Breaking Dependencies: Type Erasure - A Design Analysis

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 and SD tracks

Email: klaus.iglberger@gmx.de

Klaus Iglberger
Content

- The Challenge of Software Design
- The Naive Solution: Inheritance
- The Classic Solution: Design Patterns
- The Modern Solution: Type Erasure
The Challenge of Software Design

The Naive Solution: Inheritance

The Classic Solution: Design Patterns

The Modern Solution: Type Erasure
What is the root source of all problems in software development?

Change
The truth in our industry:

**Software must be adaptable to frequent changes**
The truth in our industry:

**Software must be adaptable to frequent changes**
What is the core problem of adaptable software and software development in general?

Dependencies
“Dependency is the key problem in software development at all scales.”
(Kent Beck, TDD by Example)
The Challenge of Software Design

The Naive Solution: Inheritance

The Classic Solution: Design Patterns

The Modern Solution: Type Erasure
Our Toy Problem: Shapes

"I’m tired of this example, but I don’t know any better one.”
(Lukas Bergdoll, MUC++ organizer)
Designing the Shape Hierarchy

![Class Diagram]

- **Shape**
  - virtual draw() = 0
  - virtual serialize() = 0

- **Circle**
  - virtual draw()

  - **OpenGLCircle**
    - virtual draw()

  - **MetalCircle**
    - virtual draw()

- **Square**
  - virtual draw()

  - **OpenGLSquare**
    - virtual draw()

  - **MetalSquare**
    - virtual draw()
Designing the Shape Hierarchy

Shape

virtual draw() = 0
virtual serialize() = 0

Circle

OpenGLLittleEndianCircle
virtual draw()
virtual serialize()

OpenGLBigEndianCircle
virtual draw()
virtual serialize()

MetalLittleEndianCircle
virtual draw()
virtual serialize()

MetalBigEndianCircle
virtual draw()
virtual serialize()

Square

OpenGLSquare
virtual draw()
virtual serialize()

MetalSquare
virtual draw()

OpenGLLittleEndianSquare
virtual serialize()

OpenGLBigEndianSquare
virtual serialize()
Using inheritance naively to solve our problem easily leads to ...

- ... many derived classes;
- ... ridiculous class names;
- ... deep inheritance hierarchies;
- ... duplication between similar implementations (DRY);
- ... (almost) impossible extensions (OCP);
- ... impeded maintenance.
”Inheritance is Rarely the Answer.
Delegate to Services: Has-A Trumps Is-A.”
(Andrew Hunt, David Thomas, The Pragmatic Programmer)
The Challenge of Software Design
The Naive Solution: Inheritance
The Classic Solution: Design Patterns
The Modern Solution: Type Erasure
The Solution: Design Patterns

A design pattern ...
- ... has a name;
- ... carries an intent;
- ... aims at reducing dependencies;
- ... provides some sort of abstraction;
- ... has proven to work over the years.

The Gang-of-Four (GoF) book: Origin of 23 of the most commonly used design patterns.
The Strategy Design Pattern

New “responsibilities” can be added without modifying any existing code; this fulfills the Open-Closed Principle (OCP).

This represents any concrete shape, i.e. Circle, Square, etc., and not a base class.

The aspect that changes is extracted and isolated; this fulfills the Single-Responsibility Principle (SRP).

```cpp
Circle
  draw()

DrawStrategy
  virtual draw(Circle) = 0

OpenGLStrategy
  virtual draw(Circle)

TestStrategy
  virtual draw(Circle)
```
The Strategy Design Pattern

- ... is not limited to object-oriented programming;
- ... is not limited to dynamic polymorphism;
- ... is not a language-specific idiom.
Examples from the Standard Library

```cpp
std::vector<int> numbers{ 1, 2, 3, 4, 5, 6, 7 };

std::accumulate( begin(numbers), end(numbers), 0,
                 std::plus<>{} );
```

Strategy
Examples from the Standard Library

template<
  class T,
  class Allocator = std::allocator<T>
>
class vector;

Strategy
Examples from the Standard Library

```cpp
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```
Examples from the Standard Library

template<
  class T,
  class Deleter = std::default_delete<T>
> class unique_ptr;
A Strategy-Based Solution

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

        virtual void draw( /*...*/ const = 0;
        virtual void serialize( /*...*/ const = 0;
            // ...
    }

    class Circle;

    class DrawCircleStrategy
    {
        public:
            virtual ~DrawCircleStrategy() {} 

            virtual void draw( Circle const& circle, /*...*/ const = 0;
                // ...
    }

    class Circle : public Shape
    {
        public:
            Circle( double rad,
                std::unique_ptr<DrawCircleStrategy> strategy )
                : radius{ rad } 
                , // ... Remaining data members
A Strategy-Based Solution

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

        virtual void draw( /*...*/ ) const = 0;
        virtual void serialize( /*...*/ ) const = 0;
        // ...
};

class Circle;

class DrawCircleStrategy
{
    public:
        virtual ~DrawCircleStrategy() {}

        virtual void draw( Circle const& circle, /*...*/ ) const = 0;
};

class Circle : public Shape
{
    public:
        Circle( double rad
            , std::unique_ptr<DrawCircleStrategy> strategy )
            : radius{ rad }
            , // ... Remaining data members
A Strategy-Based Solution

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

    virtual void draw( /*...*/ ) const = 0;
    virtual void serialize( /*...*/ ) const = 0;
    // ...
};

class Circle;

class DrawCircleStrategy
{
public:
    virtual ~DrawCircleStrategy() {}

    virtual void draw( Circle const& circle, /*...*/ ) const = 0;
};

class Circle : public Shape
{
public:
    Circle( double rad,
            std::unique_ptr<DrawCircleStrategy> strategy )
        : radius{ rad }
    , // ... Remaining data members
};

A Strategy-Based Solution

```cpp
class Circle : public Shape
{
public:
    Circle( double rad,
             std::unique_ptr<DrawCircleStrategy> strategy ) :
        radius{ rad } ,
        // ... Remaining data members
        drawing{ std::move(strategy) } {}

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

void draw( /*...*/ ) const override
{
    drawing->draw( this, /*...*/ );
}
void serialize( /*...*/ ) const override;
// ...

private:
    double radius;
    // ... Remaining data members
    std::unique_ptr<DrawStrategy> drawing;
};

class Square;
```
A Strategy-Based Solution

```cpp
// ... 

private:
    double radius;
    // ... Remaining data members
    std::unique_ptr<DrawStrategy> drawing;
};

class Square;

class DrawSquareStrategy
{
public:
    virtual ~DrawSquareStrategy() {}

    virtual void draw(Square const& square, /*...*/ ) const = 0;
};

class Square : public Shape
{
public:
    Square( double s
        , std::unique_ptr<DrawSquareStrategy> strategy )
        : side{ s }
        , // ... Remaining data members
        drawing{ std::move(strategy) }
    
    double getSide() const noexcept;
};
```
A Strategy-Based Solution

```cpp
class Square : public Shape {
public:
    Square( double s, std::unique_ptr<DrawSquareStrategy> strategy ) :
        side{ s }
    , // ... Remaining data members
    , drawing{ std::move(strategy) }
    {};

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

    void draw( /*...*/ ) const override {
        drawing->draw( this, /*...*/ );
    }
    void serialize( /*...*/ ) const override;
    // ...

private:
    double side;
    // ... Remaining data members
    std::unique_ptr<DrawSquareStrategy> drawing;
};

class OpenGLCircleStrategy : public DrawCircleStrategy {
    // ...
};
```
A Strategy-Based Solution

```cpp
private:
    double side;
    // ... Remaining data members
    std::unique_ptr<DrawSquareStrategy> drawing;
};

class OpenGLCircleStrategy : public DrawCircleStrategy
{
    public:
        virtual ~OpenGLStrategy() {} 

        void draw(Circle const& circle) const override;
};

class OpenGLSquareStrategy : public DrawSquareStrategy
{
    public:
        virtual ~OpenGLStrategy() {} 

        void draw(Square const& square) const override;
};

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

    // Creating some shapes
    Shapes shapes;
    shapes.emplace_back( std::make_unique<Circle>(2.0,
```
A Strategy-Based Solution

class OpenGLSquareStrategy : public DrawSquareStrategy
{
  public:
    virtual ~OpenGLStrategy() {};

    void draw( Square const& square ) const override;
};

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

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

  // Drawing all shapes
  drawAllShapes( shapes );
}
A Strategy-Based Solution — Design Analysis

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

class Circle : public Shape
{
  public:
    Circle(double rad, std::unique_ptr<DrawCircleStrategy> strategy);
    // ...
};
class DrawCircleStrategy
{
  public:
    virtual ~DrawCircleStrategy() {}
    virtual void draw(const Circle& circle) const = 0;
};
class OpenGLStrategy : public DrawCircleStrategy
{
  public:
    virtual void draw(const Circle& circle) const;
};
A Strategy-Based Solution — Summary

Good job! We have used the Strategy design pattern to...

- ... extract implementation details (SRP);
- ... create the opportunity for easy extension (OCP);
- ... separate interfaces (ISP);
- ... reduce duplication (DRY);
- ... limit the depth of the inheritance hierarchy;
- ... simplify maintainance.

But...

- ... performance is reduced due to a second indirection;
- ... performance is reduced due to many small, manual allocations;
- ... performance is affected due to many pointers;
- ... we need one Strategy for every operation (draw(), serialize(), ...);
- ... we need to manage lifetimes explicitly;
- ... we have a proliferation of inheritance hierarchies;
- ... circles, squares, etc. still know about all operations (affordances).
Content

- The Challenge of Software Design
- The Naive Solution: Inheritance
- The Classic Solution: Design Patterns
- The Modern Solution: Type Erasure
Towards a Value-Based Solution
Type Erasure — The Origin

Valued Conversions (C++ Report 12(7), July-August 2000)

FROM MECHANISM TO METHOD

Valued Conversions

“HOW WOULD YOU like to pay for that?” Good question. Digging deep into pockets, wallets, and bags uncovered a wealth of possibilities, a handful of different currencies and mechanisms to choose from: credit cards, debit cards, coins, bills, and a couple of IOUs, each form in some way substitutable for another when realizing monetary value.

Cash is the simplest, least troublesome form for small amounts and quick transactions. However, sifting through the metal and copper, it turned that many coins won a round. Well, that’s to any concrete form of inheritance. Substitutability here is based on values and conversions between values. Sometimes the use is implicit, at other times it must be made explicit. Conversions can be fully value preserving, widening, or narrowing. Widening conversions are always safe and typically acceptable (e.g., tipping), whereas narrowing conversions may not be (e.g., shortchanging tends to lead to exceptional or even undefined behavior).

Rescuing me from further metaphor stretching, the point-of-sale system and the assistant’s smile bunched into life.

Kevin Henney is an independent consultant and trainer based in the UK. He may be contacted at kevin@curbralan.com.
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.
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
};
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
};

struct ShapeConcept
{
  virtual ~ShapeConcept() {}

Circles and squares …
• … don’t need a base class;
• … don’t know about each other;
• … don’t know anything about their operations (affordances).
A Type-Erased Solution

```cpp
struct ShapeConcept
{
    virtual ~ShapeConcept() {}

    // ...
};

template<typename T>
struct ShapeModel : ShapeConcept
{
    ShapeModel(T&& value)
        : object(std::forward<T>(value)) {}

    // ...
    T object;
};
```
A Type-Erased Solution

```cpp
struct ShapeConcept
{
    virtual ~ShapeConcept() {}

    virtual void serialize /*...*/ const = 0;
    virtual void draw /*...*/ const = 0;
    // ...
};

template< typename T >
struct ShapeModel : ShapeConcept
{
    ShapeModel( T&& value )
        : object{ std::forward<T>(value) }
    {}
    // ...
    T object;
};
```
A Type-Erased Solution

```cpp
double
// ... Remaining data members
};

struct ShapeConcept
{
    virtual ~ShapeConcept() {};
    virtual void serialize( /*...*/ ) const = 0;
    virtual void draw( /*...*/ ) const = 0;
    // ...
};

template< typename T >
struct ShapeModel : ShapeConcept
{
    ShapeModel( T&& value )
        : object{ std::forward<T>(value) } {}
    // ...
    void serialize( /*...*/ ) const override
    {
        serialize( object, /*...*/ );
    }
    void draw( /*...*/ ) const override
    {
        draw( object, /*...*/ );
    }
    T object;
};
```

The implementation of the virtual functions in the ShapeModel class defines the affordances required by the type T.
A Type-Erased Solution

```cpp
struct ShapeConcept
{
    virtual ~ShapeConcept() {}

    virtual void serialize( /*...*/ ) const = 0;
    virtual void draw( /*...*/ ) const = 0;
    // ...
};

template< typename T >
struct ShapeModel : ShapeConcept
{
    ShapeModel( T&& value )
        : object{ std::forward<T>(value) } {}

    // ...

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

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

    T object;
};
```

The External Polymorphism Design Pattern
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


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 functionality onto these classes. One solution is to allow an application to extend the functionality of a C++ class without the need for inheritance. This is accomplished by the External Polymorphism Design Pattern.

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. int main (void) {
3.   SOCK_Stream stream; // Automatic storage
4.   acceptor.accept (stream, new StreamOp) ;
5.}

45
The External Polymorphism Design Pattern

- ... allows any object to be treated polymorphically;
- ... extracts implementation details (SRP);
- ... removes dependencies to operations (affordances);
- ... creates the opportunity for easy extension (OCP).
A Type-Erased Solution

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

    virtual void serialize(/*...*/ const = 0;
    virtual void draw(/*...*/ const = 0;
    // ...
};

template<typename T>
struct ShapeModel : ShapeConcept {
    ShapeModel( T&& value ) :
        object{ std::forward<T>(value) } {}
    // ...

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

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

    T object;
};
```
A Type-Erased Solution

```cpp
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

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

    void serialize( /*...*/ ) const override
    {
        serialize( object, /*...*/ );
    }
    void draw( /*...*/ ) const override
    {
        draw( object, /*...*/ );
    };
}
```

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

```cpp
void drawAllShapes(const std::vector<std::unique_ptr<ShapeConcept>>& 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);
}
```

```cpp
void serialize(Circle const&, /*...*/);
void draw(Circle const&, /*...*/);
void serialize(Square const&, /*...*/);
void draw(Square const&, /*...*/);
```
A Type-Erased Solution

```cpp
void translate( 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

```cpp
struct ShapeConcept
{
    virtual ~ShapeConcept() {}

    virtual void serialize( /*...*/ ) const = 0;
    virtual void draw( /*...*/ ) const = 0;
    // ...
};

template< typename T >
struct ShapeModel : ShapeConcept
{
    ShapeModel( T&& value )
        : object{ std::forward<T>(value) }
    {}

    // ...

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

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

    T object;
};
```
A Type-Erased Solution

class Shape
{
private:
    struct ShapeConcept
    {
        virtual ~ShapeConcept() {}

        virtual void serialize( /*...*/ ) const = 0;
        virtual void draw( /*...*/ ) const = 0;
        // ...
    };

template< typename T >
struct ShapeModel : ShapeConcept
{
    ShapeModel( T&& value )
    : object{ std::forward<T>(value) } 
    {}

    // ...

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

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

    T object;
};
A Type-Erased Solution

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

public:
  template<typename T>
  Shape(T const& x) : pimpl{ new ShapeModel<T>(x) } {}

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

  // ...
```

The Bridge Design Pattern

A templated constructor, creating a bridge
A Type-Erased Solution

```cpp
T object;
};
friend void serialize( Shape const& shape, /*...*/ )
{
    shape.pimpl->serialize( /*...*/ );
}
friend void draw( Shape const& shape, /*...*/ )
{
    shape.pimpl->draw( /*...*/ );
}
std::unique_ptr<ShapeConcept> pimpl;
public:
    template< typename T >
    Shape( T const& x ) :
        pimpl{ new ShapeModel<T>( x ) }
    {}
    // Special member functions
    Shape( Shape const& s );
    Shape& operator=( Shape const& s );
    Shape( Shape&& s ) = default;
    Shape& operator=( Shape&& s ) = default;
    // ...
};
```

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

T object;
};
friend void serialize( Shape const& shape, /*...*/ )
{
    shape.pimpl->serialize( /*...*/ );
}
friend void draw( Shape const& shape, /*...*/ )
{
    shape.pimpl->draw( /*...*/ );
}
std::unique_ptr<ShapeConcept> pimpl;

public:
    template< typename T >
    Shape( T const& x )
        : pimpl{ new ShapeModel<T>( x ) }
    {}

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

    // ...
};
A Type-Erased Solution

class Shape
{
private:
  struct ShapeConcept
  {
    virtual ~ShapeConcept() {}

    virtual void serialize( /*...*/ ) const = 0;
    virtual void draw( /*...*/ ) const = 0;
    // ...
  };

template< typename T >
struct ShapeModel : ShapeConcept
{
  ShapeModel( T&& value )
    : object{ std::forward<T>(value) }
  {}

  // ...

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

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

  T object;
}
A Type-Erasable Solution

```cpp
class Shape
{
private:
    struct ShapeConcept
    {
        virtual ~ShapeConcept() {};

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

    template<typename T>
    struct ShapeModel : ShapeConcept
    {
        ShapeModel( T&& value ) :
            object{ std::forward<T>(value) } {}

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

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

        void draw( /*...*/ ) const override
        {
            draw( object, /*...*/ );
        }
    }
};
```

The Prototype Design Pattern
A Type-Erased Solution

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

void translate( 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 );
void draw( Circle const&, /*...*/ );
void translate( 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 — Design Analysis

```cpp
class Shape
{
public:
    template< typename S >
    Shape( const S& );
    // ...
private:
    struct ShapeConcept;
    template< typename S > struct ShapeModel;
    // ...
    std::unique_ptr<ShapeConcept> pimpl_;
    void draw( Shape& shape );
};
```

```cpp
void draw( Shape& shape );
```

```cpp
class Circle
{
public:
    Circle( double radius );
    // ...
};
```

```cpp
void draw( const Circle& circle ) const;
```

```cpp
class Square
{
public:
    Square( double side );
    // ...
};
```

```cpp
void draw( const Square& square ) const;
```

Architectural Boundary

No dependency on any operation (affordance)

High-level

Low-level
A Type-Erased Solution — Design Analysis

The Bridge injected by the templated constructor

No dependency on any operation (affordance)

Architectural Boundary

Architectural Boundary

High-level

Low-level
Type Erasure — Motivation

https://twitter.com/ericniebler/status/1274031123522220033

If I could go back in time and had the power to change C++, rather than adding virtual functions, I would add language support for type erasure and concepts. Define a single-type concept, automatically generate a type-erasing wrapper for it.

7:27 PM · Jun 19, 2020 · Twitter for Android

6 Retweets 2 Quote Tweets 125 Likes

Eric Niebler #BLM @ericniebler · Jun 20
Replying to @ericniebler
This got more likes than I expected, and not one person saying "but but but my elaborate class hierarchies..." Huh.

4 Likes 15 Retweets
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.
Type Erasure — Performance Optimization

```
template< typename StoragePolicy >
class Shape  
{
    private:
    struct ShapeConcept;
    template< typename S > struct ShapeModel;
    // ...

    public:
    template< typename T >
    Shape( T const& shape );
    // ...

};
```

Strategy
Type Erasure – Performance Optimization
Large Performance Gains by Data Orientation Design Principles Made Practical Through Generic Programming Components

EDUARDO MADRID
Libraries

- Zoo (https://github.com/thecppzoo/zoo)
- Dyno (https://github.com/ldionne/dyno)
- Boost Type Erasure (https://www.boost.org)
- ...
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.
Breaking Dependencies:
Type Erasure - A Design Analysis

KLAUS IGLBERGER