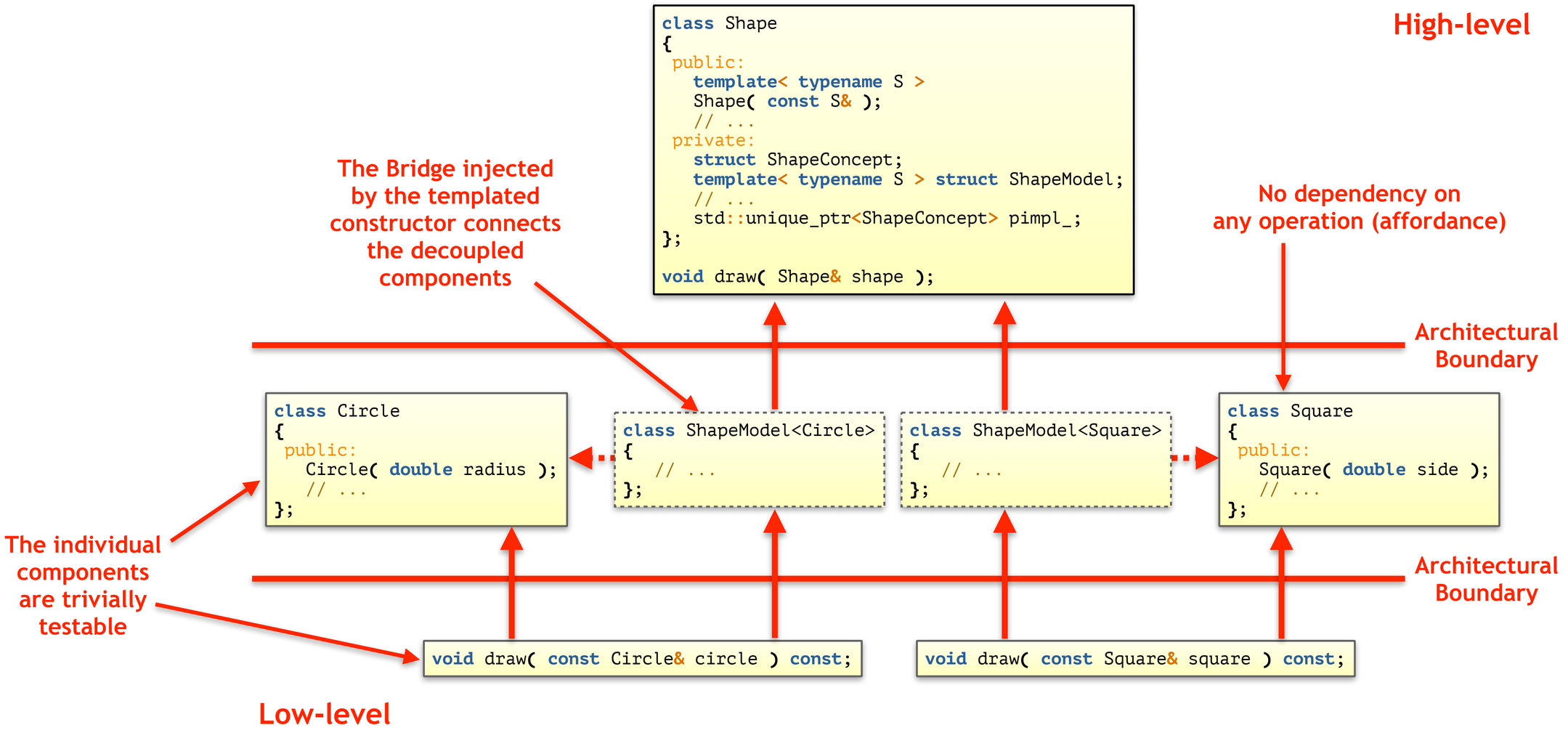
A Type-Erased Solution — Testability

What about testability?

A Type-Erased Solution — Design Analysis



A Type-Erased Solution — Design Analysis



A Type-Erased Solution

```
private:
  struct ShapeConcept
  {
     virtual ~ShapeConcept() = default;

     virtual void do_serialize( /*...*/ ) const = 0;
     virtual void do_draw( /*...*/ ) const = 0;
     virtual std::unique_ptr<ShapeConcept> clone() const = 0;
  };

  template< typename ShapeT >
  struct ShapeModel : public ShapeConcept
  {
     ShapeModel( ShapeT shape )
        : shape_{ std::move(shape) }
     {}

     std::unique_ptr<ShapeConcept> clone() const override
     {
        return std::make_unique<ShapeModel>(*this);
     }

     void do_serialize( /*...*/ ) const override
     {
        serialize( shape_, /*...*/ );
     }

     void do_draw( /*...*/ ) const override
     {
        draw( shape_, /*...*/ );
     }

     ShapeT shape_;
  };
```

A Type-Erased Solution

```
};

template< typename ShapeT
        , typename DrawStrategy >
struct ExtendedModel : public Concept
{
   explicit ExtendedModel( ShapeT shape
                         , DrawStrategy drawer )
      : shape_( std::move(shape) )
      , drawer_( std::move(drawer) )
   {}

   void do_draw() const override
   {
      drawer_( shape_, /*...*/ );
   }

   void do_serialize( /*...*/ ) const override
   {
      serialize( shape_, /*...*/ );
   }

   std::unique_ptr<Concept> clone() const override
   {
      return std::make_unique<ExtendedModel>(*this);
   }

   ShapeT shape_;
   DrawStrategy drawer_;
};

friend void serialize( Shape const& shape, /*...*/ )
{
```

The Strategy design pattern

Also deriving from the Concept base class

A Type-Erased Solution

```
  friend void draw( Shape const& shape, /*...*/ )
  {
     shape.pimpl->do_draw( /*...*/ );
  }

  std::unique_ptr<ShapeConcept> pimpl;

public:
  template< typename ShapeT >
  Shape( ShapeT shape )
     : pimpl{ std::make_unique<ShapeModel<ShapeT>>( std::move(shape) ) }
  {}

  template< typename ShapeT, typename DrawStrategy >
  Shape( ShapeT shape, DrawStrategy drawer )
     : pimpl( std::make_unique<ExtendedModel<ShapeT,DrawStrategy>>(
                  std::move(shape), std::move(drawer) ) )
  {}

  // Copy operations
  Shape( Shape const& other )
     : pimpl( other.pimpl->clone() )
  {}

  Shape& operator=( Shape const& other )
  {
     other.pimpl->clone().swap( pimpl );
     return *this;
  }

  // Move operations
  //Shape( Shape&& other );
  Shape& operator=( Shape&& other ) noexcept
```

The point of dependency injection

A Type-Erased Solution

```
void drawAllShapes( std::vector<Shape> const& shapes )
{
   for( auto const& shape : shapes )
   {
      draw( shape );
   }
}


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 }, [/*...*/]( Circle const& circle, /*...*/ ){
      /* ... Implementing the logic for drawing a circle ... */
   } );

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

A Type-Erased Solution — Summary

Amazing job! We have used Type Erasure to …

- … extract implementation details (SRP);
- … create the opportunity for easy extension (OCP);
- … separate interfaces (ISP);
- … reduce duplication (DRY);
- … remove all dependencies to operations (affordances);
- … remove all inheritance hierarchies;
- … remove all pointers;
- … remove all manual dynamic allocations;
- … remove all manual lifetime management;
- … improve performance.

A Type-Erased Solution — Summary

Amazing job! We have used Type Erasure to …

- … extract implementation details (SRP);
- … create the opportunity for easy extension (OCP);
- … separate interfaces (ISP);
- … reduce duplication (DRY);
- … remove all dependencies to operations (affordances);
- … remove all inheritance hierarchies;
- … remove all pointers;
- … remove all manual dynamic allocations;
- … remove all manual lifetime management;
- … improve performance.

Performance Comparison

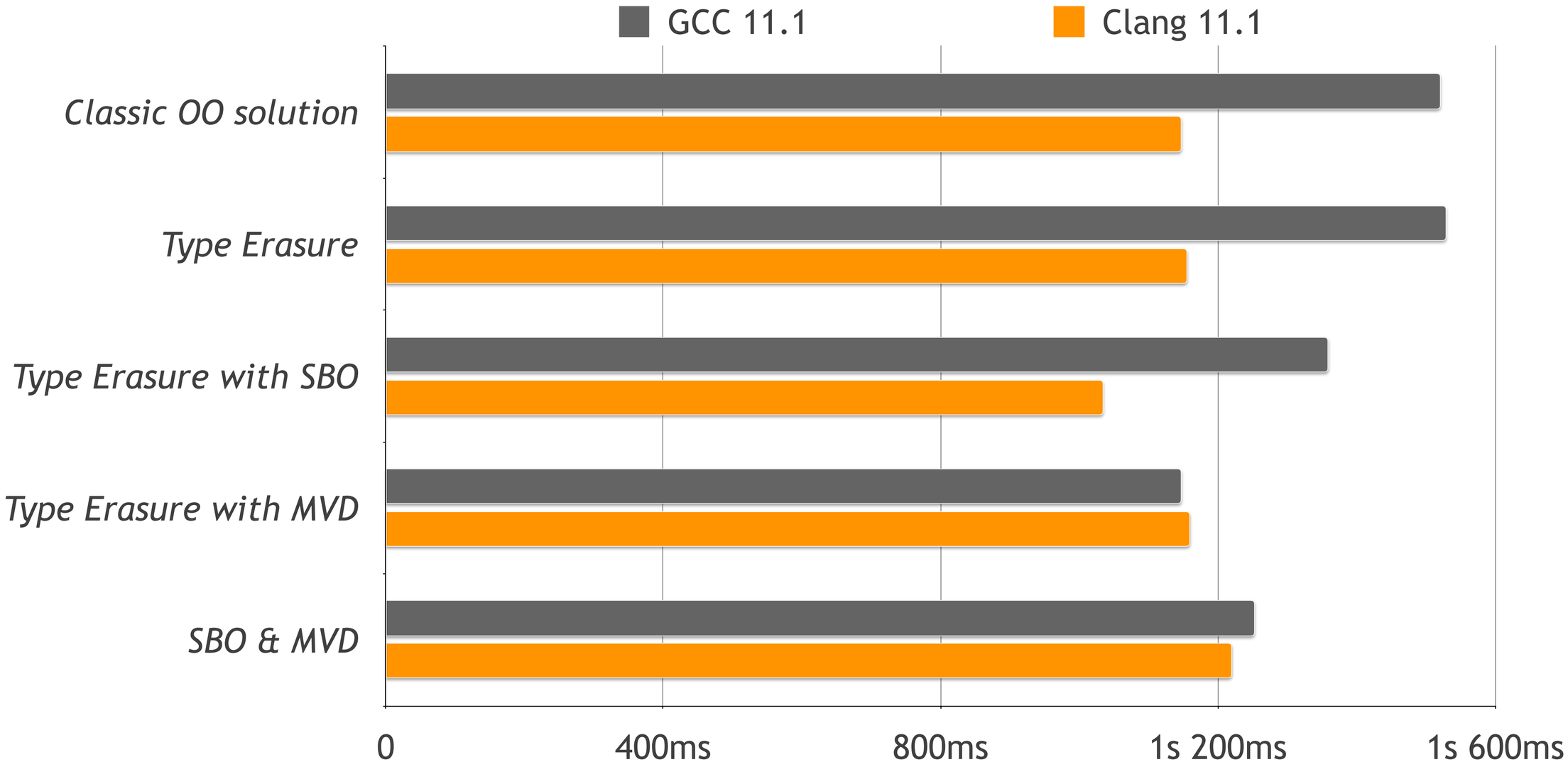
Performance … *sigh*

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



Content

- The Motivation for Type Erasure
- A Basic Type Erasure Implementation
- Type Erasure with Small Buffer Optimization (SBO)
- Type Erasure with Manual Virtual Dispatch

A Type-Erased Solution

```
            draw( shape_, /*...*/ );
         }

         ShapeT shape_;
      };

      friend void serialize( Shape const& shape, /*...*/ )
      {
         shape.pimpl->do_serialize( /*...*/ );
      }

      friend void draw( Shape const& shape, /*...*/ )
      {
         shape.pimpl->do_draw( /*...*/ );
      }

      std::unique_ptr<ShapeConcept> pimpl;

   public:
      template< typename ShapeT >
      Shape( ShapeT const& x )
         : pimpl{ std::make_unique<ShapeModel<ShapeT>>( x ) }
      {}

      // Special member functions
      Shape( Shape const& s );
      Shape& operator=( Shape const& s );
      Shape( Shape&& s ) = default;
      Shape& operator=( Shape&& s ) = default;

      // ...
};
```

std::make_unique performs a dynamic memory allocation via new (even for tiny shapes)

A Type-Erased Solution with SBO

```
   {
      shape.pimpl->do_draw( /*...*/ );
   }

   Concept* pimpl() noexcept
   {
      return reinterpret_cast<Concept*>( buffer.data() );
   }

   const Concept* pimpl() const noexcept
   {
      return reinterpret_cast<const Concept*>( buffer.data() );
   }

   static constexpr size_t buffersize = 128UL;
   static constexpr size_t alignment  =  16UL;

   alignas(alignment) std::array<std::byte,buffersize> buffer;

public:
   template< typename ShapeT >
   Shape( ShapeT const& x )
   {
      using M = Model<ShapeT>;
      static_assert( sizeof(M) <= buffersize, "Given type is too large" );
      static_assert( alignof(M) <= alignment, "Given type is overaligned" );
      ::new (pimpl()) M( shape );
   }

   // Special member functions
  ~Shape() { pimpl()->~Concept(); }

   Shape( Shape const& other )
   {
```

A Type-Erased Solution with SBO

```
    {
       shape.pimpl->do_draw( /*...*/ );
    }

    Concept* pimpl() noexcept
    {
       return reinterpret_cast<Concept*>( buffer.data() );
    }

    const Concept* pimpl() const noexcept
    {
       return reinterpret_cast<const Concept*>( buffer.data() );
    }

    static constexpr size_t buffersize = 128UL;
    static constexpr size_t alignment  =  16UL;

    alignas(alignment) std::array<std::byte,buffersize> buffer;

  public:
    template< typename ShapeT >
    Shape( ShapeT const& x )
    {
       using M = Model<ShapeT>;
       static_assert( sizeof(M) <= buffersize, "Given type is too large" );
       static_assert( alignof(M) <= alignment, "Given type is overaligned" );
       ::new (pimpl()) M( shape );
    }

    // Special member functions
   ~Shape() { pimpl()->~Concept(); }

    Shape( Shape const& other )
    {
```

A Type-Erased Solution with SBO

```
   {
      shape.pimpl->do_draw( /*...*/ );
   }

   Concept* pimpl() noexcept
   {
      return reinterpret_cast<Concept*>( buffer.data() );
   }

   const Concept* pimpl() const noexcept
   {
      return reinterpret_cast<const Concept*>( buffer.data() );
   }

   static constexpr size_t buffersize = 128UL;
   static constexpr size_t alignment  =  16UL;

   alignas(alignment) std::array<std::byte,buffersize> buffer;

public:
   template< typename ShapeT >
   Shape( ShapeT const& x )
   {
      using M = Model<ShapeT>;
      static_assert( sizeof(M) <= buffersize, "Given type is too large" );
      static_assert( alignof(M) <= alignment, "Given type is overaligned" );
      ::new (pimpl()) M( shape );
   }

   // Special member functions
  ~Shape() { pimpl()->~Concept(); }

   Shape( Shape const& other )
   {
```

The Bridge Design Pattern

A Type-Erased Solution with SBO

```
   {
      shape.pimpl->do_draw( /*...*/ );
   }

   Concept* pimpl() noexcept
   {
      return reinterpret_cast<Concept*>( buffer.data() );
   }

   const Concept* pimpl() const noexcept
   {
      return reinterpret_cast<const Concept*>( buffer.data() );
   }

   static constexpr size_t buffersize = 128UL;
   static constexpr size_t alignment  =  16UL;

   alignas(alignment) std::array<std::byte,buffersize> buffer;

public:
   template< typename ShapeT >
   Shape( ShapeT const& x )
   {
      using M = Model<ShapeT>;
      static_assert( sizeof(M) <= buffersize, "Given type is too large" );
      static_assert( alignof(M) <= alignment, "Given type is overaligned" );
      ::new (pimpl()) M( shape );
   }

   // Special member functions
  ~Shape() { pimpl()->~Concept(); }

   Shape( Shape const& other )
   {
```

A Type-Erased Solution with SBO

```
class Shape
{
 private:
   struct ShapeConcept
   {
      virtual ~ShapeConcept() = default;

      virtual void do_serialize( /*...*/ ) const = 0;
      virtual void do_draw( /*...*/ ) const = 0;
      virtual void clone( Concept* memory ) const = 0;
      virtual void move( Concept* memory ) const = 0;
   };

   template< typename ShapeT >
   struct ShapeModel : public ShapeConcept
   {
      ShapeModel( ShapeT shape )
         : shape_{ std::move(shape) }
      {}

      void clone( Concept* memory ) const override
      {
         ::new (memory) Model(*this);
      }

      void move( Concept* memory ) const override
      {
         ::new (memory) Model(std::move(*this));
      }

      void do_serialize( /*...*/ ) const override
      {
```

The External Polymorphism Design Pattern

A Type-Erased Solution with SBO

```
class Shape
{
 private:
   struct ShapeConcept
   {
      virtual ~ShapeConcept() = default;

      virtual void do_serialize( /*...*/ ) const = 0;
      virtual void do_draw( /*...*/ ) const = 0;
      virtual void clone( Concept* memory ) const = 0;
      virtual void move( Concept* memory ) const = 0;
   };

   template< typename ShapeT >
   struct ShapeModel : public ShapeConcept
   {
      ShapeModel( ShapeT shape )
         : shape_{ std::move(shape) }
      {}

      void clone( Concept* memory ) const override
      {
         ::new (memory) Model(*this);
      }

      void move( Concept* memory ) const override
      {
         ::new (memory) Model(std::move(*this));
      }

      void do_serialize( /*...*/ ) const override
      {
```

The Prototype Design Pattern

Note the explicit use of the global placement new operator. This is recommended to avoid overloads of placement new.

A Type-Erased Solution with SBO

```
class Shape
{
 private:
   struct ShapeConcept
   {
      virtual ~ShapeConcept() = default;

      virtual void do_serialize( /*...*/ ) const = 0;
      virtual void do_draw( /*...*/ ) const = 0;
      virtual void clone( Concept* memory ) const = 0;
      virtual void move( Concept* memory ) const = 0;
   };

   template< typename ShapeT >
   struct ShapeModel : public ShapeConcept
   {
      ShapeModel( ShapeT shape )
         : shape_{ std::move(shape) }
      {}

      void clone( Concept* memory ) const override
      {
         ::new (memory) Model(*this);
      }

      void move( Concept* memory ) const override
      {
         ::new (memory) Model(std::move(*this));
      }

      void do_serialize( /*...*/ ) const override
      {
         serialize( shape_, /*...*/ );
```

A Type-Erased Solution with SBO

```
   }

   // Special member functions
  ~Shape() { pimpl()->~Concept(); }

   Shape( Shape const& other )
   {
      other.pimpl()->clone( pimpl() );
   }

   Shape& operator=( Shape const& other )
   {
      // Copy-and-swap idiom
      Shape copy( other );
      buffer.swap( copy.buffer );
      return *this;
   }

   Shape( Shape&& other )
   {
      other.pimpl()->move( pimpl() );
   }

   Shape& operator=( Shape&& other )
   {
      // Move-and-swap idiom
      Shape tmp( std::move(other) );
      buffer.swap( tmp.buffer );
      return *this;
   }

   // ...
};
```

We are now required to perform the destruction of shapes manually, since the compiler only sees an array of bytes

A Type-Erased Solution with SBO

```
   }

   // Special member functions
  ~Shape() { pimpl()->~Concept(); }

   Shape( Shape const& other )
   {
      other.pimpl()->clone( pimpl() );
   }

   Shape& operator=( Shape const& other )
   {
      // Copy-and-swap idiom
      Shape copy( other );
      buffer.swap( copy.buffer );
      return *this;
   }

   Shape( Shape&& other )
   {
      other.pimpl()->move( pimpl() );
   }

   Shape& operator=( Shape&& other )
   {
      // Move-and-swap idiom
      Shape tmp( std::move(other) );
      buffer.swap( tmp.buffer );
      return *this;
   }

   // ...
};
```

A Type-Erased Solution with SBO

```
   }

   // Special member functions
  ~Shape() { pimpl()->~Concept(); }

   Shape( Shape const& other )
   {
      other.pimpl()->clone( pimpl() );
   }

   Shape& operator=( Shape const& other )
   {
      // Copy-and-swap idiom
      Shape copy( other );
      buffer.swap( copy.buffer );
      return *this;
   }

   Shape( Shape&& other )
   {
      other.pimpl()->move( pimpl() );
   }

   Shape& operator=( Shape&& other )
   {
      // Move-and-swap idiom
      Shape tmp( std::move(other) );
      buffer.swap( tmp.buffer );
      return *this;
   }

   // ...
};
```

A Type-Erased Solution with SBO

```
   {
      shape.pimpl->do_draw( /*...*/ );
   }

   Concept* pimpl() noexcept
   {
      return reinterpret_cast<Concept*>( buffer.data() );
   }

   const Concept* pimpl() const noexcept
   {
      return reinterpret_cast<const Concept*>( buffer.data() );
   }

   static constexpr size_t buffersize = 128UL;
   static constexpr size_t alignment  =  16UL;

   alignas(alignment) std::array<std::byte,buffersize> buffer;

public:
   template< typename ShapeT >
   Shape( ShapeT const& x )
   {
      using M = Model<ShapeT>;
      static_assert( sizeof(M) <= buffersize, "Given type is too large" );
      static_assert( alignof(M) <= alignment, "Given type is overaligned" );
      ::new (pimpl()) M( shape );
   }

   // Special member functions
  ~Shape() { pimpl()->~Concept(); }

   Shape( Shape const& other )
   {
```

A Type-Erased Solution with SBO

```
template< size_t buffersize = 128UL, size_t alignment = 16UL >
class Shape
{
 private:
   struct ShapeConcept
   {
      virtual ~ShapeConcept() = default;

      virtual void do_serialize( /*...*/ ) const = 0;
      virtual void do_draw( /*...*/ ) const = 0;
      virtual void clone( Concept* memory ) const = 0;
      virtual void move( Concept* memory ) const = 0;
   };

   template< typename ShapeT >
   struct ShapeModel : public ShapeConcept
   {
      ShapeModel( ShapeT shape )
         : shape_{ std::move(shape) }
      {}

      void clone( Concept* memory ) const override
      {
         ::new (memory) Model(*this);
      }

      void move( Concept* memory ) const override
      {
         ::new (memory) Model(std::move(*this));
      }
```

A Type-Erased Solution with SBO

The Strategy Design Pattern
(aka Policy-based Design)

```
template< typename StoragePolicy >
class Shape
{
 private:
   struct ShapeConcept
   {
      virtual ~ShapeConcept() = default;

      virtual void do_serialize( /*...*/ ) const = 0;
      virtual void do_draw( /*...*/ ) const = 0;
      virtual void clone( Concept* memory ) const = 0;
      virtual void move( Concept* memory ) const = 0;
   };

   template< typename ShapeT >
   struct ShapeModel : public ShapeConcept
   {
      ShapeModel( ShapeT shape )
         : shape_{ std::move(shape) }
      {}

      void clone( Concept* memory ) const override
      {
         ::new (memory) Model(*this);
      }

      void move( Concept* memory ) const override
      {
         ::new (memory) Model(std::move(*this));
      }
```

A Type-Erased Solution with SBO

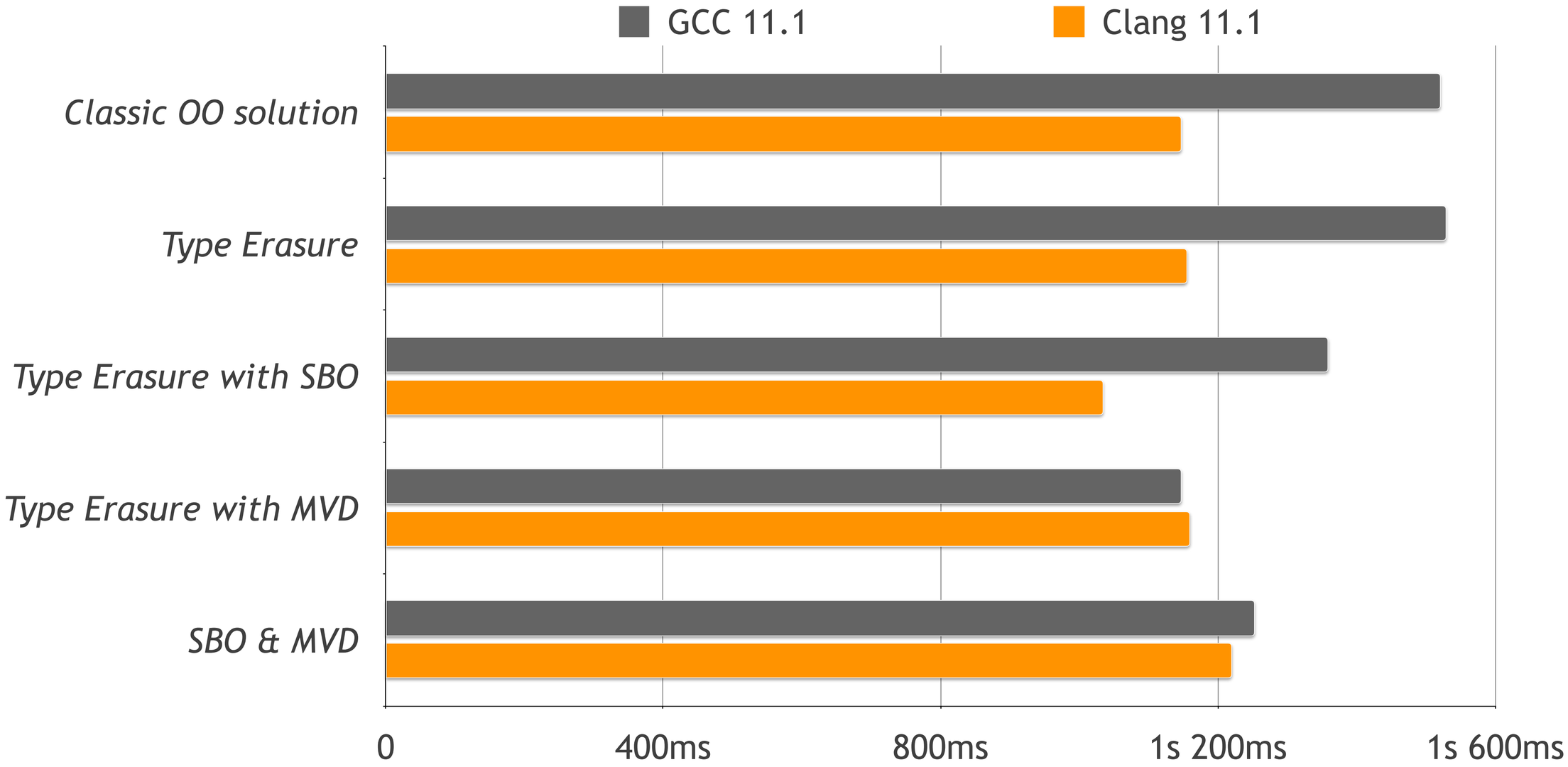
The Strategy Design Pattern
(aka Policy-based Design)

```
template< typename StoragePolicy >
class Shape
{
   // ...
};


class DynamicStorage { /*...*/ };  // Always performs dynamic allocation
class StackStorage { /*...*/ };    // Never performs dynamic allocation
class HybridStorage { /*...*/ };   // Performs dynamic allocation based on size


int main()
{
   Shape<DynamicStorage> shape1{ /*...*/ };

   Shape<StackStorage> shape2{ shape1 };                // Requires special operation

   Shape<HybridStorage> shape3{ std::move(shape1) };    // Requires special operation

   shape1 = shape2;                                     // Requires special operation

   shape1 = std::move(shape3);                          // Requires special operation
}
```

This discussion would dig too deep …

… and you remember what happens if you dig too deep …

Performance Comparison

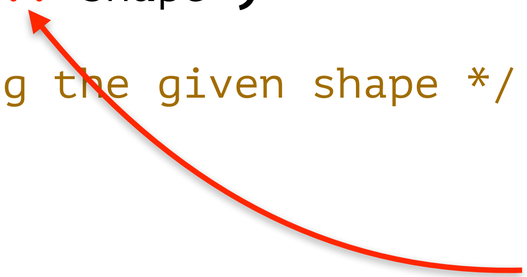


Content

- The Motivation for Type Erasure
- A Basic Type Erasure Implementation
- Type Erasure with Small Buffer Optimization (SBO)
- Type Erasure with Manual Virtual Dispatch

Type Erasure

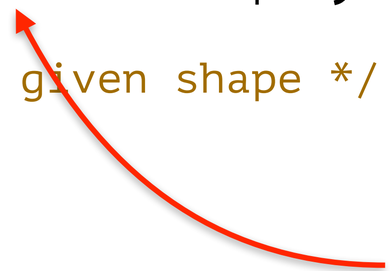
```
class Circle { /*...*/ };
class Square { /*...*/ };


void draw( ??? shape )
{
   /* Drawing the given shape */
}


int main()
{
   Circle circle( 2.3 );
   Square square( 1.2 );

   draw( circle );
   draw( square );

   // ...
}
```



How to provide a non-intrusive abstraction for passing any kind of shape to the draw function?

Type Erasure

```
class Circle { /*...*/ };
class Square { /*...*/ };


void draw( Shape const& shape )
{
   /* Drawing the given shape */
}


int main()
{
   Circle circle( 2.3 );
   Square square( 1.2 );

   draw( circle );
   draw( square );

   // ...
}
```

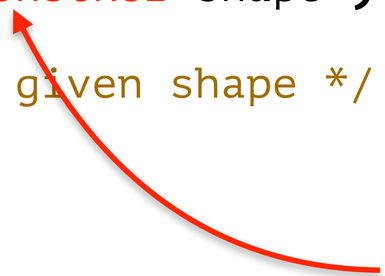


This owning abstraction would potentially require a memory allocation and a copy operation

What we need is a non-owning abstraction, which represents a reference …

Type Erasure

```
class Circle { /*...*/ };
class Square { /*...*/ };


void draw( ShapeConstRef shape )
{
   /* Drawing the given shape */
}


int main()
{
   Circle circle( 2.3 );
   Square square( 1.2 );

   draw( circle );
   draw( square );

   // ...
}
```



This non-owning abstraction does neither allocate nor copy

Type Erasure with Manual Virtual Dispatch

```
class ShapeConstRef
{
 public:









 private:









};
```

Type Erasure with Manual Virtual Dispatch

```
class ShapeConstRef
{
 public:
   template< typename ShapeT >
   ShapeConstRef( ShapeT const& shape )
   {}

 private:

};
```

Type Erasure with Manual Virtual Dispatch

```
class ShapeConstRef
{
 public:
   template< typename ShapeT >
   ShapeConstRef( ShapeT const& shape )
      : shape_{ &shape }



   {}

 private:









   void const* shape_{ nullptr };

};
```

Type Erasure with Manual Virtual Dispatch

```
class ShapeConstRef
{
 public:
   template< typename ShapeT >
   ShapeConstRef( ShapeT const& shape )
      : shape_{ std::addressof(shape) }



   {}

 private:









   void const* shape_{ nullptr };

};
```

Type Erasure with Manual Virtual Dispatch

```
class ShapeConstRef
{
 public:
   template< typename ShapeT >
   ShapeConstRef( ShapeT const& shape )
      : shape_{ std::addressof(shape) }



   {}

 private:




   using DrawOperation = void(void const*);

   void const* shape_{ nullptr };
   DrawOperation* draw_{ nullptr };
};
```

Type Erasure with Manual Virtual Dispatch

```
class ShapeConstRef
{
 public:
   template< typename ShapeT >
   ShapeConstRef( ShapeT const& shape )
      : shape_{ std::addressof(shape) }
      , draw_{ []( void const* shape ){
                  draw( *static_cast<ShapeT const*>(shape) );
               } }
   {}

 private:
```

```
   using DrawOperation = void(void const*);

   void const* shape_{ nullptr };
   DrawOperation* draw_{ nullptr };
};
```

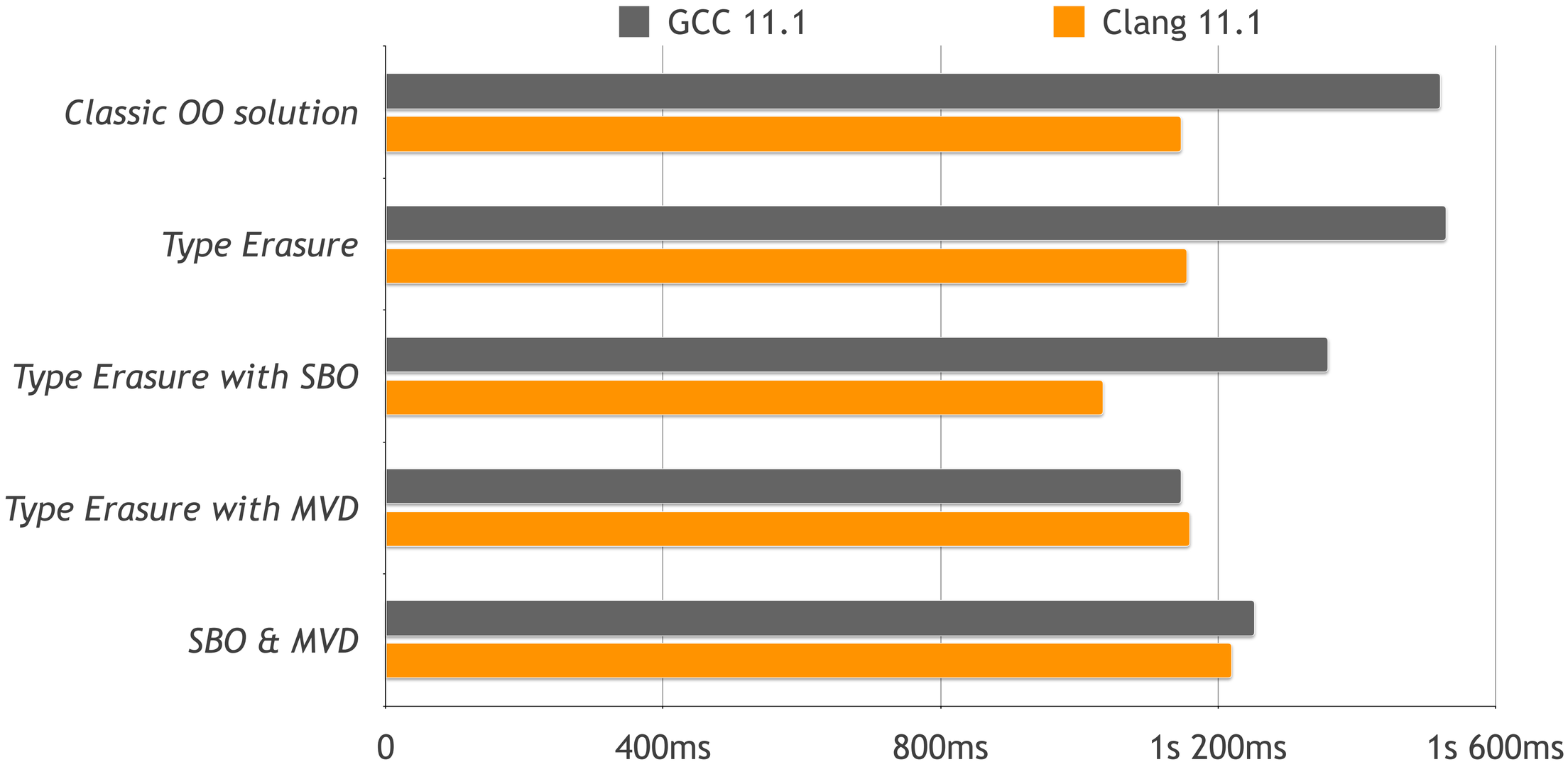
Type Erasure with Manual Virtual Dispatch

```
class ShapeConstRef
{
 public:
   template< typename ShapeT >
   ShapeConstRef( ShapeT const& shape )
      : shape_{ std::addressof(shape) }
      , draw_{ []( void const* shape ){
                  draw( *static_cast<ShapeT const*>(shape) );
               } }
   {}

 private:
   friend void draw( ShapeConstRef const& shape )
   {
      shape.draw_( shape.shape_ );
   }

   using DrawOperation = void(void const*);

   void const* shape_{ nullptr };
   DrawOperation* draw_{ nullptr };
};
```

Type Erasure with Manual Virtual Dispatch

```
class ShapeConstRef
{
 public:
   template< typename ShapeT >
   ShapeConstRef( ShapeT const& shape )
      : shape_{ std::addressof(shape) }
      , draw_{ []( void const* shape ){
                  draw( *static_cast<ShapeT const*>(shape) );
               } }
   {}

 private:
   friend void draw( ShapeConstRef const& shape )
   {
      shape.draw_( shape.shape_ );
   }

   using DrawOperation = void(void const*);

   void const* shape_{ nullptr };
   DrawOperation* draw_{ nullptr };
};
```

Performance Comparison



Summary

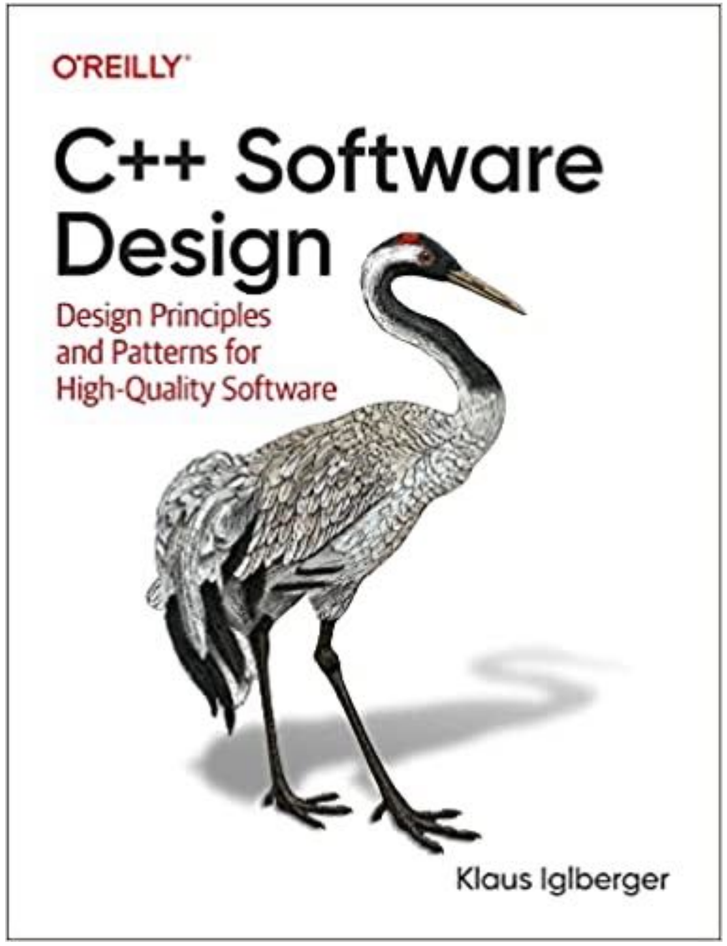
Type Erasure is …

- … a **templated constructor** plus …
- … a completely **non-virtual interface;**
- … **External Polymorphism + Bridge + Prototype;**
- … one of the most interesting **design patterns** today.

Type Erasure …

- … significantly reduces **dependencies**;
- … enables **value semantics**;
- … improves **performance**;
- … improves **readability** and **comprehensibility**;
- … eases **maintenance;**
- … is for good reason the default choice for dynamic polymorphism in many other languages.

Book Reference



www.oreilly.com

Breaking Dependencies:

Type Erasure - The Implementation Details

KLAUS IGLBERGER