Design Patterns

The Most Common Misconceptions

(2 of N)

Klaus Iglberger, Meeting C++ 2024

klaus.iglberger@gmx.de

C++ Trainer/Consultant

Author of “C++ Software Design”

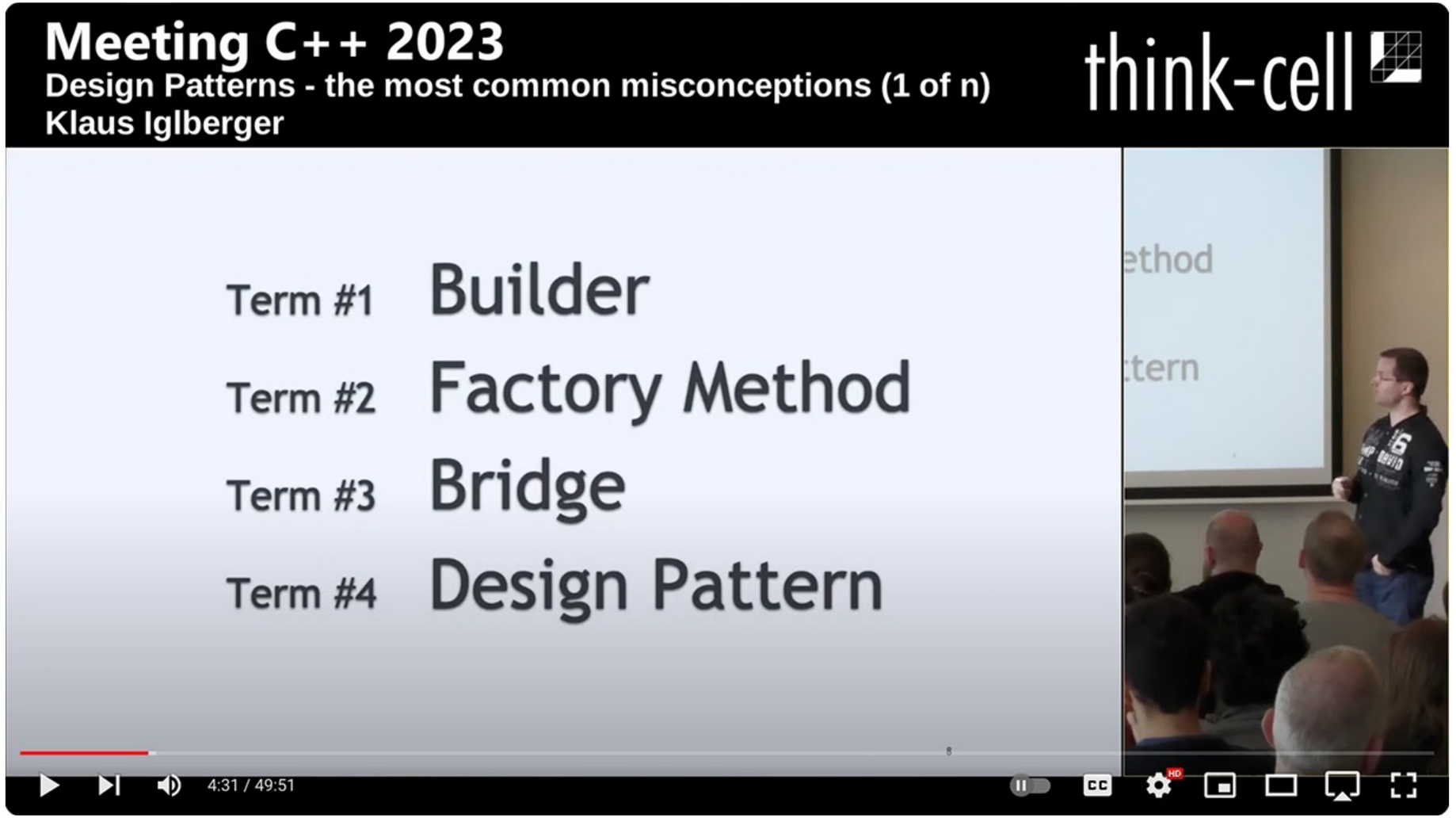
(Co-)Organizer of the Munich C++ user group

Chair of the CppCon Back-to-Basics track

Email: klaus.iglberger@gmx.de

Klaus Iglberger

The 2nd in a Row



Let’s again talk about

design patterns

and

virtual functions

The Overhead of Inheritance Hierarchies

The Overhead of Inheritance Hierarchies

Issue #1

CRTP

Issue #2

std::variant

Issue #1

CRTP

CRTP - Curiously Recurring Template Pattern

Curiously Recurring Template Patterns

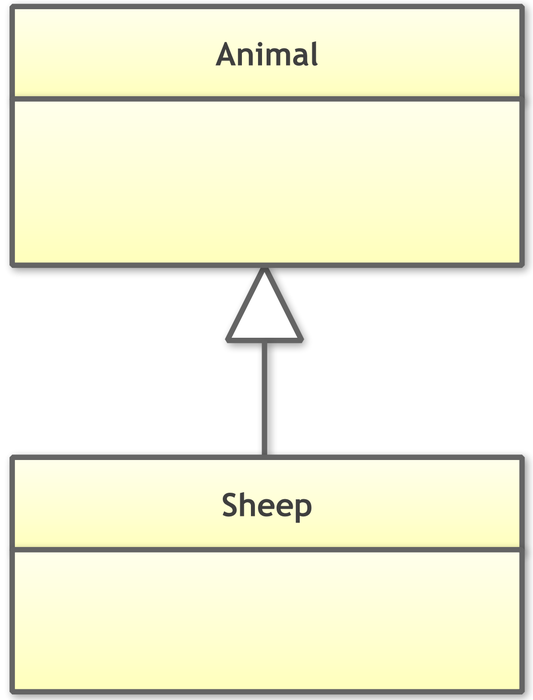
James O. Coplien
Software Production Research Dept., AT&T Bell Laboratories
1000 East Warrenville Rd., Naperville, IL 60566 USA
(708) 713-5384
cope@research.att.com

Abstract

Software design patterns capture recurring good practice in a domain. Many good patterns enjoy independent discovery by different people at different times. I have now seen three distinct uses of what I thought was an obscure template pattern. Each use arose in a unique domain; one instance was outside the C++ community. We can capture the technique in a pattern that solves a problem of factoring circular dependencies in code structure and behavior. The pattern form makes an otherwise opaque framework more accessible.

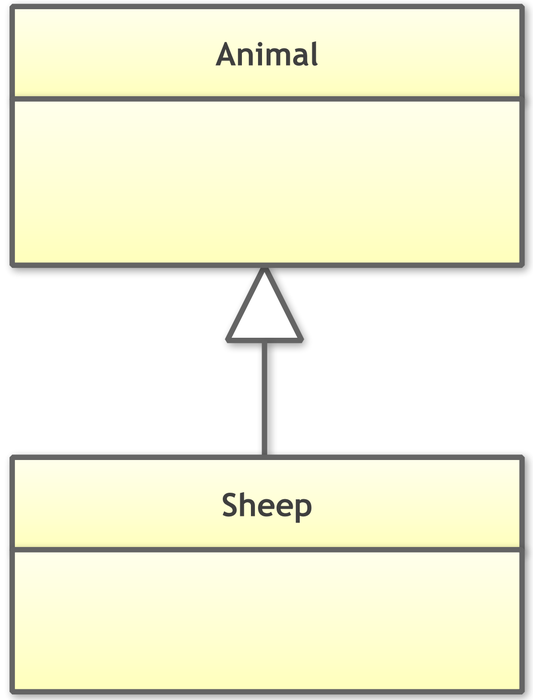
CRTP - Curiously Recurring Template Pattern

```
class Animal
{




};

class Sheep
{



};
```



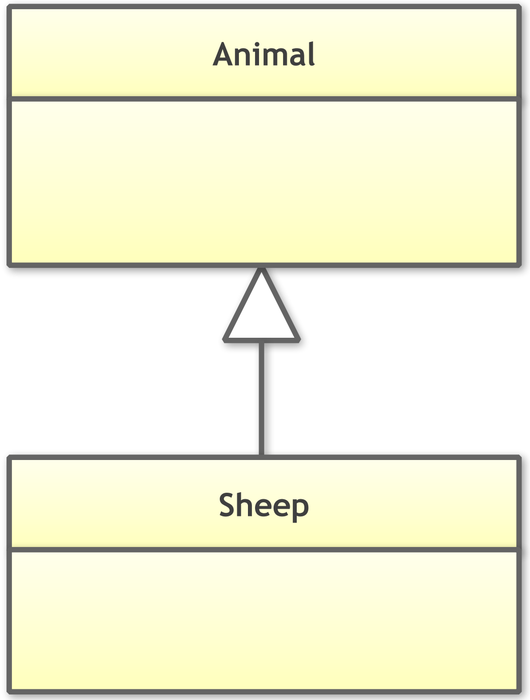
CRTP - Curiously Recurring Template Pattern

```
template< typename T >
class Animal
{









};

class Sheep
{



};
```



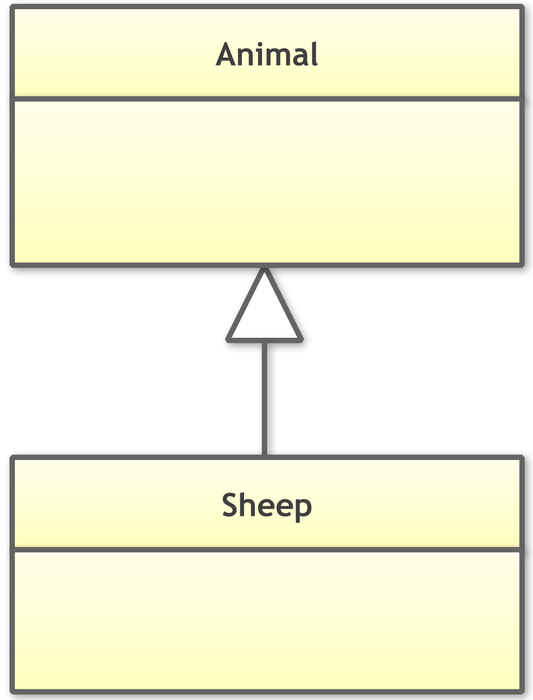
CRTP - Curiously Recurring Template Pattern

```
template< typename T >
class Animal
{




};

class Sheep : public Animal<Sheep>
{



};
```



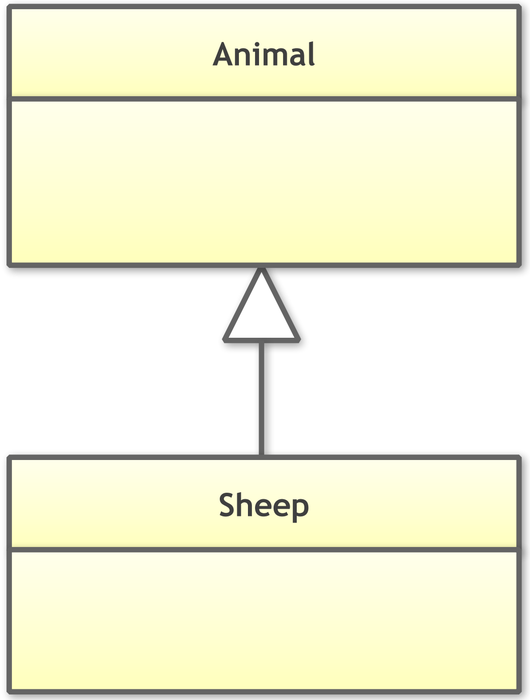
CRTP - Curiously Recurring Template Pattern

```
template< typename Derived >
class Animal
{




};

class Sheep : public Animal<Sheep>
{



};
```



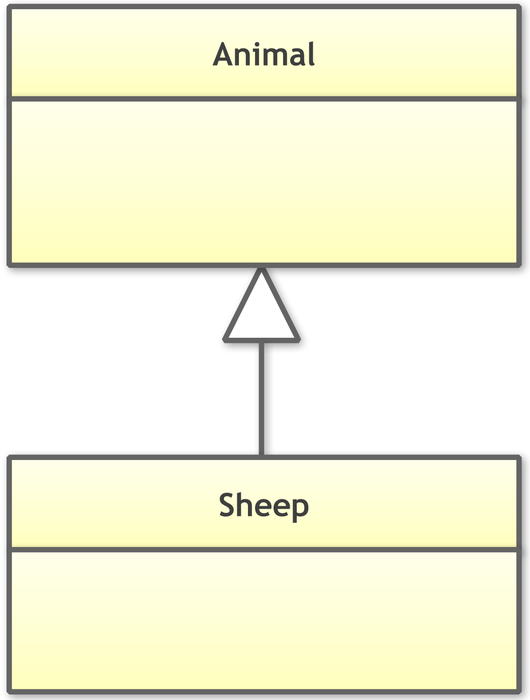
CRTP - Curiously Recurring Template Pattern

```
template< typename Derived >
class Animal
{
 private:
   Animal() = default;
   ~Animal() = default;









};

class Sheep : public Animal<Sheep>
{



};
```



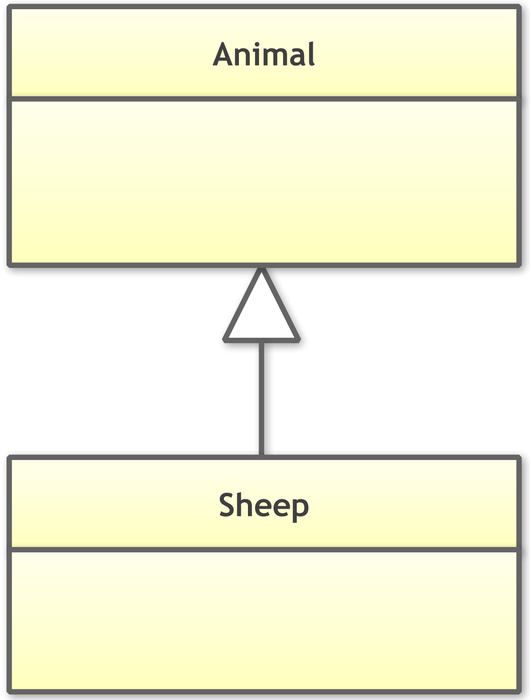
CRTP - Curiously Recurring Template Pattern

```
template< typename Derived >
class Animal
{
 private:
   Animal() = default;  // Protects against
   ~Animal() = default; // “wrong” Derived class
   friend Derived;




};

class Sheep : public Animal<Sheep>
{



};
```



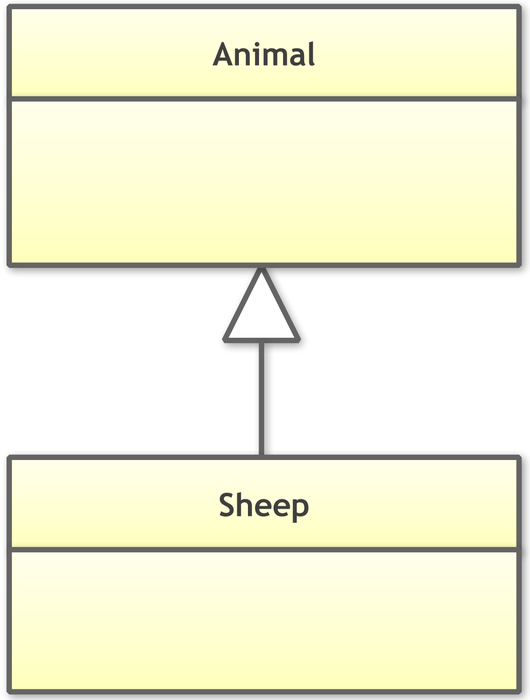
CRTP - Curiously Recurring Template Pattern

```
template< typename Derived >
class Animal
{
 private:
   Animal() = default;  // Protects against
   ~Animal() = default; // “wrong” Derived class
   friend Derived;









};

class Sheep : public Animal<Sheep>
{
 public:
   //~Sheep();  Remember the Rule-of-5


};
```



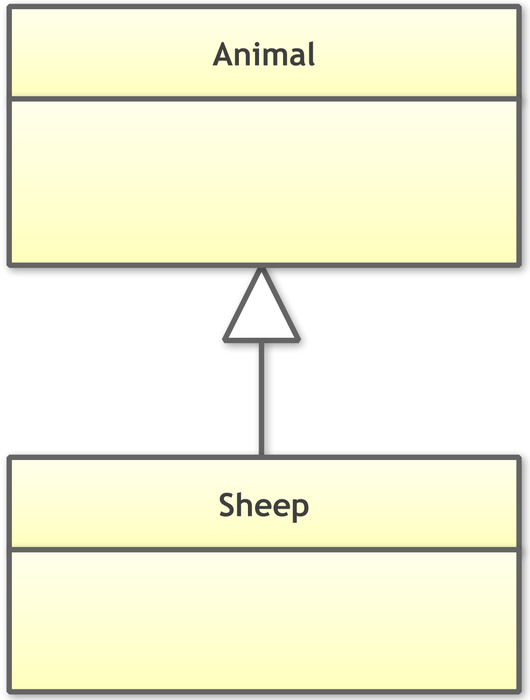
CRTP - Curiously Recurring Template Pattern

```
template< typename Derived >
class Animal
{
 private:
   Animal() = default;  // Protects against
   ~Animal() = default; // “wrong” Derived class
   friend Derived;

 public:
   void make_sound() const {
      // ...
   }
};

class Sheep : public Animal<Sheep>
{
 public:
   //~Sheep();  Remember the Rule-of-5


};
```



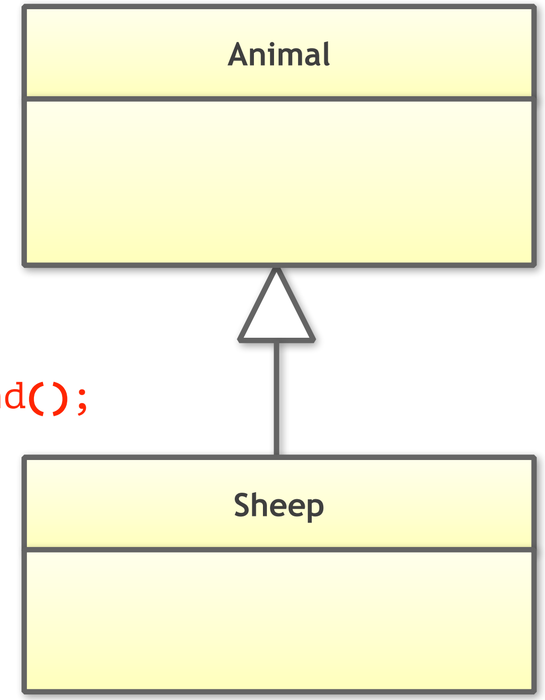
CRTP - Curiously Recurring Template Pattern

```
template< typename Derived >
class Animal
{
 private:
   Animal() = default;  // Protects against
   ~Animal() = default; // “wrong” Derived class
   friend Derived;

 public:
   void make_sound() const {
      // ...
   }
};

class Sheep : public Animal<Sheep>
{
 public:
   //~Sheep();  Remember the Rule-of-5
   // ...
   void make_sound() const { std::cout << "baa"; }
};
```



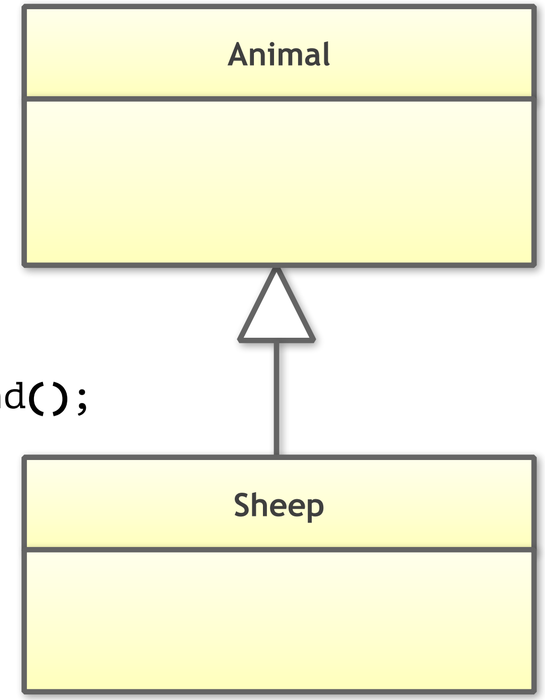
CRTP - Curiously Recurring Template Pattern

```
template< typename Derived >
class Animal
{
 private:
   Animal() = default;  // Protects against
   ~Animal() = default; // “wrong” Derived class
   friend Derived;

 public:
   void make_sound() const {
      static_cast<Derived const&>(*this).make_sound();
   }
};

class Sheep : public Animal<Sheep>
{
 public:
   //~Sheep();  Remember the Rule-of-5
   // ...
   void make_sound() const { std::cout << "baa"; }
};
```



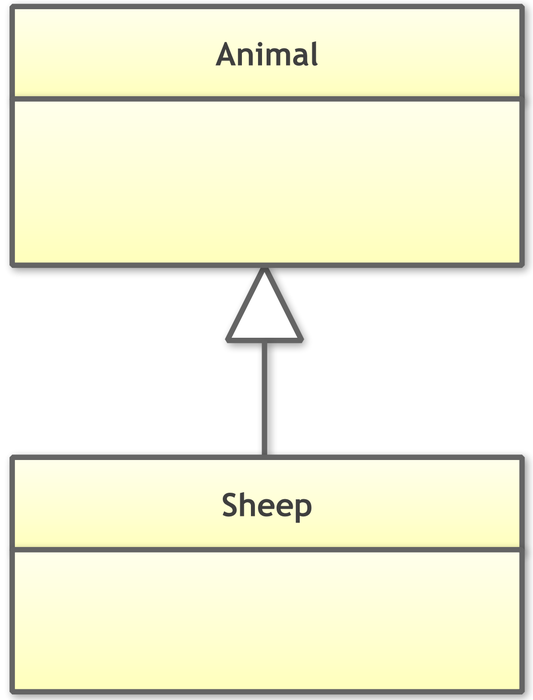
CRTP - Curiously Recurring Template Pattern

```
template< typename Derived >
class Animal
{
 private:
   Animal() = default;  // Protects against
   ~Animal() = default; // “wrong” Derived class
   friend Derived;

 public:
   void make_sound() const {
      static_cast<Derived const&>(*this).make_sound();
   }
};

class Sheep : public Animal<Sheep>
{
 public:
   //~Sheep();  Remember the Rule-of-5
   // ...
   void make_sound() const { std::cout << "baa"; }
};
```



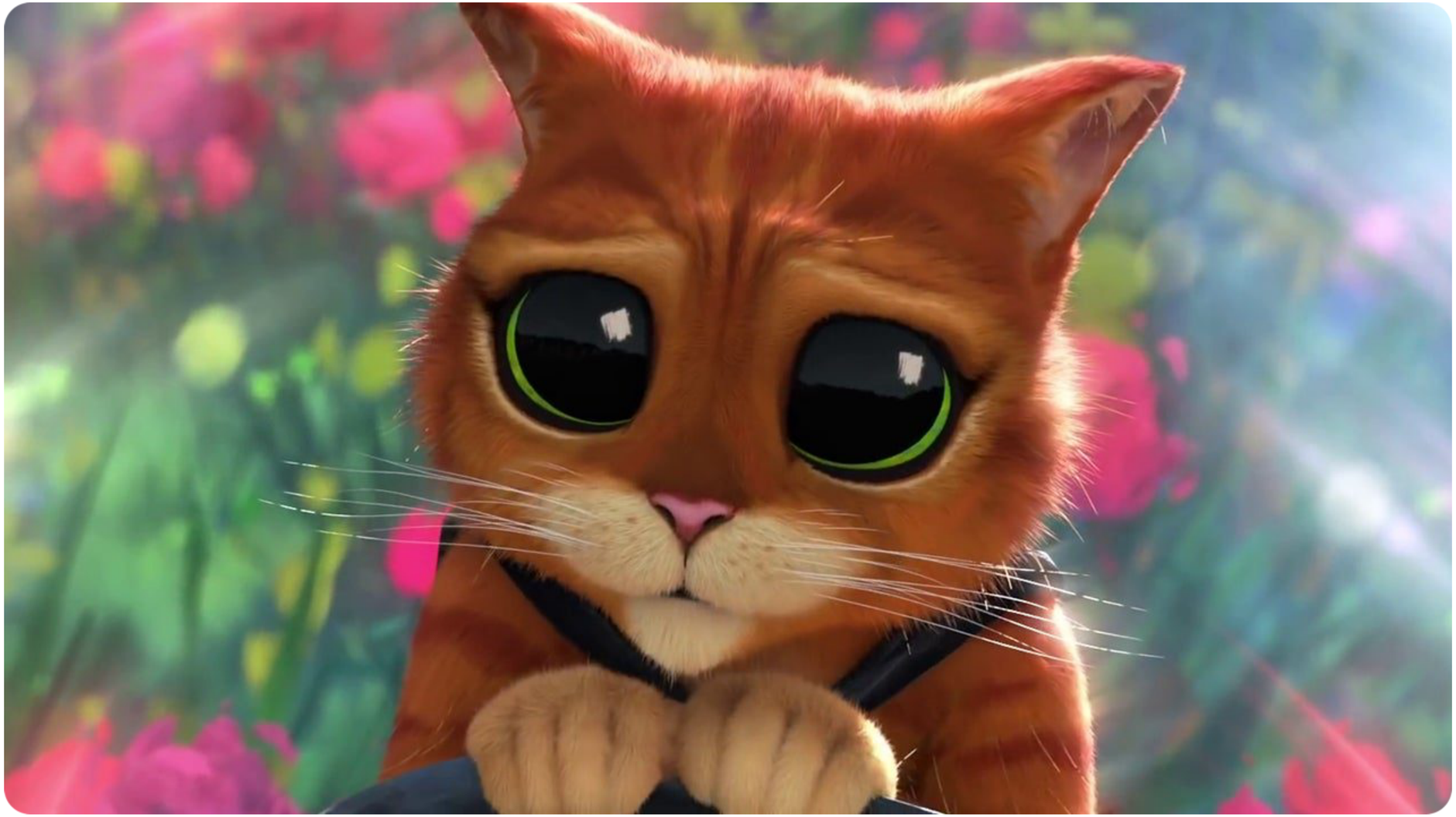
CRTP - Curiously Recurring Template Pattern

```
template< typename Derived >
class Animal
{
   // ...
};

class Sheep : public Animal<Sheep>
{
   // ...
};

template< typename Derived >
void print( Animal<Derived> const& animal )
{
   // ...
}
```



Works for all animals and does not trigger any virtual function call (functions might even be inlined)

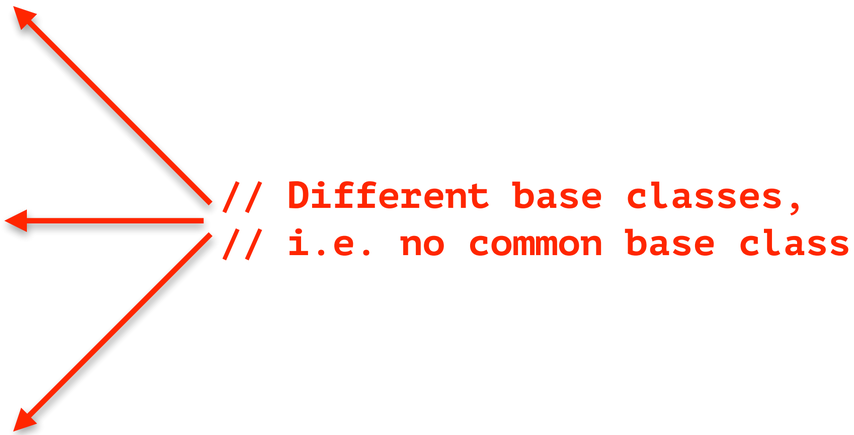
CRTP: The Replacement for Virtual Functions?



The Limitations of CRTP

1. Limitation: There Is No Common Base Class

```
template< typename T >
class Animal
{
   // ...
};

class Sheep : public Animal<Sheep>
{
   // ...
};

class Dog : public Animal<Dog>
{
   // ...
};

class Cat : public Animal<Cat>
{
   // ...
};
```

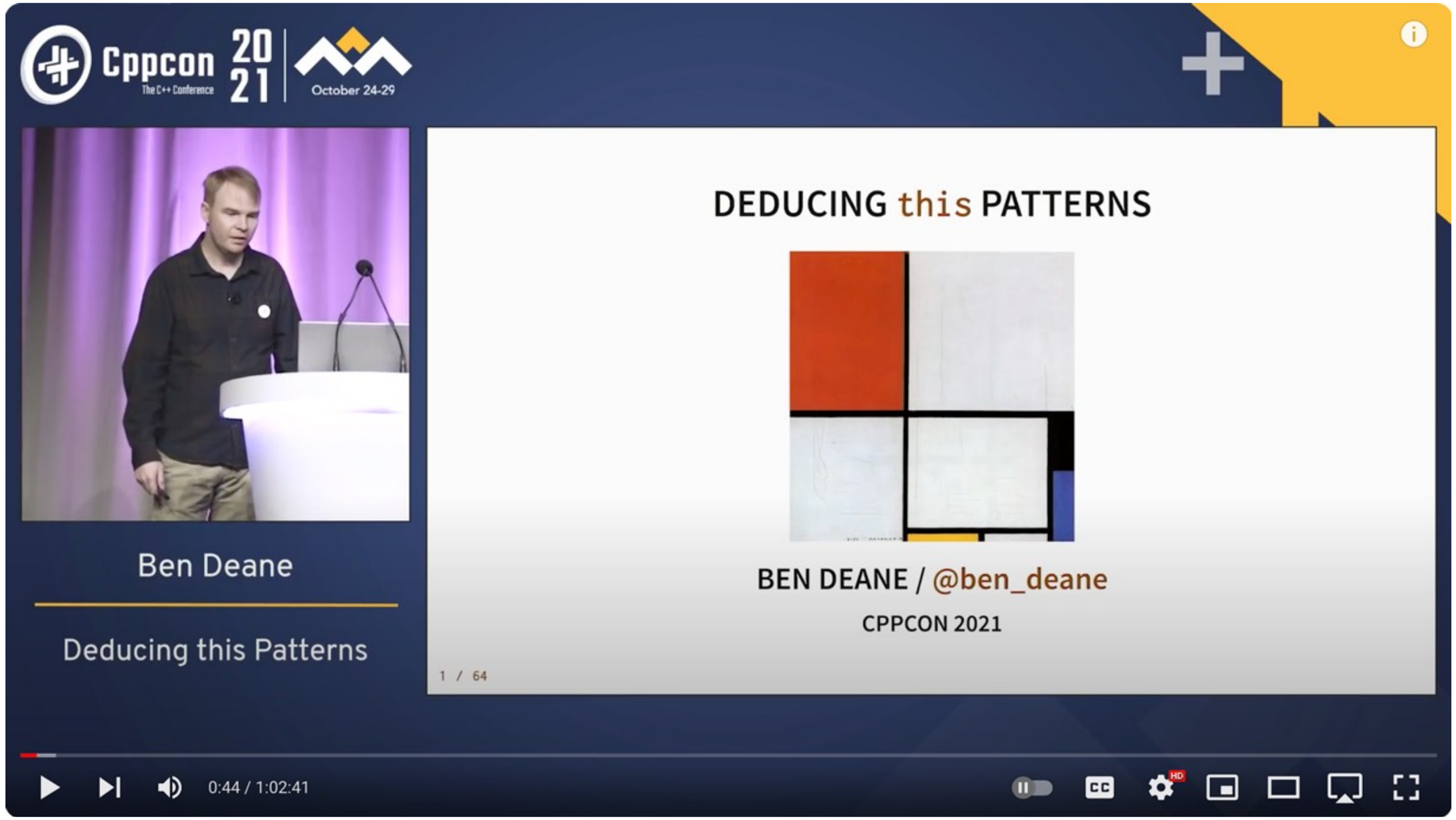


2. Limitation: Everything Is A Template

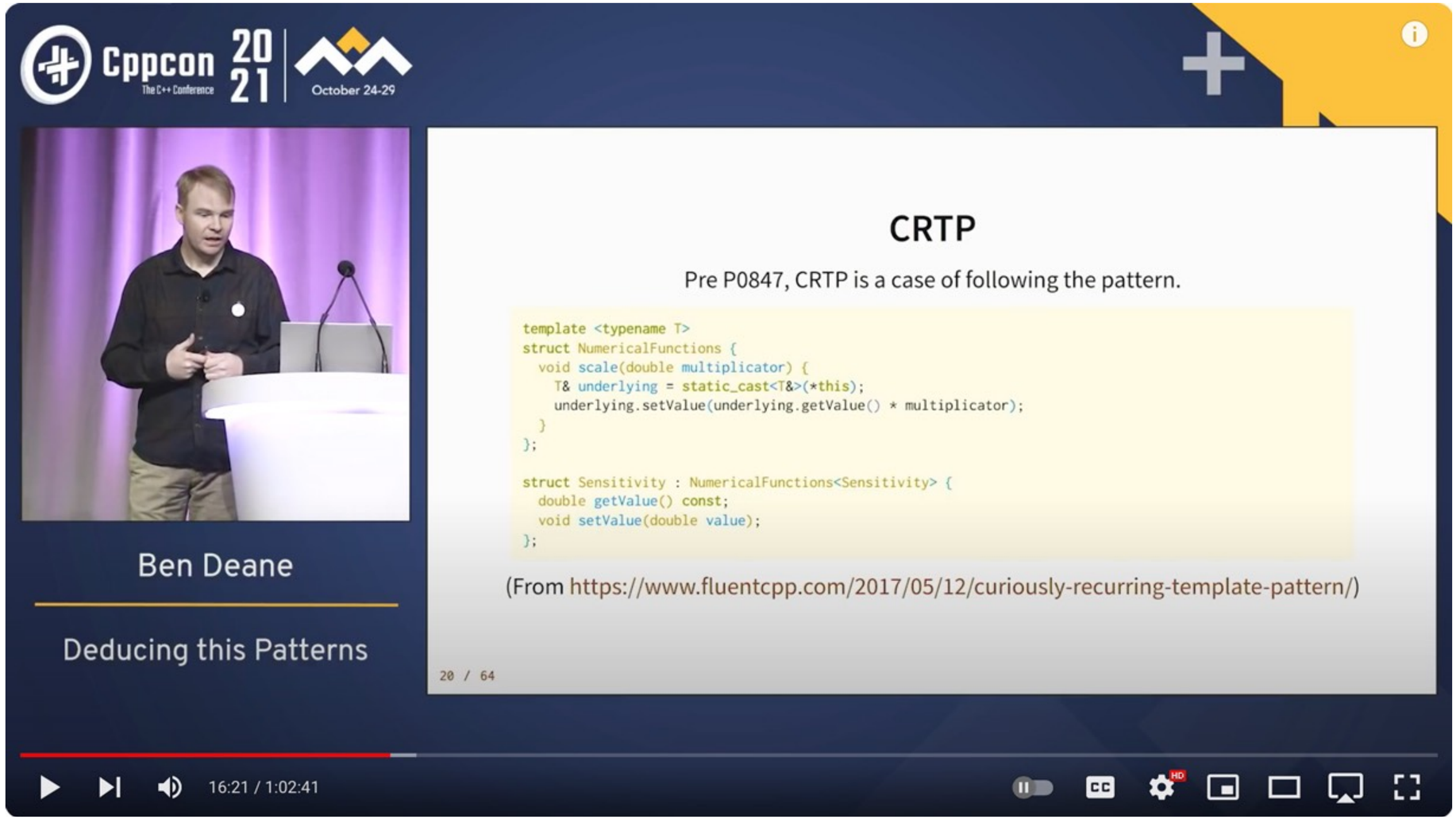
```
template< typename Derived >
class Animal
{
   // ...
};

class Sheep : public Animal<Sheep>
{
   // ...
};

template< typename Derived >
void print( Animal<Derived> const& animal )
{
   // ...
}
```

This function must be a function template to take any kind of animal. Thus CRTP can act like a virus: Everything touching CRTP is or becomes a template (including higher compile times).

The Future of CRTP?



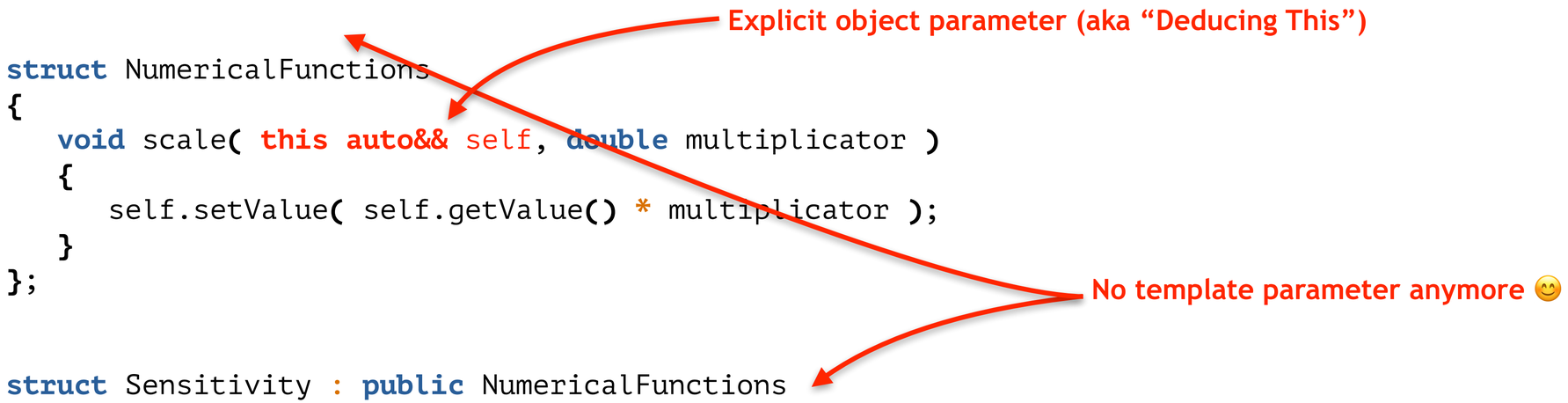
The Future of CRTP?



Adding Functionality with CRTP

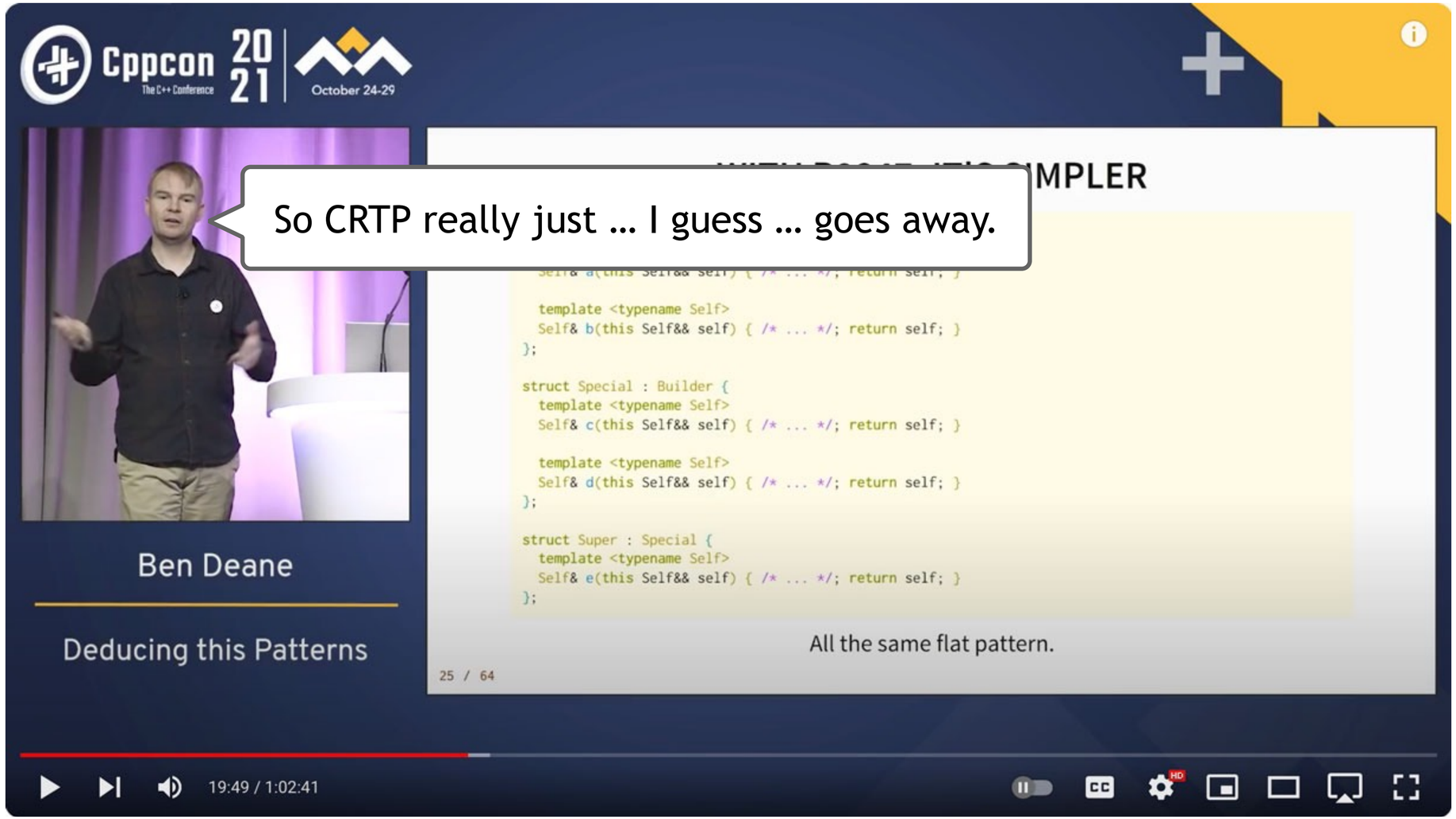
```
template< typename Derived >
struct NumericalFunctions
{
   void scale( double multiplicator )
   {
      Derived& underlying = static_cast<Derived&>(*this);
      underlying.setValue( underlying.getValue() * multiplicator );
   }
};

struct Sensitivity : public NumericalFunctions<Sensitivity>
{
   double getValue() const { return value; }
   void setValue( double v ) { value = v; }
   double value;
};

int main()
{
   Sensitivity s{ 1.2 };
   s.scale( 2.0 );

   std::println( std::cout, "s.getValue() = {}", s.getValue() );
}
```

Adding Functionality with C++23

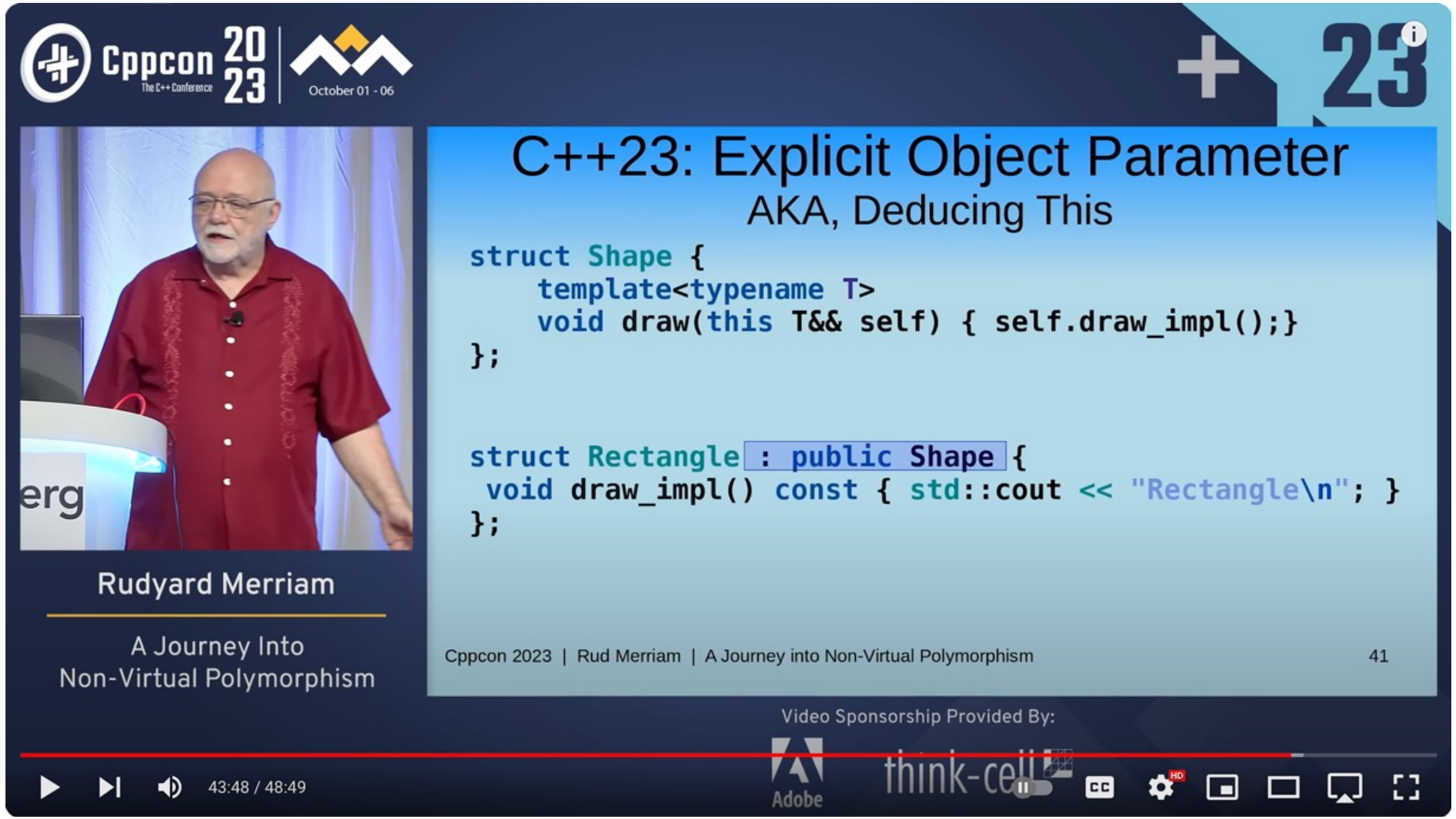



```cpp
struct NumericalFunctions
{
   void scale( this auto&& self, double multiplicator )
   {
      self.setValue( self.getValue() * multiplicator );
   }
};


struct Sensitivity : public NumericalFunctions
{
   double getValue() const { return value; }
   void setValue( double v ) { value = v; }
   double value;
};

int main()
{
   Sensitivity s{ 1.2 };
   s.scale( 2.0 );

   std::println( std::cout, "s.getValue() = {}", s.getValue() );
}
```

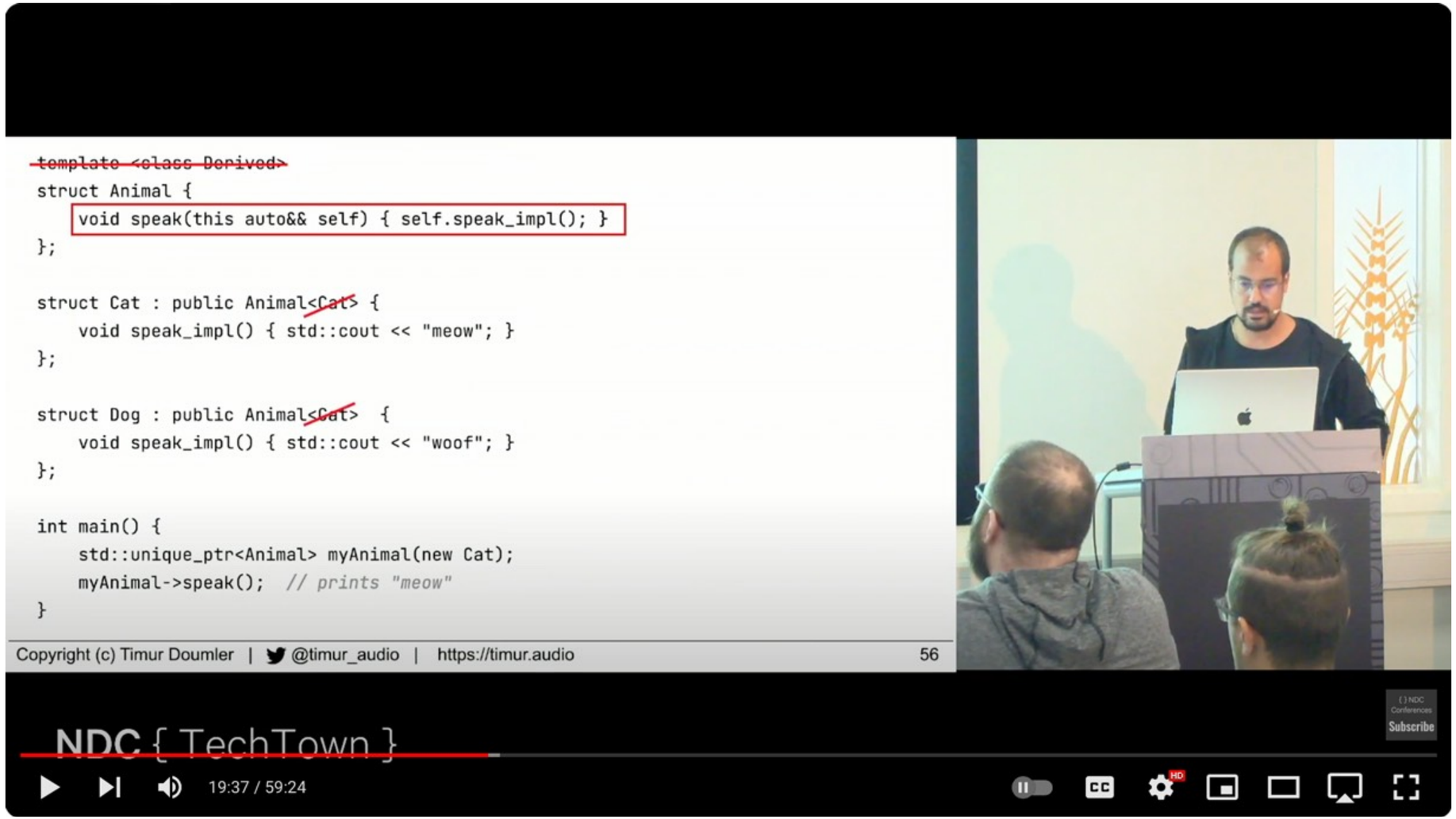
The Future of CRTP?



The Future of CRTP?



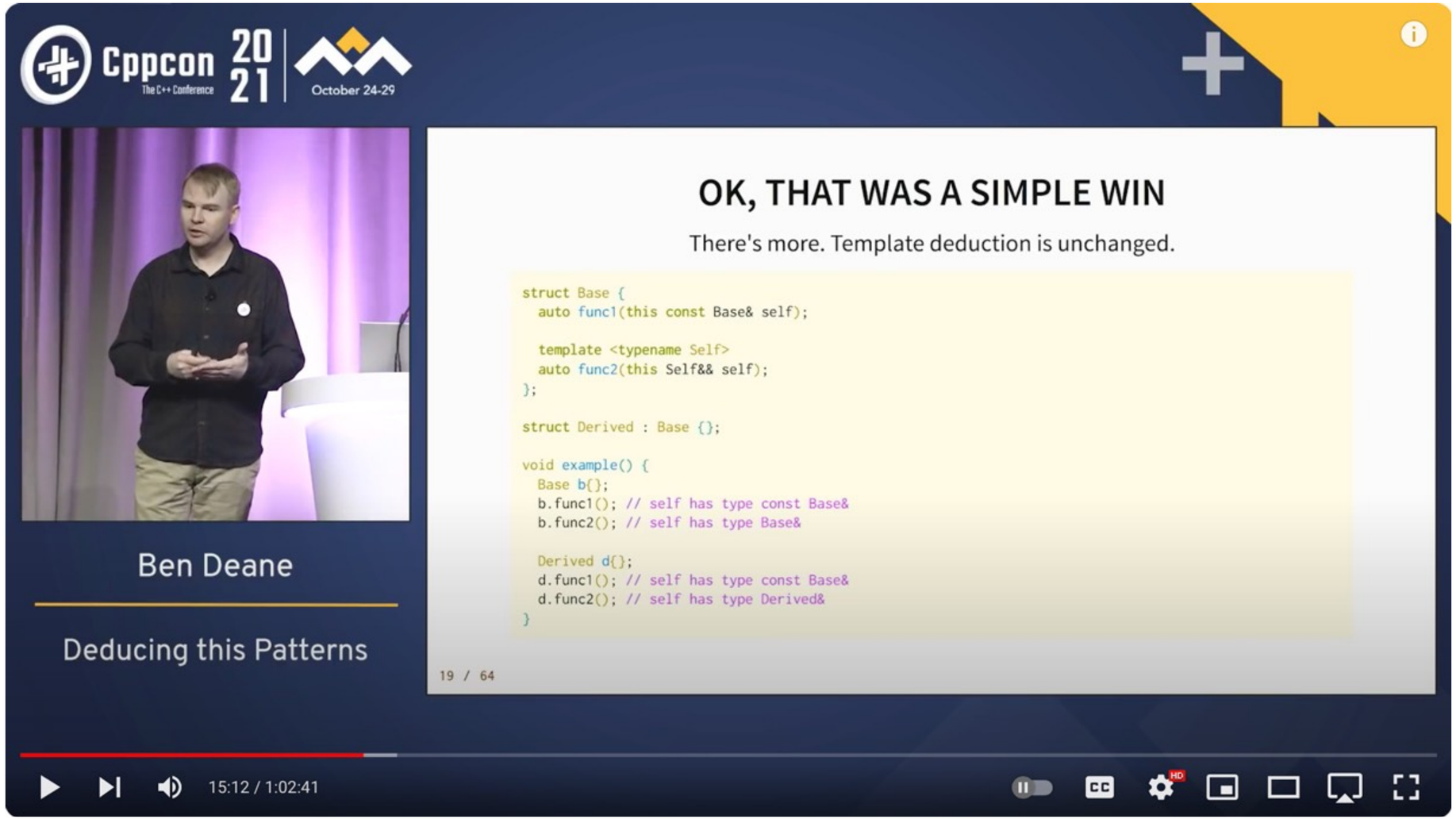
The Future of CRTP?



The Future of CRTP?

```
class Animal
{
 public:
   template< typename Self >
   void make_sound( this Self const& self )
   {
      self.make_sound_impl();
   }
};


class Sheep : public Animal
{
 public:
   void make_sound_impl() const { std::cout << "baa"; }
};


int main()
{
   Sheep sheep;
   Animal& animal = sheep;

   sheep.make_sound();
   animal.make_sound();    // Compilation error!
}
```

The Future of CRTP?

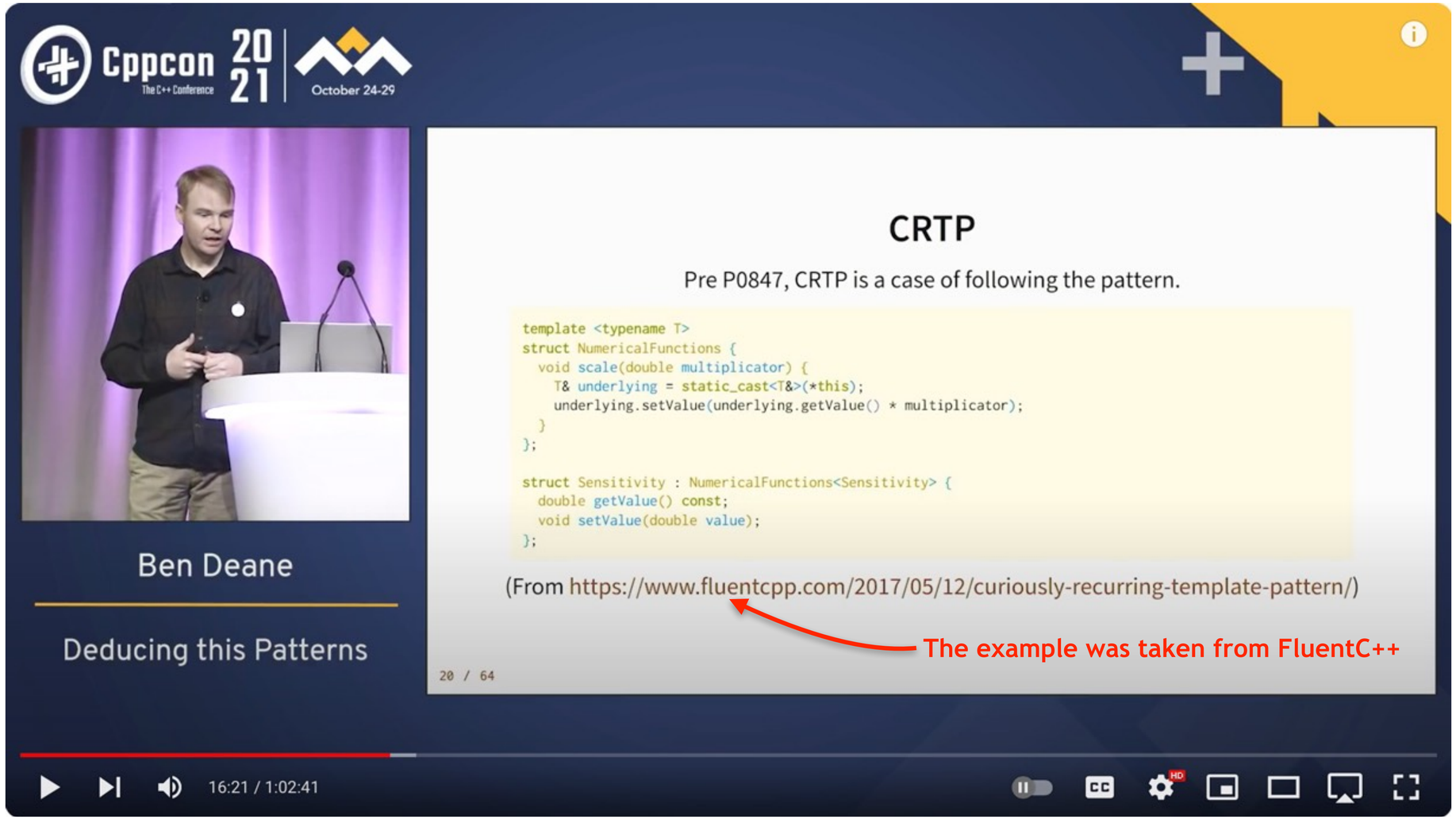


The Future of CRTP?

```
struct Base
{
   auto func1(this const Base& self);

   template <typename Self>
   auto func2(this Self&& self);
};

struct Derived : Base {};

void example()
{
   Base b{};
   b.func1();  // self has type const Base&
   b.func2();  // self has type Base&

   Derived d{};
   d.func1();  // self has type const Base&
   d.func2();  // self has type Derived&
}
```

When called via the base class, the explicit object parameter is always deduced to be the type of the base class (the static type).

The Future of CRTP?



Tweet

What the Curiously Recurring Template Pattern can bring to your code

Published May 16, 2017 - 16 Comments

After having defined the basics on the CRTP in episode #1 of the series, let's now consider how the CRTP can be helpful in day-to-day code.

The episodes in this series are:

- The CRTP, episode One: Definition
- The CRTP, episode Two: What the CRTP can bring to your code

RECENT POSTS

Usage First, Implementation After: A Principle of Software Development

Design Patterns VS Design Principles: Factory method

How to Store an lvalue or an rvalue in the Same Object

Copy-Paste Developments

Design Patterns VS Design Principles: Abstract Factory

How to Generate All the Combinations from Several Collections

After having defined the basics on the CRTP in episode #1 of the series, let's now consider how the CRTP can be helpful in day-to-day code.
The episodes in this series are:

- The CRTP, episode One: Definition
- The CRTP, episode Two: What the CRTP can bring to your code
- The CRTP, episode Three: An implementation helper for the CRTP

I don't know about you, but the first few times I figured how the CRTP worked I ended up forgetting soon after, and in the end could never remember what the CRTP exactly was. This happened because a lot of definitions of CRTP stop there, and don't show you *what value* the CRTP can bring to your code.

But there are several ways the CRTP can be useful. Here I am presenting the one that I see most in code, **Adding Functionality**, and another one that is interesting but that I don't encounter as often: creating **Static Interfaces**.

There are two forms of CRTP!

In order to make the code examples shorter, I have omitted the private-constructor-and-template-friend trick seen in episode One. But in practice you would find it useful to prevent the wrong class from being passed to the CRTP template.

Adding functionality

Adding Functionality with CRTP

```
template< typename Derived >
struct NumericalFunctions
{
   void scale( double multiplicator )
   {
      Derived& underlying = static_cast<Derived&>(*this);
      underlying.setValue( underlying.getValue() * multiplicator );
   }
};

struct Sensitivity : public NumericalFunctions<Sensitivity>
{
   double getValue() const { return value; }
   void setValue( double v ) { value = v; }
   double value;
};

int main()
{
   Sensitivity s{ 1.2 };
   s.scale( 2.0 );

   std::println( std::cout, "s.getValue() = {}", s.getValue() );
}
```

Adding Functionality with C++23

```cpp
struct NumericalFunctions
{
   void scale( this auto&& self, double multiplicator )
   {
      self.setValue( self.getValue() * multiplicator );
   }
};


struct Sensitivity : public NumericalFunctions
{
   double getValue() const { return value; }
   void setValue( double v ) { value = v; }
   double value;
};

int main()
{
   Sensitivity s{ 1.2 };
   s.scale( 2.0 );

   std::println( std::cout, "s.getValue() = {}", s.getValue() );
}
```

Static Interfaces with CRTP

```
template< typename Derived >
class Animal
{
 private:
   // ... Private default ctor and dtor

 public:
   void make_sound() const {
      static_cast<Derived const&>(*this).make_sound_impl();
   }
};

class Sheep : public Animal<Sheep>
{
 public:
   void make_sound_impl() const { std::cout << "baa"; }
};

int main()
{
   Sheep sheep;
   Animal<Sheep>& animal = sheep;

   sheep.make_sound();
   animal.make_sound();
}
```

Static Interfaces with C++23

```
class Animal
{
 public:
   template< typename Self >
   void make_sound( this Self const& self )
   {
      self.make_sound_impl();
   }
};


class Sheep : public Animal
{
 public:
   void make_sound_impl() const { std::cout << "baa"; }
};

int main()
{
   Sheep sheep;
   Animal& animal = sheep;

   sheep.make_sound();
   animal.make_sound();
}
```

Cannot compile since the ‘Self’ type cannot be deduced to be the dynamic type

Two Forms of CRTP …

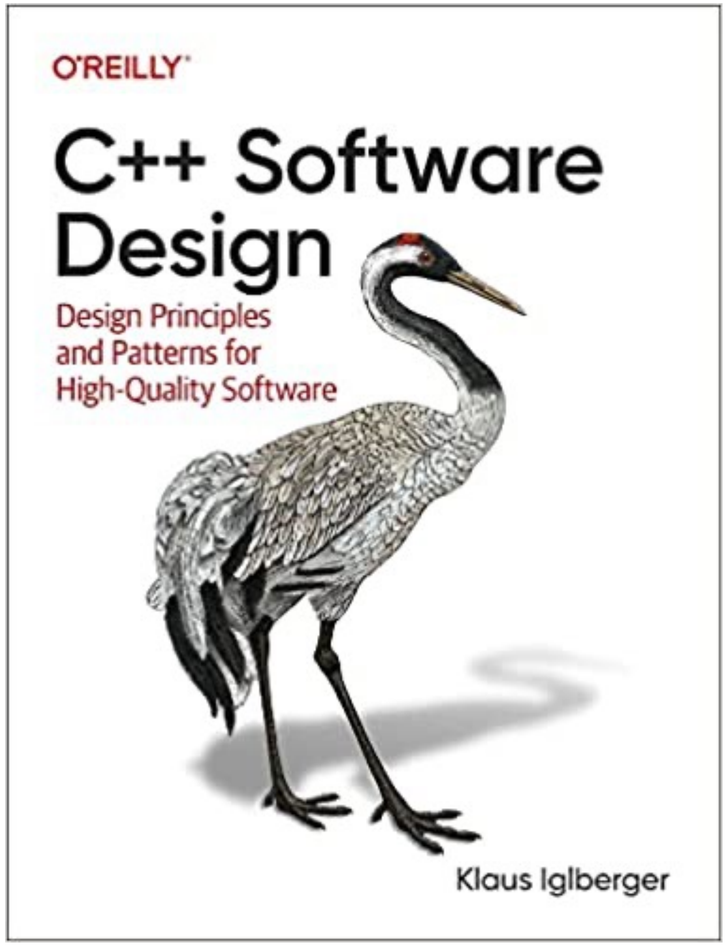
The term CRTP is ambiguous …

The ambiguity leads to misunderstandings …

We need new terms!

I apologize!

I failed to provide these two terms.



www.oreilly.com

The Two Forms of CRTP

CRTP for Static Interfaces …

- … provides a base class for a related set/family of types;
- … defines a common interface;
- … is used via the base class interface;
- … introduces an abstraction and is a design pattern.
- … should be called “**Static Interface**”

Adding Functionality via CRTP …

- … provides implementation details for the derived class;
- … does **not** define a common interface;
- … is not used via the base class interface;
- … does not introduce an abstraction, hence is no design pattern.
- … should be called “**Mixin**”

Guidelines

Guideline: Prefer to use the term "**Static Interface**" to express the intent to create a static family of types.

Guideline: "**Static Interface**" is a design pattern. Explicit object parameters (an implementation detail) cannot replace CRTP.

Guideline: Prefer to use the term "**Mixin**" to express the intent to inherit implementation details from a base class.

Guideline: "**Mixins**" are an implementation detail. Explicit object parameters are an alternative and can be a replacement for CRTP.

Wait a second! …

Couldn’t we just replace CRTP for static interfaces with a C++20 concept?

Static Interfaces with Concepts

```
template< typename T >
concept Animal =
   requires( T animal ) { animal.make_sound(); };


template< Animal T >
void print( T const& animal )
{
   animal.make_sound();
}

class Sheep
{
 public:
   void make_sound() const { std::cout << "baa"; }
};


int main()
{
   Sheep sheep;
   print( sheep );
}
```

"I can be printed, too!"

“Hooo…pahhh, me too!”

“It is logical to assume that I can be printed, too!”

This is not the same as Static Interfaces!
Anything can be passed, not just a specified set of types.
Static Interfaces are about an explicit opt-in.

Static Interfaces with Concepts

```
class AnimalTag {};

template< typename T >
concept Animal =
   requires( T animal ) { animal.make_sound(); } &&
   std::derived_from<T,AnimalTag>;

template< Animal T >
void print( T const& animal )
{
   animal.make_sound();
}

class Sheep : public AnimalTag
{
 public:
   void make_sound() const { std::cout << "baa"; }
};


int main()
{
   Sheep sheep;
   print( sheep );
}
```

"I can't be printed anymore 😕"

"Hooo...pahhh, me neither!"

"It is logical to assume that now I'm unprintable as well!"

This is a CRTP-free Static Interface: the Sheep class explicitly opts-in to be an animal.

This could not be achieved with a nested type/member. Only the base cannot be present by coincident.

Guidelines

Guideline: Prefer to use the term "**Static Interface**" to express the intent to create a static family of types.

Guideline: "**Static Interface**" is a design pattern. Explicit object parameters (an implementation detail) are not an alternative.

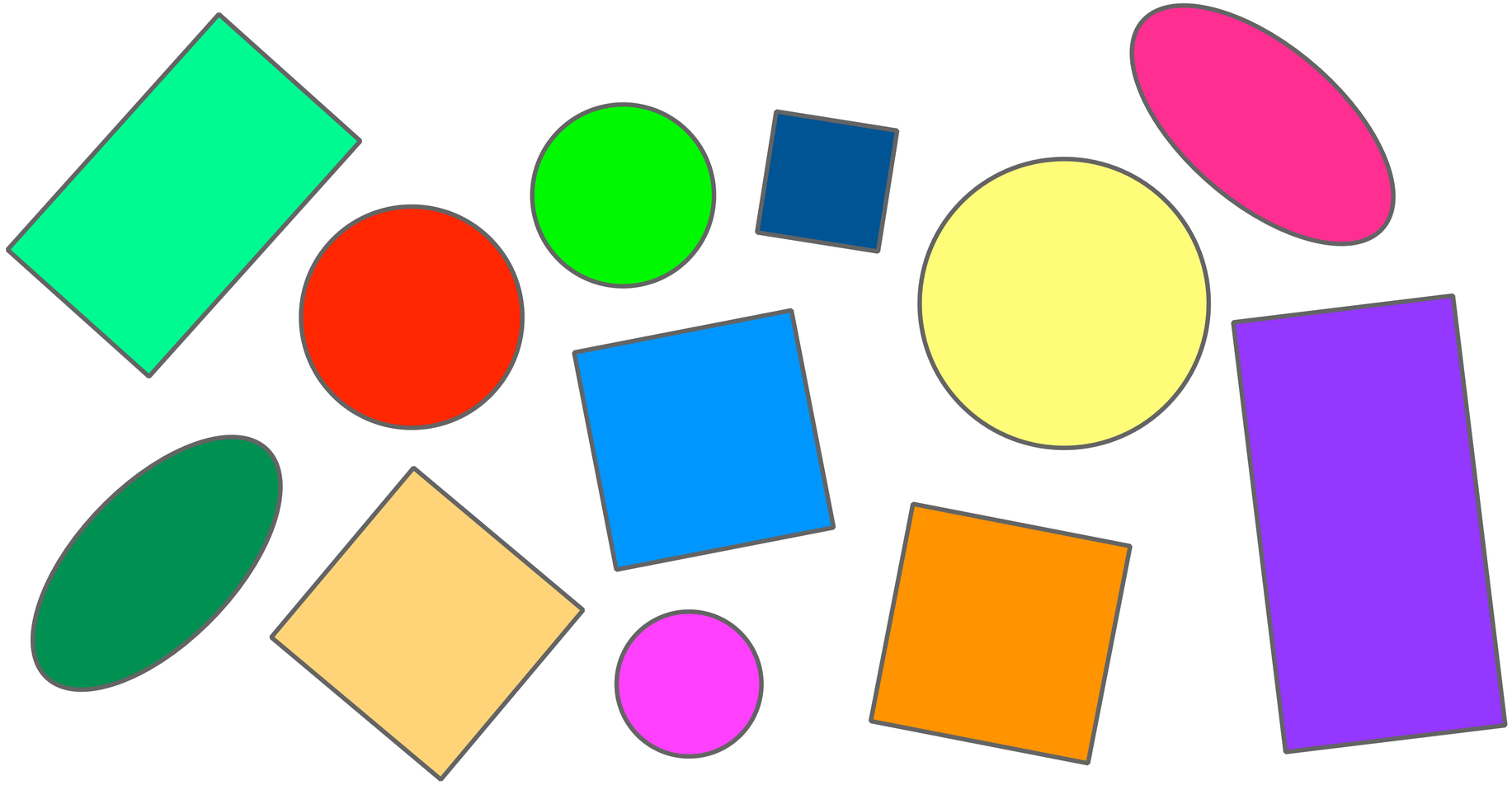
Guideline: Prefer to use the term "**Mixin**" to express the intent to inherit implementation details from a base class.

Guideline: "**Mixins**" are an implementation detail. Explicit object parameters are an alternative and can be a replacement for CRTP.

Issue #2

std::variant

Our Toy Problem: Drawing Shapes



Our Toy Problem: Drawing Shapes

Requirements:

- Extensible by new kinds of shapes
- 10M+ lines of code
- 100+ developers

A Classic Object-Oriented Solution

```
template< typename ConcreteShape >
class DrawStrategy
{
 public:
   virtual ~DrawStrategy() = default;

   virtual void draw( ConcreteShape const& shape ) const = 0;
};


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

   virtual void draw() const = 0;
   // ... several other virtual functions
};



class Circle : public Shape
{
 public:
   Circle( double rad, std::unique_ptr<DrawStrategy<Circle>>&& ds )
      : radius{ rad }
      , // ... Remaining data members
      , drawer{ std::move(ds) }
   {}

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

A Classic Object-Oriented Solution

```
template< typename ConcreteShape >
class DrawStrategy
{
 public:
   virtual ~DrawStrategy() = default;

   virtual void draw( ConcreteShape const& shape ) const = 0;
};


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

   virtual void draw() const = 0;
   // ... several other virtual functions
};


class Circle : public Shape
{
 public:
   Circle( double rad, std::unique_ptr<DrawStrategy<Circle>>&& ds )
      : radius{ rad }
      , // ... Remaining data members
      , drawer{ std::move(ds) }
   {}

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

A Classic Object-Oriented Solution

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

   virtual void draw() const = 0;
   // ... several other virtual functions
};


class Circle : public Shape
{
 public:
   Circle( double rad, std::unique_ptr<DrawStrategy<Circle>>&& ds )
      : radius{ rad }
      , // ... Remaining data members
      , drawer{ std::move(ds) }
   {}

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

   void draw() const override;
   // ... several other virtual functions

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

A Classic Object-Oriented Solution

```
template< typename ConcreteShape >
class DrawStrategy
{
 public:
   virtual ~DrawStrategy() = default;

   virtual void draw( ConcreteShape const& shape ) const = 0;
};


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

   virtual void draw() const = 0;
   // ... several other virtual functions
};


class Circle : public Shape
{
 public:
   Circle( double rad, std::unique_ptr<DrawStrategy<Circle>>&& ds )
      : radius{ rad }
      , // ... Remaining data members
      , drawer{ std::move(ds) }
   {}

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

A Classic Object-Oriented Solution

```
   virtual void draw() const = 0;
   // ... several other virtual functions
};


class Circle : public Shape
{
 public:
   Circle( double rad, std::unique_ptr<DrawStrategy<Circle>>&& ds )
      : radius{ rad }
      , // ... Remaining data members
      , drawer{ std::move(ds) }
   {}

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

   void draw() const override;
   // ... several other virtual functions

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


class Square : public Shape
{
 public:
   Square( double s, std::unique_ptr<DrawStrategy<Square>>&& ds )
      : side{ s }
```

A Classic Object-Oriented Solution

```
    double radius;
    // ... Remaining data members
    std::unique_ptr<DrawStrategy<Circle>> drawer;
};


class Square : public Shape
{
 public:
    Square( double s, std::unique_ptr<DrawStrategy<Square>>&& ds )
       : side{ s }
       , // ... Remaining data members
       , drawer{ std::move(ds) }
    {}

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

    void draw() const override;
    // ... several other virtual functions

 private:
    double side;
    // ... Remaining data members
    std::unique_ptr<DrawStrategy<Square>> drawer;
};


using Shapes = std::vector<std::unique_ptr<Shape>>;


class ShapesFactory
```

A Classic Object-Oriented Solution

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

    void draw() const override;
    // ... several other virtual functions

  private:
    double side;
    // ... Remaining data members
    std::unique_ptr<DrawStrategy<Square>> drawer;
};


using Shapes = std::vector<std::unique_ptr<Shape>>;


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

    virtual Shapes create( std::string_view filename ) const = 0;
};


void drawAllShapes( Shapes const& shapes )
{
    for( auto const& s : shapes )
    {
        s->draw();
    }
```

A Classic Object-Oriented Solution

```
 private:
   double side;
   // ... Remaining data members
   std::unique_ptr<DrawStrategy<Square>> drawer;
};


using Shapes = std::vector<std::unique_ptr<Shape>>;


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

   virtual Shapes create( std::string_view filename ) const = 0;
};



void drawAllShapes( Shapes const& shapes )
{
   for( auto const& s : shapes )
   {
      s->draw();
   }
}


void createAndDrawShapes( ShapesFactory const& factory, std::string_view filename )
{
```

A Classic Object-Oriented Solution

```
class ShapesFactory
{
 public:
   virtual ~ShapesFactory() = default;

   virtual Shapes create( std::string_view filename ) const = 0;
};



void drawAllShapes( Shapes const& shapes )
{
   for( auto const& s : shapes )
   {
      s->draw();
   }
}


void createAndDrawShapes( ShapesFactory const& factory, std::string_view filename )
{
   Shapes shapes = factory.create( filename );
   drawAllShapes( shapes );
}



class OpenGLDrawer : public DrawStrategy<Circle>
                   , public DrawStrategy<Square>
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}
```

A Classic Object-Oriented Solution

```
void drawAllShapes( Shapes const& shapes )
{
   for( auto const& s : shapes )
   {
      s->draw();
   }
}


void createAndDrawShapes( ShapesFactory const& factory, std::string_view filename )
{
   Shapes shapes = factory.create( filename );
   drawAllShapes( shapes );
}


class OpenGLDrawer : public DrawStrategy<Circle>
                   , public DrawStrategy<Square>
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

   void draw( Circle const& circle ) const override;

   void draw( Square const& square ) const override;

 private:
   // ... Data members (color, texture, transparency, ...)
};
```

A Classic Object-Oriented Solution

```
void createAndDrawShapes( ShapesFactory const& factory, std::string_view filename )
{
   Shapes shapes = factory.create( filename );
   drawAllShapes( shapes );
}
```

```
class OpenGLDrawer : public DrawStrategy<Circle>
                   , public DrawStrategy<Square>
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

   void draw( Circle const& circle ) const override;

   void draw( Square const& square ) const override;

 private:
   // ... Data members (color, texture, transparency, ...)
};
```

```
class YourShapesFactory : public ShapesFactory
{
 public:
   Shapes create( std::string_view filename ) const override
   {
      Shapes shapes{};
      std::string shape{};
      std::ifstream shape_file{ filename };
```

A Classic Object-Oriented Solution

```
class YourShapesFactory : public ShapesFactory
{
 public:
   Shapes create( std::string_view filename ) const override
   {
      Shapes shapes{};
      std::string shape{};
      std::ifstream shape_file{ filename };

      while( shape_file >> shape )
      {
         if( shape == "circle" ) {
            double radius;
            shape_file >> radius /* >> color, texture, transparency, ... */;
            shapes.emplace_back(
               std::make_unique<Circle>( radius
                                       , std::make_unique<OpenGLDrawer>(/*...*/) ) );
         }
         else if( shape == "square" ) {
            double side;
            shape_file >> side /* >> color, texture, transparency, ... */;
            shapes.emplace_back(
               std::make_unique<Square>( side
                                       , std::make_unique<OpenGLDrawer>(/*...*/) ) );
         }
         else {
            break;
         }
      }

      return shapes;
   }
};
```

A Classic Object-Oriented Solution

```
            else if( shape == "square" ) {
               double side;
               shape_file >> side /* >> color, texture, transparency, ... */;
               shapes.emplace_back(
                  std::make_unique<Square>( side
                                          , std::make_unique<OpenGLDrawer>(/*...*/) ) );
            }
            else {
               break;
            }
         }

         return shapes;
      }
};


int main()
{
   YourShapesFactory factory{};

   createAndDrawShapes( factory, “shapes.txt” );
}
```

A Classic Object-Oriented Solution

```
void drawAllShapes( Shapes const& shapes )
{
   for( auto const& s : shapes )
   {
      s->draw();
   }
}


void createAndDrawShapes( ShapesFactory const& factory, std::string_view filename )
{
   Shapes shapes = factory.create( filename );
   drawAllShapes( shapes );
}


class OpenGLDrawer : public DrawStrategy<Circle>
                   , public DrawStrategy<Square>
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

   void draw( Circle const& circle ) const override;

   void draw( Square const& square ) const override;

 private:
   // ... Data members (color, texture, transparency, ...)
};
```

A Classic Object-Oriented Solution

```
void drawAllShapes( Shapes const& shapes )
{
   for( auto const& s : shapes )
   {
      s->draw();
   }
}


void createAndDrawShapes( ShapesFactory const& factory, std::string_view filename )
{
   Shapes shapes = factory.create( filename );
   drawAllShapes( shapes );
}
```

High level (stable, low dependencies)

Architectural Boundary

Low level (volatile, malleable, high dependencies)

```
class OpenGLDrawer : public DrawStrategy<Circle>
                   , public DrawStrategy<Square>
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

   void draw( Circle const& circle ) const override;

   void draw( Square const& square ) const override;

 private:
```

A Classic Object-Oriented Solution

```
void drawAllShapes( Shapes const& shapes )
{
   for( auto const& s : shapes )
   {
      s->draw();
   }
}


void createAndDrawShapes( ShapesFactory const& factory, std::string_view filename )
{
   Shapes shapes = factory.create( filename );
   drawAllShapes( shapes );
}
```

My Code

Architectural Boundary

Your Code

```
class OpenGLDrawer : public DrawStrategy<Circle>
                   , public DrawStrategy<Square>
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

   void draw( Circle const& circle ) const override;

   void draw( Square const& square ) const override;

 private:
```

A Classic Object-Oriented Solution

My Code

Architectural Boundary

Your Code

```
class Rectangle : public Shape
{
 public:
   Rectangle( double width, double height
            , std::unique_ptr<DrawStrategy<Rectangle>>&& drawer )
      : width_{ width }
      , height_{ height }
      , // ... Remaining data members
      , drawer_{ std::move(drawer) }
   {}

   double width() const { return width_; }
   double height() const { return height_; }
   // ... getCenter(), getRotation(), ...

   void draw() const override { drawer_->draw(*this); }
   // ... several other virtual functions

 private:
   double width_;
   double height_;
   // ... Remaining data members
   std::unique_ptr<DrawStrategy<Rectangle>> drawer_;
};
```

A Classic Object-Oriented Solution

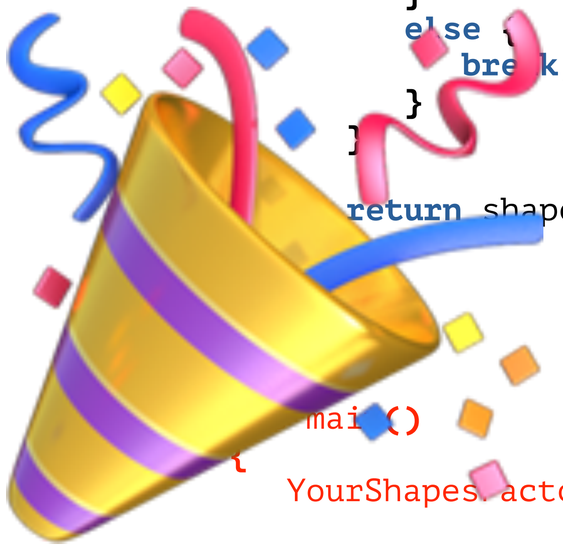
```
 private:
   double width_;
   double height_;
   // ... Remaining data members
   std::unique_ptr<DrawStrategy<Rectangle>> drawer_;
};


class OpenGLDrawer : public DrawStrategy<Circle>
                   , public DrawStrategy<Square>
                   , public DrawStrategy<Rectangle>
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

   void draw( Circle const& circle ) const override;

   void draw( Square const& square ) const override;

   void draw( Rectangle const& rectangle ) const override;

 private:
   // ... Data members (color, texture, transparency, ...)
};


class YourShapesFactory : public ShapesFactory
{
 public:
   Shapes create( std::string_view filename ) const override
   {
      Shapes shapes{};
```

A Classic Object-Oriented Solution

```
class YourShapesFactory : public ShapesFactory
{
 public:
   Shapes create( std::string_view filename ) const override
   {
      Shapes shapes{};
      std::string shape{};
      std::ifstream shape_file{ filename };

      while( shape_file >> shape )
      {
         if( shape == "circle" ) {
            // ... Creating a circle
         }
         else if( shape == "square" ) {
            // ... Creating a square
         }
         else if( shape == "rectangle" ) {
            double width;
            double height;
            shape_file >> width >> height /* >> color, texture, transparency, ... */;
            shapes.emplace_back(
               std::make_unique<Rectangle>( width, height
                                          , std::make_unique<OpenGLDrawer>(/*...*/) ) );
         }
         else {
            break;
         }
      }

      return shapes;
   }
};
```

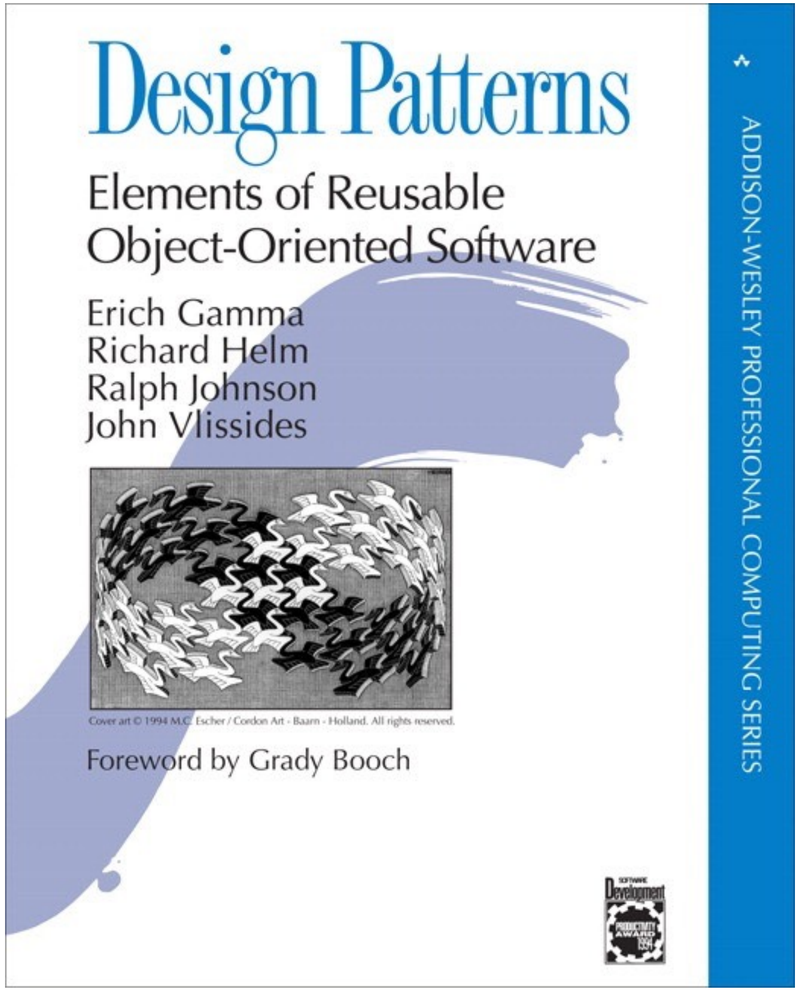
A Classic Object-Oriented Solution

```
        else if( shape == "rectangle" ) {
            double width;
            double height;
            shape_file >> width >> height /* >> color, texture, transparency, ... */;
            shapes.emplace_back(
                std::make_unique<Rectangle>( width, height
                                           , std::make_unique<OpenGLDrawer>(/*...*/) ) );
        }
        else {
            break;
        }
    }

    return shapes;
}

main()
{
    YourShapesFactory factory{};

    createAndDrawShapes( factory, "shapes.txt" );
}
```

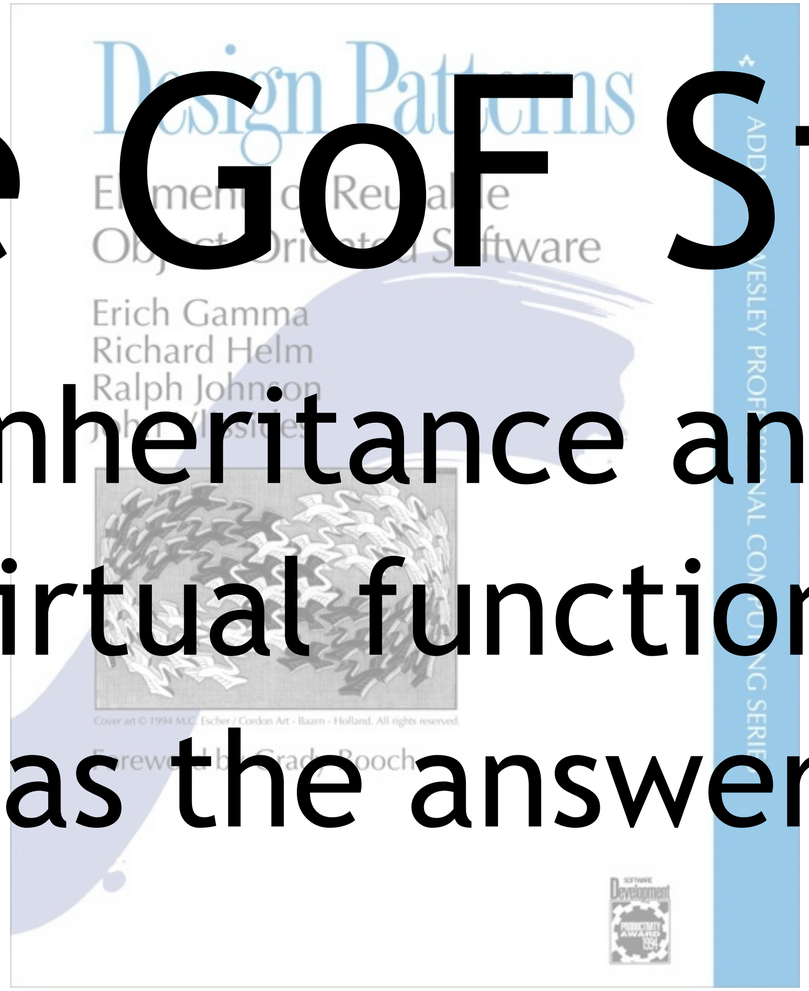


Yes… some of you are unhappy about this style of programming.

The Philosophy of the 90s

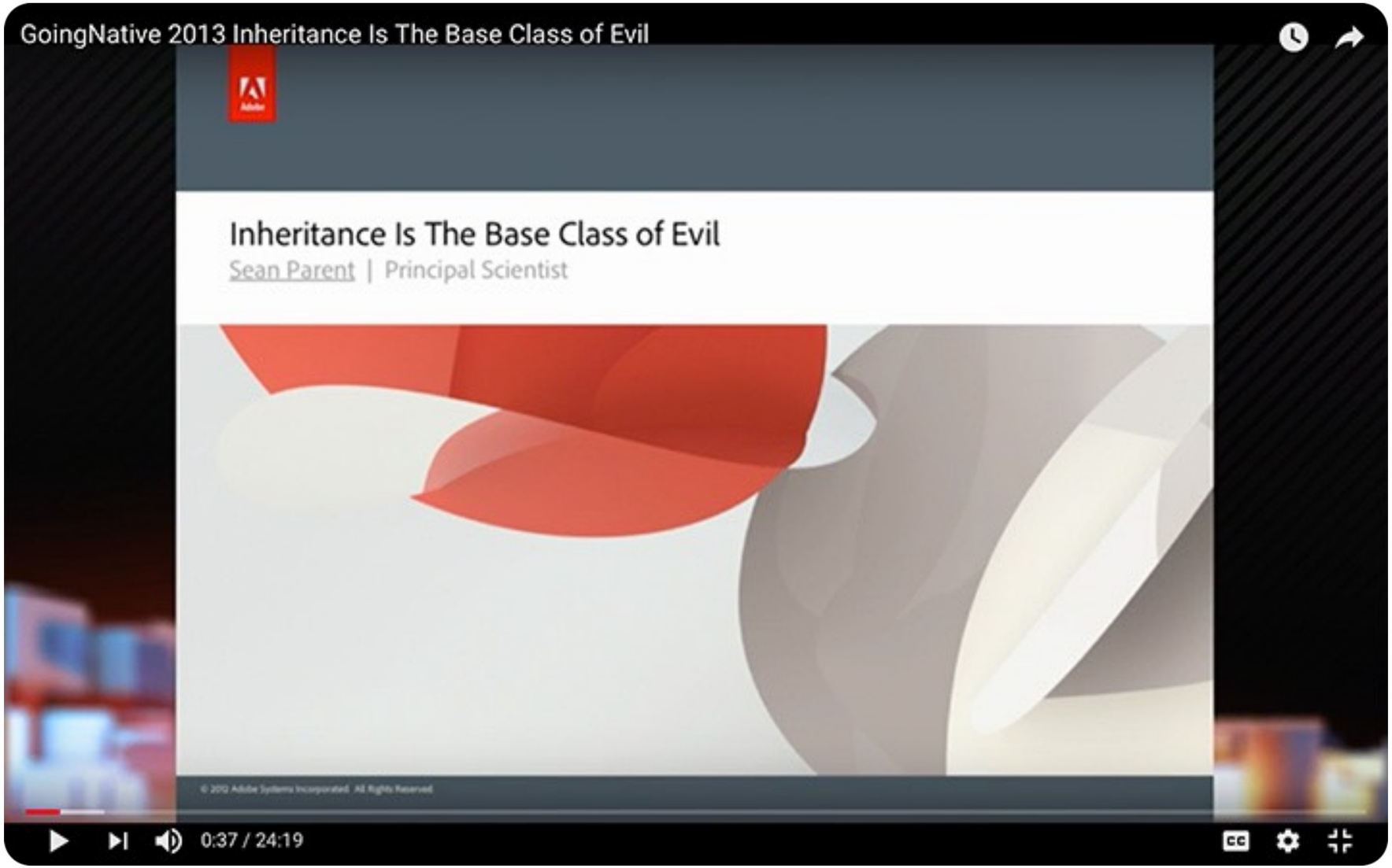


The GoF Style



Inheritance and virtual functions as the answer

The Fallen Paradigm (?)



The Fallen Paradigm (?)

The Fallen Paradigm (?)

”I believe that object-oriented programming and especially its theory is overestimated. … C++ always had templates, and now also has std::variant, which makes most of the use of inheritance unnecessary.”

(Unknown Reviewer)

The Fallen Paradigm (?)

”I believe that object-oriented programming and especially its theory is overestimated. … C++ always had templates, and now also has std::variant, which makes most of the use of inheritance unnecessary.”

(Unknown Reviewer)

A Truly Modern C++ Solution: std::variant (?)

```
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 Truly Modern C++ Solution: std::variant (?)

```
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 Truly Modern C++ Solution: std::variant (?)

```
    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
};


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


using Shapes = std::vector<Shape>;
```

Circle and Square are soooo much simpler!

- no inheritance
- no dependency on graphics code
- no (base) pointers
- no manual life-time management
- less code to write

A Truly Modern C++ Solution: std::variant (?)

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

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

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


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


using Shapes = std::vector<Shape>;


class ShapesFactory
{
 public:
   Shapes create( std::string_view filename )
   {
      Shapes shapes{};
      std::string shape{};

      std::ifstream shape_file{ filename };

      while( shape_file >> shape )
      {
```

std::variant replaces the Shape base class

A Truly Modern C++ Solution: std::variant (?)

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

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

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


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


using Shapes = std::vector<Shape>;


class ShapesFactory
{
 public:
   Shapes create( std::string_view filename )
   {
      Shapes shapes{};
      std::string shape{};

      std::ifstream shape_file{ filename };

      while( shape_file >> shape )
      {
```

We now utilize a vector of values instead of pointers

A Truly Modern C++ Solution: std::variant (?)

```
class ShapesFactory
{
 public:
   Shapes create( std::string_view filename )
   {
      Shapes shapes{};
      std::string shape{};

      std::ifstream shape_file{ filename };

      while( shape_file >> shape )
      {
         if( shape == "circle" ) {
            double radius;
            shape_file >> radius;
            shapes.emplace_back( Circle{radius} );
         }
         else if( shape == "square" ) {
            double side;
            shape_file >> side;
            shapes.emplace_back( Square{side} );
         }
         else {
            break;
         }
      }

      return shapes;
   }
};
```

No inheritance necessary!

No need to allocate dynamic memory!

A Truly Modern C++ Solution: std::variant (?)

```
            shape_file >> side;
            shapes.emplace_back( Square{side} );
         }
         else {
            break;
         }
      }

      return shapes;
   }
};
```

Replacing another inheritance hierarchy with std::variant

```
using Factory = std::variant<ShapesFactory>;


class OpenGLDrawer
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

   void operator()( Circle const& circle ) const;

   void operator()( Square const& square ) const;

 private:
   // ... Data members (color, texture, transparency, ...)
};


using Drawer = std::variant<OpenGLDrawer>;
```

A Truly Modern C++ Solution: std::variant (?)

```
using Factory = std::variant<ShapesFactory>;


class OpenGLDrawer
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

   void operator()( Circle const& circle ) const;

   void operator()( Square const& square ) const;

 private:
   // ... Data members (color, texture, transparency, ...)
};


using Drawer = std::variant<OpenGLDrawer>;


void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( []( auto d, auto s ){ d(s); }, drawer, shape );
   }
}


void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
```

Again, no inheritance necessary!

A Truly Modern C++ Solution: std::variant (?)

```
using Factory = std::variant<ShapesFactory>;


class OpenGLDrawer
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

   void operator()( Circle const& circle ) const;

   void operator()( Square const& square ) const;

 private:
   // ... Data members (color, texture, transparency, ...)
};


using Drawer = std::variant<OpenGLDrawer>;


void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( []( auto d, auto s ){ d(s); }, drawer, shape );
   }
}


void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
```

And another inheritance hierarchy gone!

A Truly Modern C++ Solution: std::variant (?)

```
using Drawer = std::variant<OpenGLDrawer>;


void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( []( auto d, auto s ){ d(s); }, drawer, shape );
   }
}


void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   Shapes shapes = std::visit( [&filename]( auto f ){ return f.create( filename ); }, factory );
   drawAllShapes( shapes, drawer );
}


int main()
{
   ShapesFactory factory{};
   OpenGLDrawer drawer{/*...*/};

   createAndDrawShapes( factory, "shapes.txt", drawer );
}
```

A runtime dispatch on two variants!

A Truly Modern C++ Solution: std::variant (?)

```
using Drawer = std::variant<OpenGLDrawer>;


void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( []( auto d, auto s ){ d(s); }, drawer, shape );
   }
}


void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   Shapes shapes = std::visit( [&filename]( auto f ){ return f.create( filename ); }, factory );
   drawAllShapes( shapes, drawer );
}


int main()
{
   ShapesFactory factory{};
   OpenGLDrawer drawer{/*...*/};

   createAndDrawShapes( factory, "shapes.txt", drawer );
}
```

A Truly Modern C++ Solution: std::variant (?)

```
using Drawer = std::variant<OpenGLDrawer>;


void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( []( auto d, auto s ){ d(s); }, drawer, shape );
   }
}


void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   Shapes shapes = std::visit( [&filename]( auto f ){ return f.create( filename ); }, factory );
   drawAllShapes( shapes, drawer );
}


int main()
{
   ShapesFactory factory{};
   OpenGLDrawer drawer{/*...*/};

   createAndDrawShapes( factory, "shapes.txt", drawer );
}
```

A Truly Modern C++ Solution: std::variant (?)

This solution is soooo much better:

- No inheritance, but a functional approach
- No (smart) pointers, but values
- Proper management of graphics code
- Automatic, elegant life-time management
- Less code to write
- Soooo much simpler
- Better performance

Performance Comparison

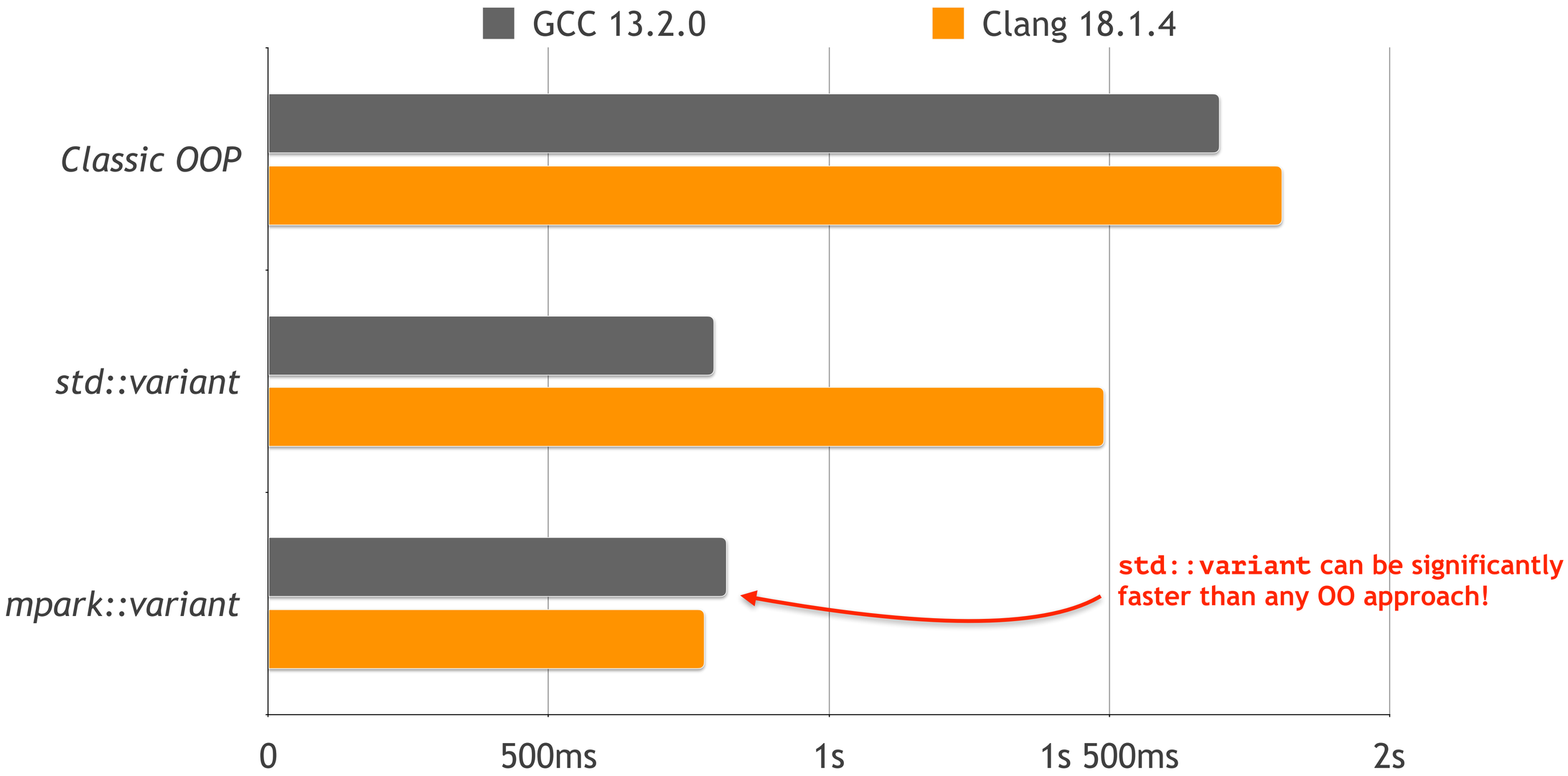
Performance … *sigh*

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

Performance Comparison

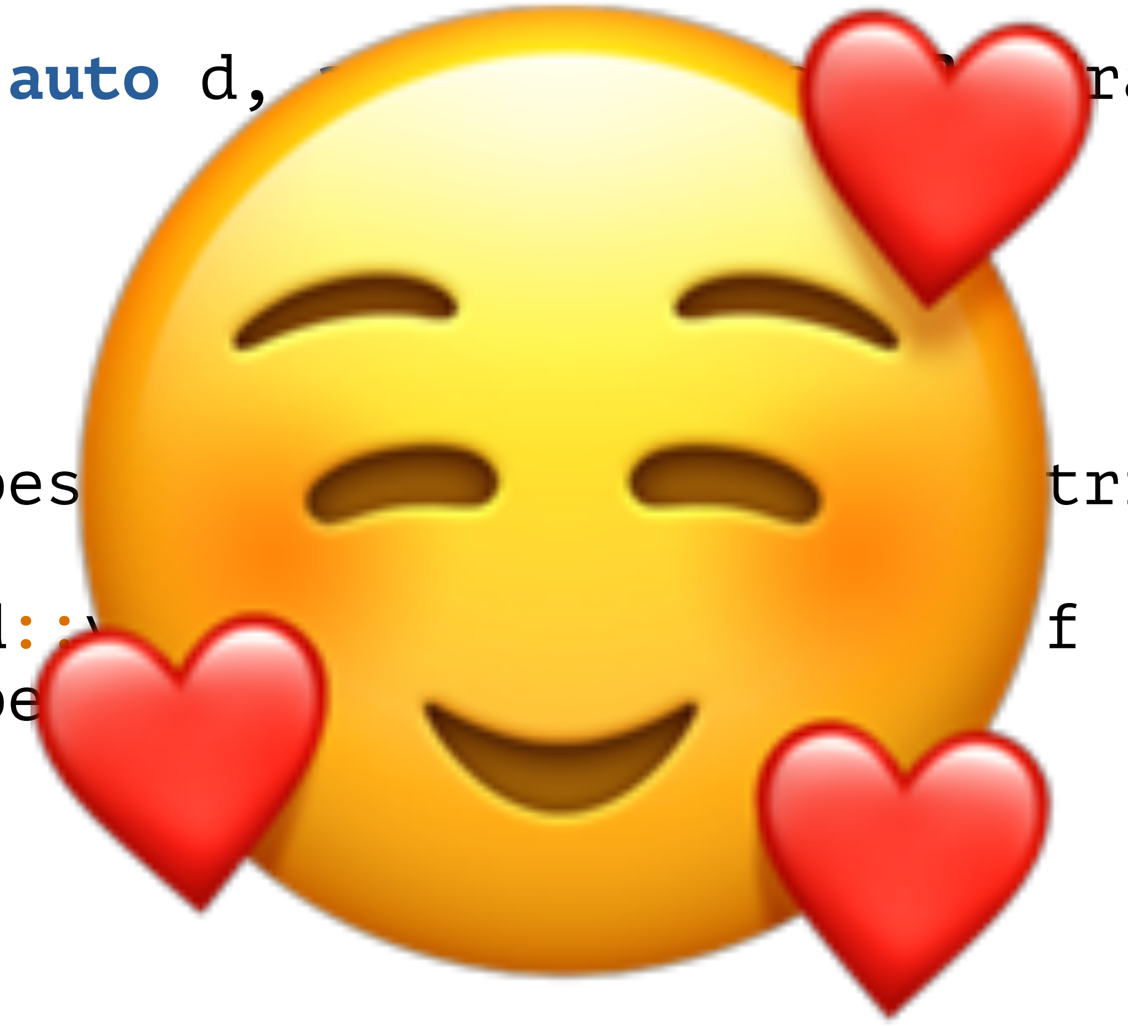
- 6 different shapes: circles, squares, ellipses, rectangles, hexagons and pentagons
- Using 10000 randomly generated shapes
- Performing 25000 `translate()` operations each
- Benchmarks with GCC-13.2.0 and Clang-18.1.4
- 8-core Intel Core i7 with 3.8 Ghz, 64 GB of main memory

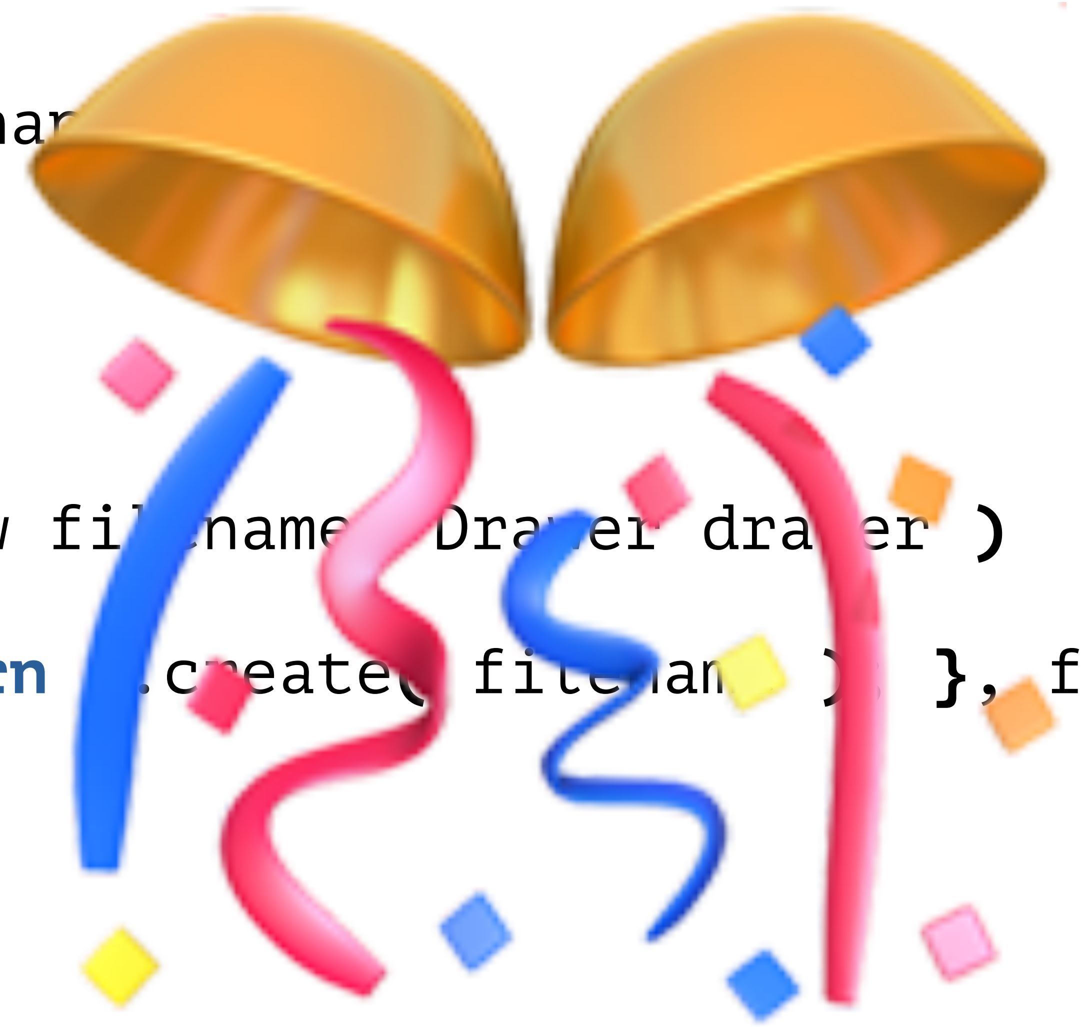
Performance Comparison



A Truly Modern C++ Solution: std::variant (?)

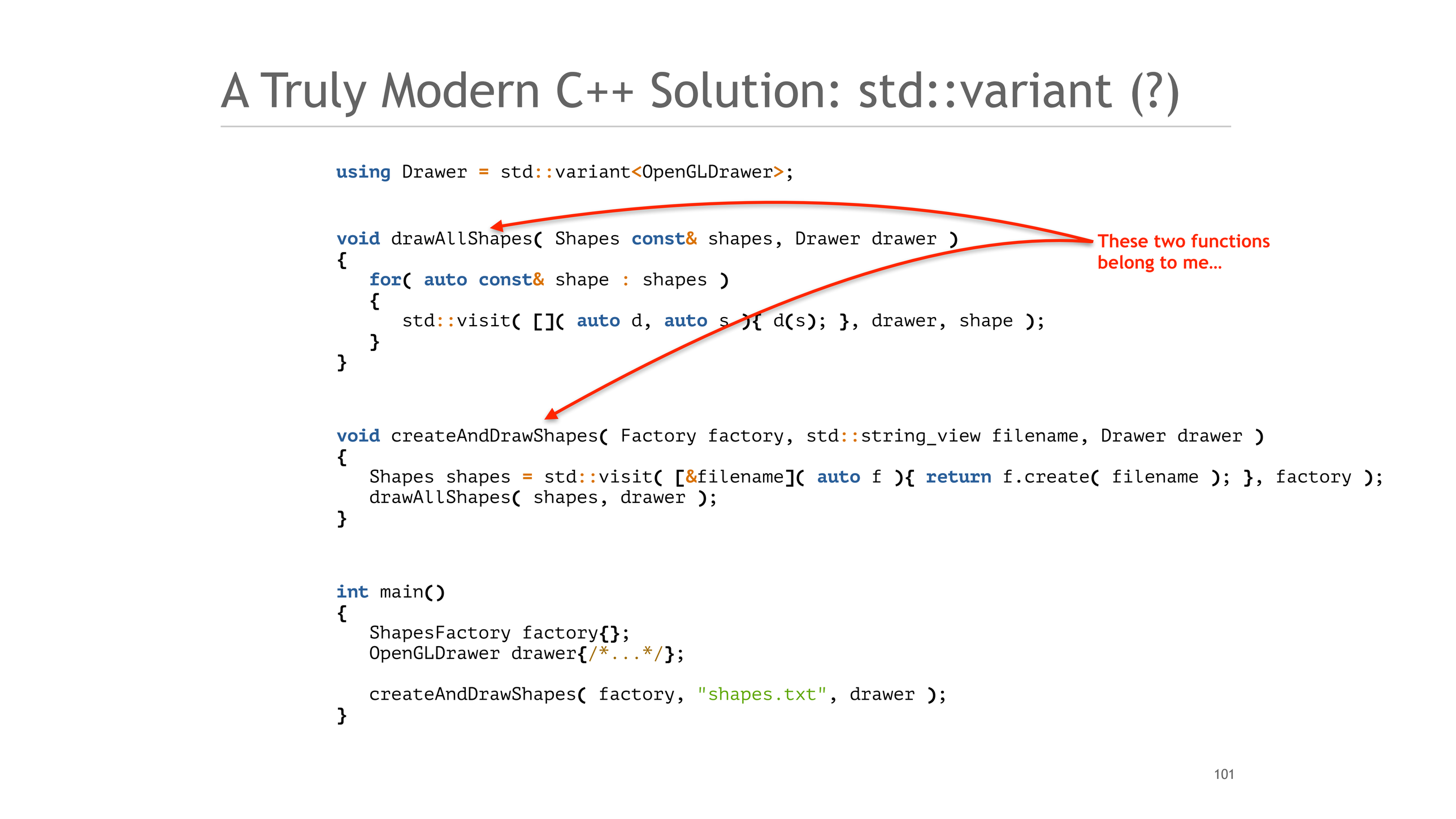
```
using Drawer = std::variant<OpenGLDrawer>;

void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::vi it( []( auto d,                rawer, sha
   }
}

   eateAndDrawShapes                 tring_view fi name  Dra er dra er )

     shapes = std::              f ){ return  create( fi  an )  }, factory );
   awAllShapes( shape

int main()
{
   ShapesFactory factory{};
   OpenGLDrawer drawer{/*...*/};

   createAndDrawShapes( factory, "shapes.txt", drawer );
}
```





A Truly Modern C++ Solution: std::variant (?)

```
using Drawer = std::variant<OpenGLDrawer>;


void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( []( auto d, auto s ){ d(s); }, drawer, shape );
   }
}



void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   Shapes shapes = std::visit( [&filename]( auto f ){ return f.create( filename ); }, factory );
   drawAllShapes( shapes, drawer );
}


int main()
{
   ShapesFactory factory{};
   OpenGLDrawer drawer{/*...*/};

   createAndDrawShapes( factory, "shapes.txt", drawer );
}
```



A Truly Modern C++ Solution: std::variant (?)

```
using Drawer = std::variant<OpenGLDrawer>;


void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( []( auto d, auto s ){ d(s); }, drawer, shape );
   }
}


void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   Shapes shapes = std::visit( [&filename]( auto f ){ return f.create( filename ); }, factory );
   drawAllShapes( shapes, drawer );
}
```

My Code

Architectural Boundary

Your Code

```
int main()
{
   ShapesFactory factory{};
   OpenGLDrawer drawer{/*...*/};

   createAndDrawShapes( factory, "shapes.txt", drawer );
}
```

A Truly Modern C++ Solution: std::variant (?)

```
        return shapes;
    }
};


using Factory = std::variant<ShapesFactory>;


class OpenGLDrawer
{
 public:
    explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

    void operator()( Circle const& circle ) const;

    void operator()( Square const& square ) const;

 private:
    // ... Data members (color, texture, transparency, ...)
};


using Drawer = std::variant<OpenGLDrawer>;


void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
    for( auto const& shape : shapes )
    {
        std::visit( []( auto d, auto s ){ d(s); }, drawer, shape );
    }
}
```

However, this is an implementation detail, so this is your code …

A Truly Modern C++ Solution: std::variant (?)

```
void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   Shapes shapes = std::visit( [&filename]( auto f ){ return f.create( filename ); }, factory );
   drawAllShapes( shapes, drawer );
}
```

Oh, but how can I use the Drawer when it is in your code…

My Code

Architectural Boundary

Your Code

```
class OpenGLDrawer
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

   void operator()( Circle const& circle ) const;

   void operator()( Square const& square ) const;

 private:
   // ... Data members (color, texture, transparency, ...)
};
```

```
using Drawer = std::variant<OpenGLDrawer>;
```

```
int main()
```

A Truly Modern C++ Solution: std::variant (?)

```
using Factory = std::variant<ShapesFactory>;


using Drawer = std::variant<OpenGLDrawer>;


void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( []( auto d, auto s ){ d(s); }, drawer, shape );
   }
}


void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   Shapes shapes = std::visit( [&filename]( auto f ){ return f.create( filename ); }, factory );
   drawAllShapes( shapes, drawer );
}
```

But no! Now I have to know about the OpenGLDrawer again!

My Code

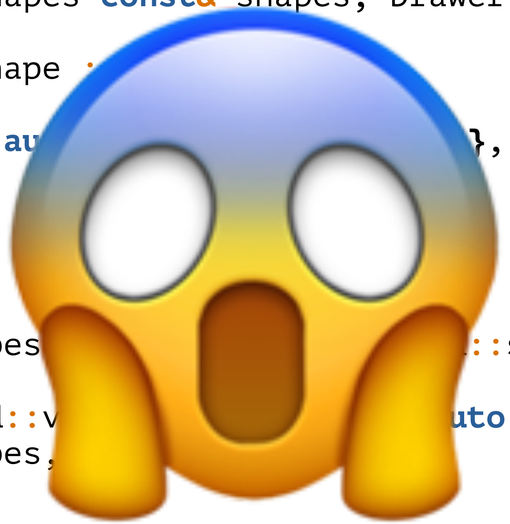
Architectural Boundary

Your Code

```
class OpenGLDrawer
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}
```

A Truly Modern C++ Solution: std::variant (?)

```
using Factory = std::variant<ShapesFactory>;


using Drawer = std::variant<OpenGLDrawer>;


void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape :
   {
      std::visit( [](au                                  }, drawer, shape );
   }
}


void createAndDrawShapes                                  ::string_view filename, Drawer drawer )
{
   Shapes shapes = std::v                                 uto f ){ return f.create( filename ); }, factory );
   drawAllShapes( shapes,
}
```



My Code

Architectural Boundary

Your Code

```
class OpenGLDrawer
{
 public:
   explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}
```

This is an architectural disaster, a total failure!

There is no architecture!

Templates to the Rescue (?)

”I believe that object-oriented programming and especially its theory is overestimated. … C++ always had templates, and now also has std::variant, which makes most of the use of inheritance unnecessary.”

(Unknown Reviewer)

Templates to the Rescue (?)

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


template< typename Shapes, typename Drawer >
void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( drawer, shape );
   }
}



template< typename Factory, typename Drawer >
void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   auto shapes = factory.create( filename );
   drawAllShapes( shapes, drawer );
}
```

My Code

Architectural Boundary

Your Code

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

Templates to the Rescue (?)

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

Let's make this a function template.
This way we invert the dependencies …

```
template< typename Shapes, typename Drawer >
void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( drawer, shape );
   }
}
```

```
template< typename Factory, typename Drawer >
void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   auto shapes = factory.create( filename );
   drawAllShapes( shapes, drawer );
}
```

My Code

Architectural Boundary

Your Code

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

Templates to the Rescue (?)

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


template< typename Shapes, typename Drawer >
void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( drawer, shape );
   }
}
```

Let's make this a function template, too.
Again, this inverts the dependencies …

```
template< typename Factory, typename Drawer >
void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   auto shapes = factory.create( filename );
   drawAllShapes( shapes, drawer );
}
```

My Code

Architectural Boundary

Your Code

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

Templates to the Rescue (?)

My Code

Architectural Boundary

Your Code

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


using Shapes = std::vector<Shape>;



class ShapesFactory
{
 public:
   Shapes create( std::string_view filename )
   {
      Shapes shapes{};
      std::string shape{};

      std::ifstream shape_file{ filename };

      while( shape_file >> shape )
      {
         if( shape == "circle" ) {
            double radius;
            shape_file >> radius;
            shapes.emplace_back( Circle{radius} );
         }
         else if( shape == "square" ) {
            double side;
            shape_file >> side;
```

Templates to the Rescue (?)

My Code

Architectural Boundary

Your Code

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


using Shapes = std::vector<Shape>;


class ShapesFactory
{
 public:
   Shapes create( std::string_view filename )
   {
      Shapes shapes{};
      std::string shape{};

      std::ifstream shape_file{ filename };

      while( shape_file >> shape )
      {
         if( shape == "circle" ) {
            double radius;
            shape_file >> radius;
            shapes.emplace_back( Circle{radius} );
         }
         else if( shape == "square" ) {
            double side;
            shape_file >> side;
```

Templates to the Rescue (?)

My Code

Architectural Boundary

Your Code

```
class Rectangle
{
 public:
   Rectangle( double width, double height )
      : width_{ width }
      , height_{ height }
      , // ... Remaining data members
   {}

   double width() const { return width_; }
   double height() const { return height_; }
   // ... getCenter(), getRotation(), ...

 private:
   double width_;
   double height_;
   // ... Remaining data members
};
```

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

```
using Shapes = std::vector<Shape>;
```

Templates to the Rescue (?)

My Code

Architectural Boundary

Your Code

```
class Rectangle
{
 public:
   Rectangle( double width, double height )
      : width_{ width }
      , height_{ height }
      , // ... Remaining data members
   {}

   double width() const { return width_; }
   double height() const { return height_; }
   // ... getCenter(), getRotation(), ...

 private:
   double width_;
   double height_;
   // ... Remaining data members
};
```

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

```
using Shapes = std::vector<Shape>;
```

Templates to the Rescue (?)

```
class ShapesFactory
{
 public:
   Shapes create( std::string_view filename )
   {
      Shapes shapes{};
      std::string shape{};

      std::ifstream shape_file{ filename };

      while( shape_file >> shape )
      {
         if( shape == "circle" ) {
            // ...
         }
         else if( shape == "square" ) {
            // ...
         }
         else if( shape == "rectangle" )
         {
            double width, height;
            shape_file >> width >> height;
            shapes.emplace_back( Rectangle{width,height} );
         }
         else {
            break;
         }
      }

      return shapes;
   }
};
```

Templates to the Rescue (?)

```
        return shapes;
    }
};


class OpenGLDrawer
{
 public:
    explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}

    void operator()( Circle const& circle ) const;

    void operator()( Square const& square ) const;

    void operator()( Rectangle const& rectangle ) const;

 private:
    // ... Data members (color, texture, transparency, ...)
};


int main()
{
    ShapesFactory factory{};
    OpenGLDrawer drawer{/*...*/};

    createAndDrawShapes( factory, "shapes.txt", drawer );
}
```

Templates to the Rescue (?)

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

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

The template approach could work…
in a small code base.
But in 10M+ lines of code?

```
template< typename Shapes, typename Drawer >
void drawAllShapes( Shapes const& shapes, Drawer drawer )
{
   for( auto const& shape : shapes )
   {
      std::visit( drawer, shape );
   }
}
```

```
template< typename Factory, typename Drawer >
void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   auto shapes = factory.create( filename );
   drawAllShapes( shapes, drawer );
}
```

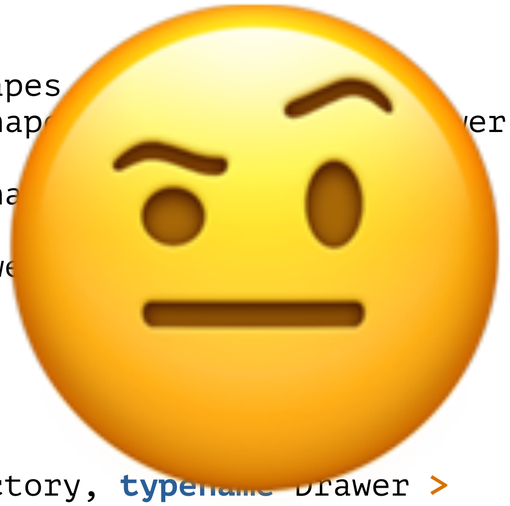
My Code

Your Code

Architectural Boundary

Templates to the Rescue (?)

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

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


template< typename Shapes
void drawAllShapes( Shap                    er drawer )
{
   for( auto const& sha
   {
      std::visit( drawe
   }
}


template< typename Factory, typename Drawer >
void createAndDrawShapes( Factory factory, std::string_view filename, Drawer drawer )
{
   auto shapes = factory.create( filename );
   drawAllShapes( shapes, drawer );
}
```



My Code

Your Code

Architectural Boundary

std::variant is not a replacement for virtual functions!

std::variant is the architectural antipode of virtual functions!

std::variant vs. Virtual Functions

std::variant	*Virtual Functions*
Dynamic polymorphism	Dynamic polymorphism
Functional programming	Object-oriented programming
Fixed set of types	Open set of types
Open set of operations	Closed set of operations

std::variant vs. Virtual Functions

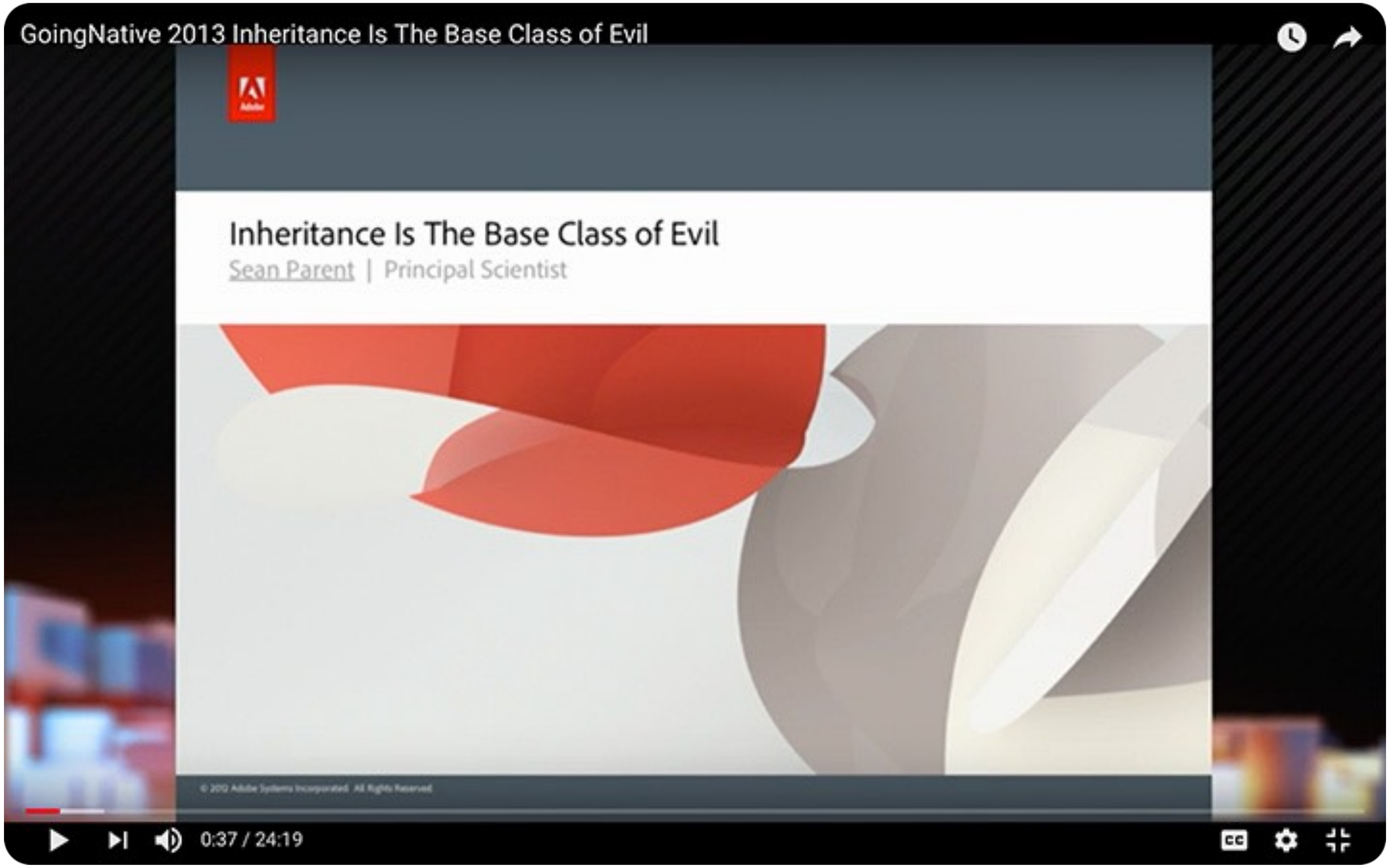
std::variant (aka Visitor pattern)	*Virtual Functions*
Dynamic polymorphism	Dynamic polymorphism
Functional programming	Object-oriented programming
Fixed set of types	Open set of types
Open set of operations	Closed set of operations

Guidelines

Guideline: std::variant is not a replacement for virtual functions and shouldn’t be used as such.

Wait a second! …

But Isn’t Inheritance Evil?



Issue #1

CRTP

Issue #2

std::variant

Neither CRTP nor std::variant are replacements for virtual functions

Design patterns represent dependency structures

Thus design patterns have architectural properties

You cannot just replace one design pattern with another one

Guidelines

Guideline: Think about your design/architecture first and about implementation details second.

Guideline: Consider only the patterns/abstractions that fit your design.

Guideline: Don’t design based on performance requirements.

Architecture and Design First!

Design Patterns

The Most Common Misconceptions

(2 of N)

Klaus Iglberger, Meeting C++ 2024

klaus.iglberger@gmx.de