

Two silver bullets are shown against a dark background. The bullet on the left is positioned horizontally and has the words "THE ONE" engraved on its side. The bullet on the right is also horizontal but angled downwards and has the word "SOLUTION" engraved on its side. The lighting creates highlights on the metallic surfaces of the bullets.

# There Is No Silver Bullet

Klaus Iglberger, Meeting C++ 2024

[klaus.iglberger@gmx.de](mailto:klaus.iglberger@gmx.de)

C++ Trainer/Consultant

Author of “C++ Software Design”

Chair of the CppCon Back-to-Basics track

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

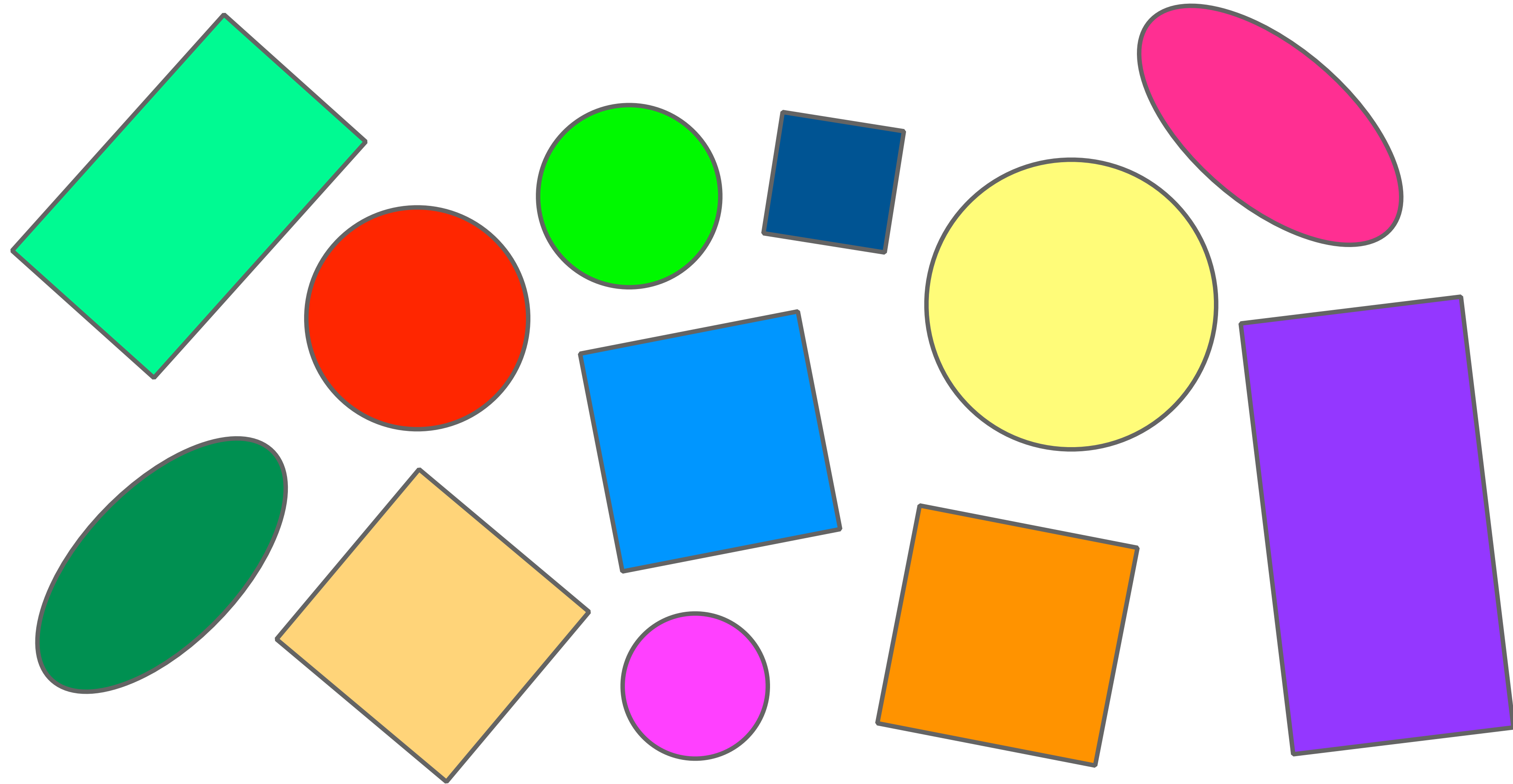
Email: [klaus.iglberger@gmx.de](mailto:klaus.iglberger@gmx.de)



**Klaus Iglberger**

# 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)

Low level (volatile, malleable, high dependencies)

Architectural  
Boundary

```
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

Your Code

Architectural  
Boundary

```
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;
};
```

```
int main()
{
    YourShapesFactory factory{};

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



# A Modern C++ Solution?

```
// ... 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
{
```

**std::unique\_ptr/smart pointers  
in combination with std::make\_unique**

**Move Semantics**

**override**

# A Modern C++ Solution?

---

```
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 )
{
    Shapes shapes = factory.create( filename );
    drawAllShapes( shapes );
}
```

auto

Range-based for loop

std::string\_view

# A Modern C++ Solution?

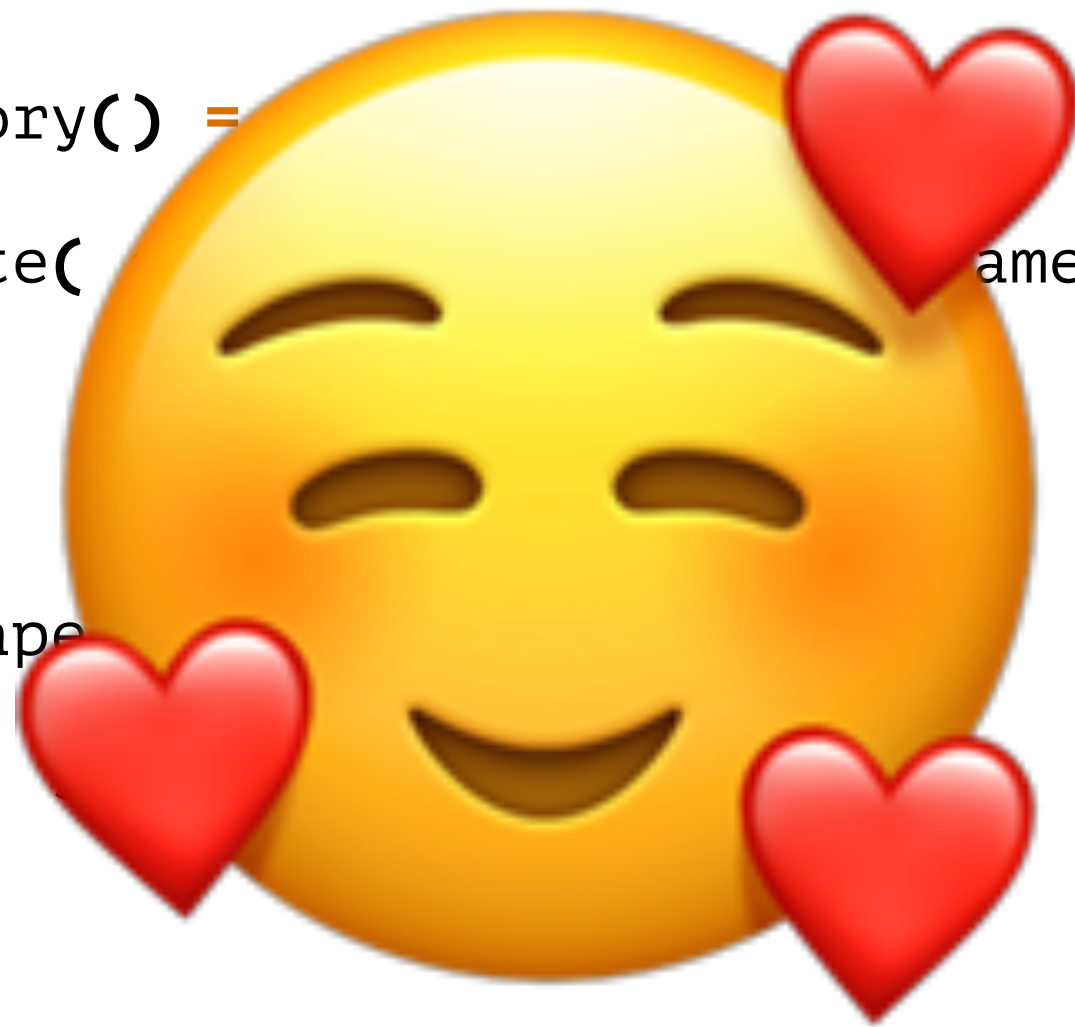
---

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

class ShapesFactory
{
public:
    virtual ~ShapesFactory() =
    virtual Shapes create( filename ) const = 0;
};

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

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



# A Modern C++ Solution?

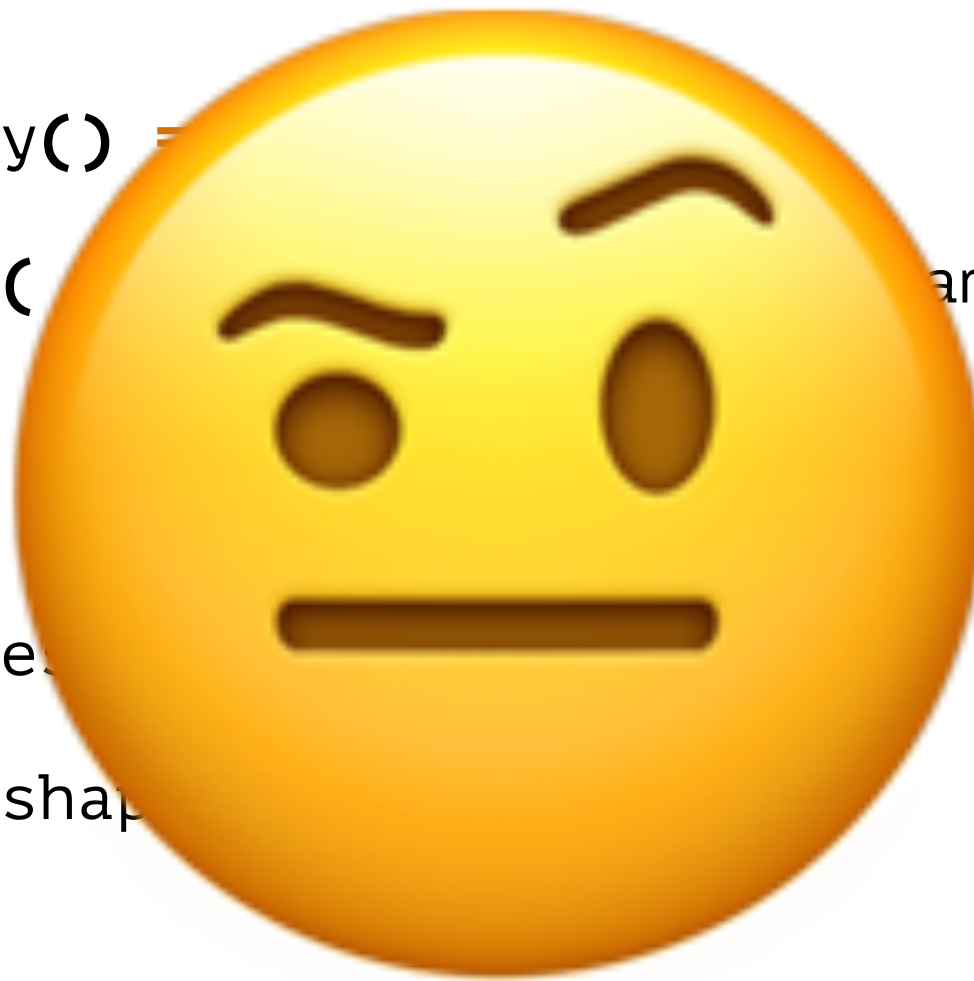
---

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

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



# A Modern C++ Solution?

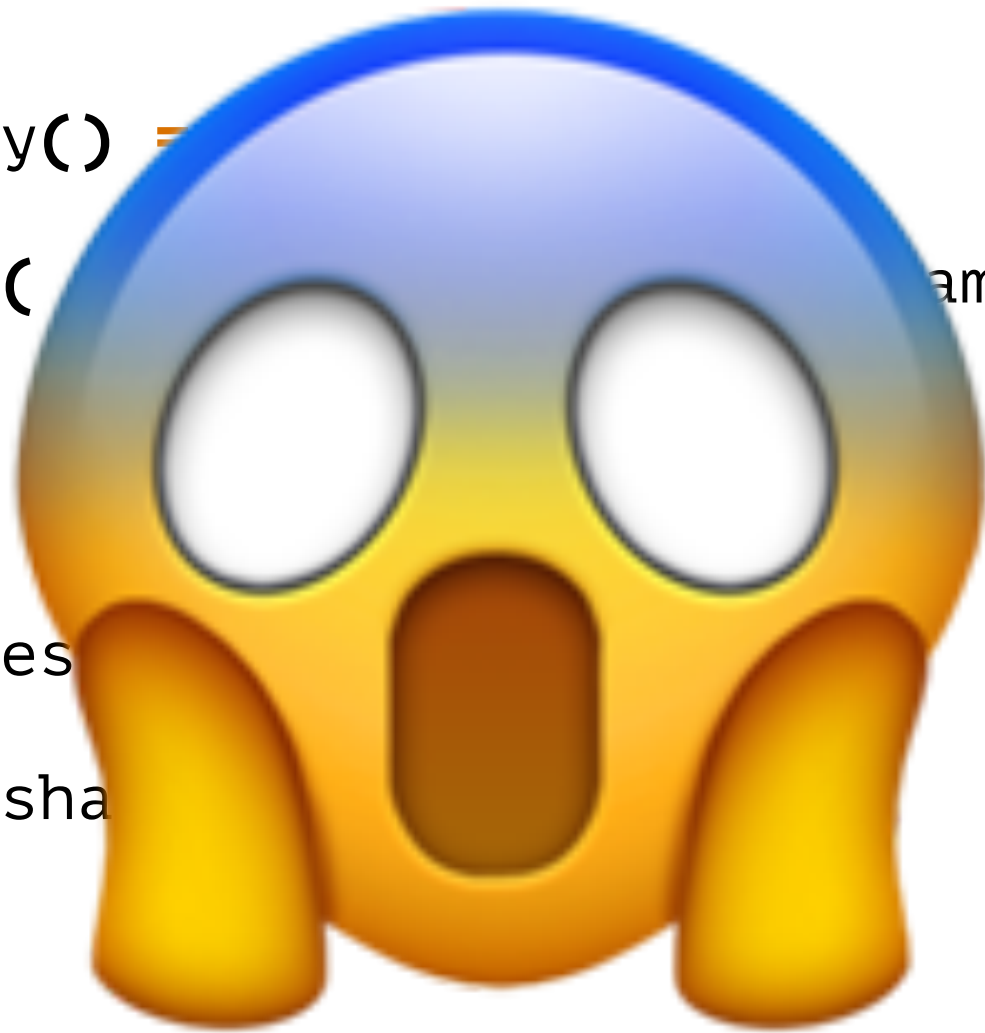
---

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

class ShapesFactory
{
public:
    virtual ~ShapesFactory() = default;
    virtual Shapes create( const std::string_view filename ) const = 0;
};

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

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

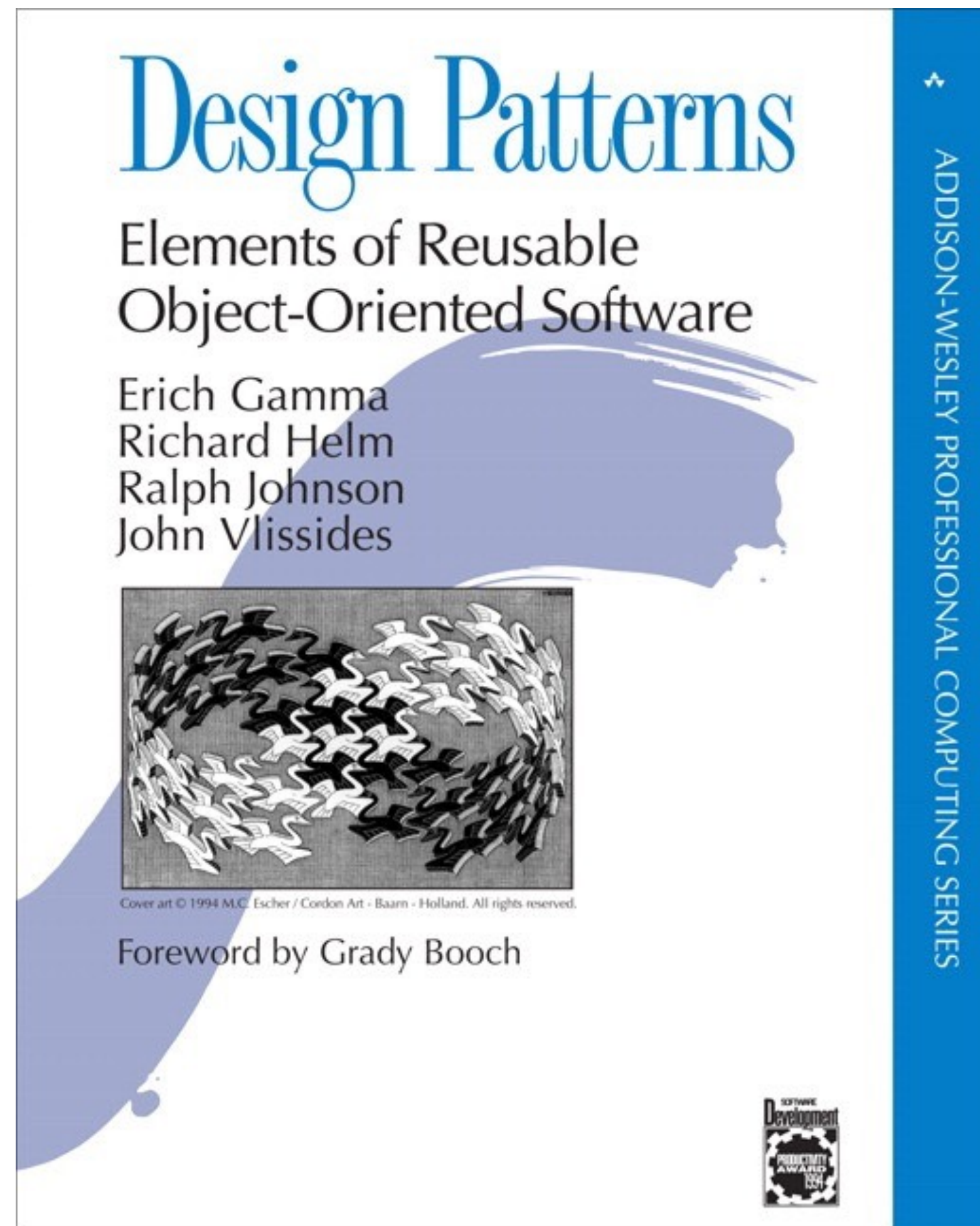


Yes... some of you are unhappy about this style of programming.



# The Philosophy of the 90s

---



# The Philosophy of the 90s

---

# The GOF Style





But let's be honest:  
this style is used in more than  
90% of all C++ projects.

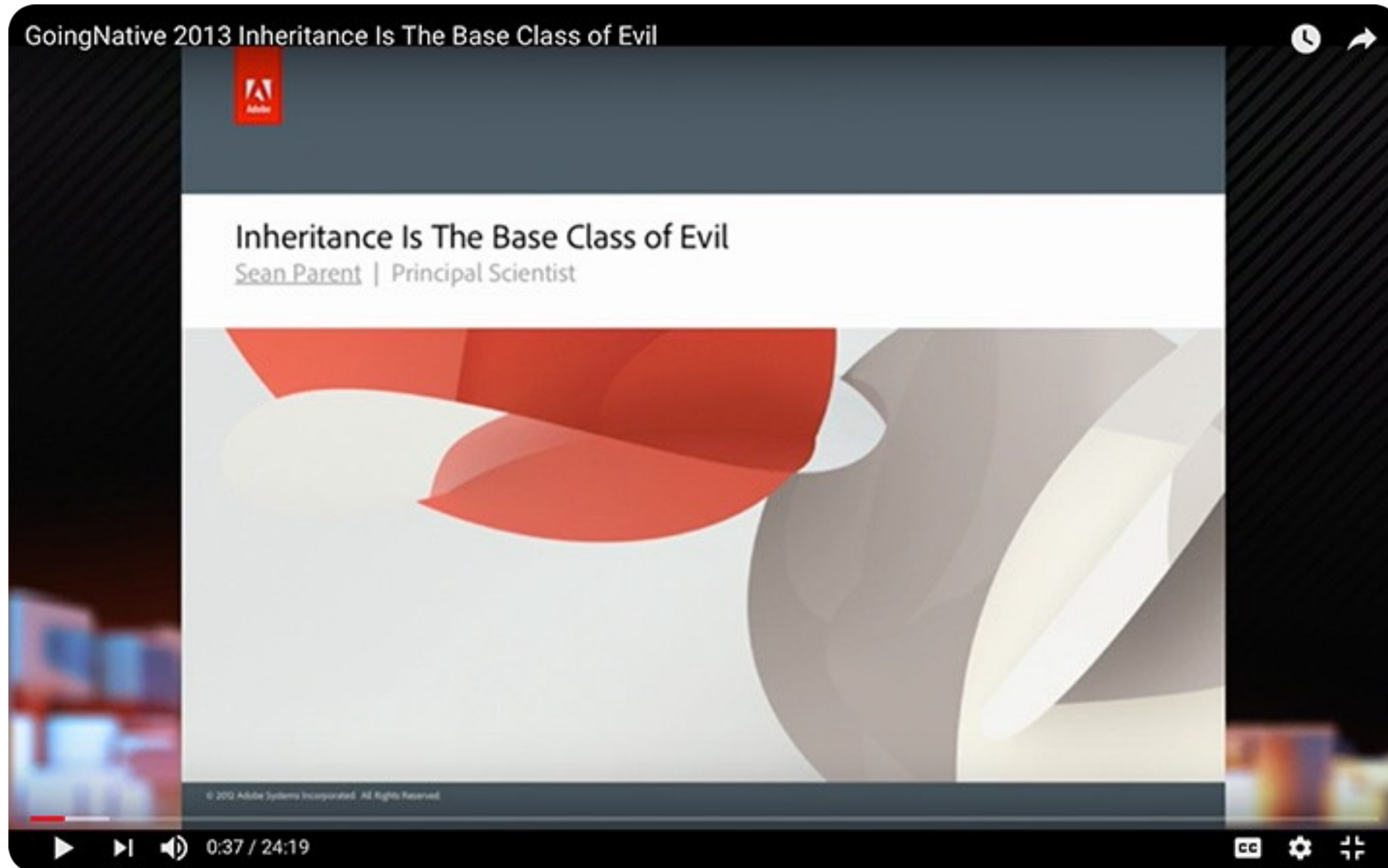


If anything about this should  
change, YOU have to spread  
the “news” ...



# The Fallen Paradigm (?)

---




# The Fallen Paradigm (?)

The image shows a video player interface. At the top left, the Cppcon logo is displayed with the text 'The C++ Conference September 13-18'. Below this, the speaker's name 'Phil Nash' is shown above a small video thumbnail of him. The main video area displays a presentation slide with a background of many yellow rubber ducks. The slide title is 'OO Considered Harmful' in large black font. Below the title, the Twitter handle '@phil\_nash' is visible. The video player controls at the bottom include a pause button, a play button, a volume icon, a progress bar showing '0:14 / 57:38', and a chapter selector showing 'Introduction >'. Other icons for closed captions, settings, and full screen are also present.


# The Fallen Paradigm (?)

Cppcon | 2019  
The C++ Conference | cppcon.org



Jon Kalb

Object-oriented programming is not what the cool kids are doing in C++. They are doing things at compile time, functional programming, ... Object-oriented programming, this is so 90s ...



Back to Basics:  
Object-Oriented  
Programming

Video Sponsorship Provided By:  
ansatz

1:27 / 59:58

2

# The Fallen Paradigm (?)

The image shows a video player interface for a presentation at CppCon 2018. The main content area displays a slide with the text "Can a browser engine be successful with data-oriented design?". The speaker is identified as Stoyan Nikolov. The video title is "OOP is dead, long live Data-oriented design". The video player includes a progress bar at the bottom left showing 2:49 / 1:00:45, and a control bar at the bottom right with the CppCon.org logo and various icons.

cppcon | 2018  
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

Can a browser engine be successful with data-oriented design?

STOYAN NIKOLOV

OOP is dead, long live Data-oriented design

CppCon 2018 | @stoyannk 3

2:49 / 1:00:45

CppCon.org

# The Fallen Paradigm (?)

---



## Using Modern C++ to Eliminate Virtual Functions

JONATHAN GOPEL



20  
22



September 12th-16th

# 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
};
```

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

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

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

# 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
};
```

std::variant replaces the Shape base class



```
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" ) {
```

# 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>;
```

We now utilize a vector of values instead of pointers



```
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" ) {
```

# 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;
}
};
```

```
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>;
```

Replacing another inheritance hierarchy with std::variant




# 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, ...)  
};
```



Again, no inheritance necessary!

```
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 )
```



# 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>;
```

And another inheritance hierarchy gone!



```
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 )
```

# 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 );  
    }  
}
```

A runtime dispatch on  
two variants!



```
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

---

```
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?

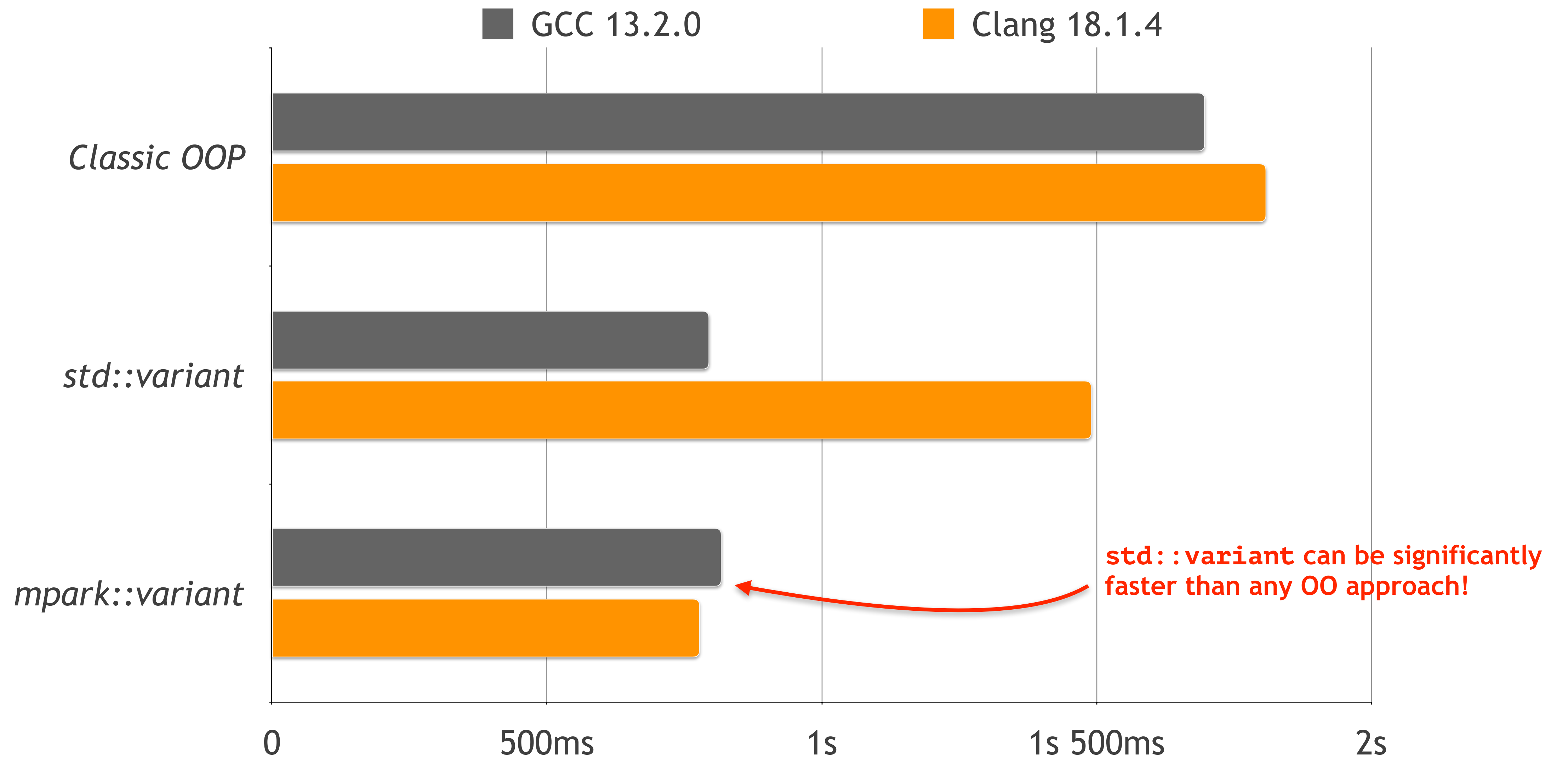


# Performance Comparison

---

- 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



**`std::variant` can be significantly faster than any OO approach!**



*std::variant*

*One feature to save them all,  
one feature to enlighten them,  
one feature to bring them all,  
and in C++ heaven bind them.*

*(Ancient folklore)*

# The Rising Paradigm (?)


Cppcon | 2019  
The C++ Conference | cppcon.org

Object-oriented programming is not what **the cool kids** are doing in C++. **They are doing** things at compile time, **functional programming**, ...  
Object-oriented programming, this is so 90s ...

Jon Kalb

Back to Basics:  
Object-Oriented  
Programming

Video Sponsorship Provided By:  
ansatz



1:27 / 59:58

2

# The Rising Paradigm (?)

---



# The Rising Paradigm (?)

---

NDC { London }  
16-20 January 2017  
Inspiring Software Developers  
since 2008

**Functional C++**

NDC { London }

0:10 / 1:01:00 • Intro >

⏪ ⏩ 🔊 ⏸ ⌂ ⚙️ 📺 🗨️ 🗑️

The image shows a YouTube video player interface. The main content is a presentation slide with a blue background and the text 'Functional C++' in large, bold, yellow font. In the top left corner of the video frame, there is a black box containing white text: 'NDC { London }', '16-20 January 2017', and 'Inspiring Software Developers since 2008'. In the bottom left corner of the video frame, there is a smaller inset video showing a man in a dark shirt speaking at a podium with 'NDC { London }' on it. The video player controls at the bottom include a progress bar, play/pause, next, volume, and a timestamp '0:10 / 1:01:00 • Intro >'. On the right side of the controls are icons for closed captions, settings, full screen, and other video controls.

# The Rising Paradigm (?)



The image shows a video player interface for a presentation. The main content area has a dark blue background with the title "The Imperatives Must Go" in yellow text. Below the title, the name "VICTOR CIURA" is displayed in yellow. In the top right corner, there is a pink notification bubble containing a white plus sign and the number "22" with a small "i" icon. The bottom of the player features a control bar with the Cppcon logo (a stylized plus sign in a circle) and the text "Cppcon The C++ Conference". To the right of the logo, the video progress is shown as "0:01 / 57:33". Further right are icons for play/pause, volume, and a settings gear. On the far right of the control bar, there is a large "20 22" logo, a date "September 12th-16th", and icons for full screen, share, and other video controls.

**The Imperatives Must Go**

**VICTOR CIURA**

Cppcon  
The C++ Conference

0:01 / 57:33

20 22  
September 12th-16th

# 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->draw( s ); }, drawer, shape );
```

```
  }
```

```
}
```

```
Shapes createAndDrawShapes( string_view filename, Drawer drawer )
```

```
{
```

```
  Shapes shapes = std::vector<Shape>{};
```

```
  drawAllShapes( shapes );
```

```
}
```

```
int main()
```

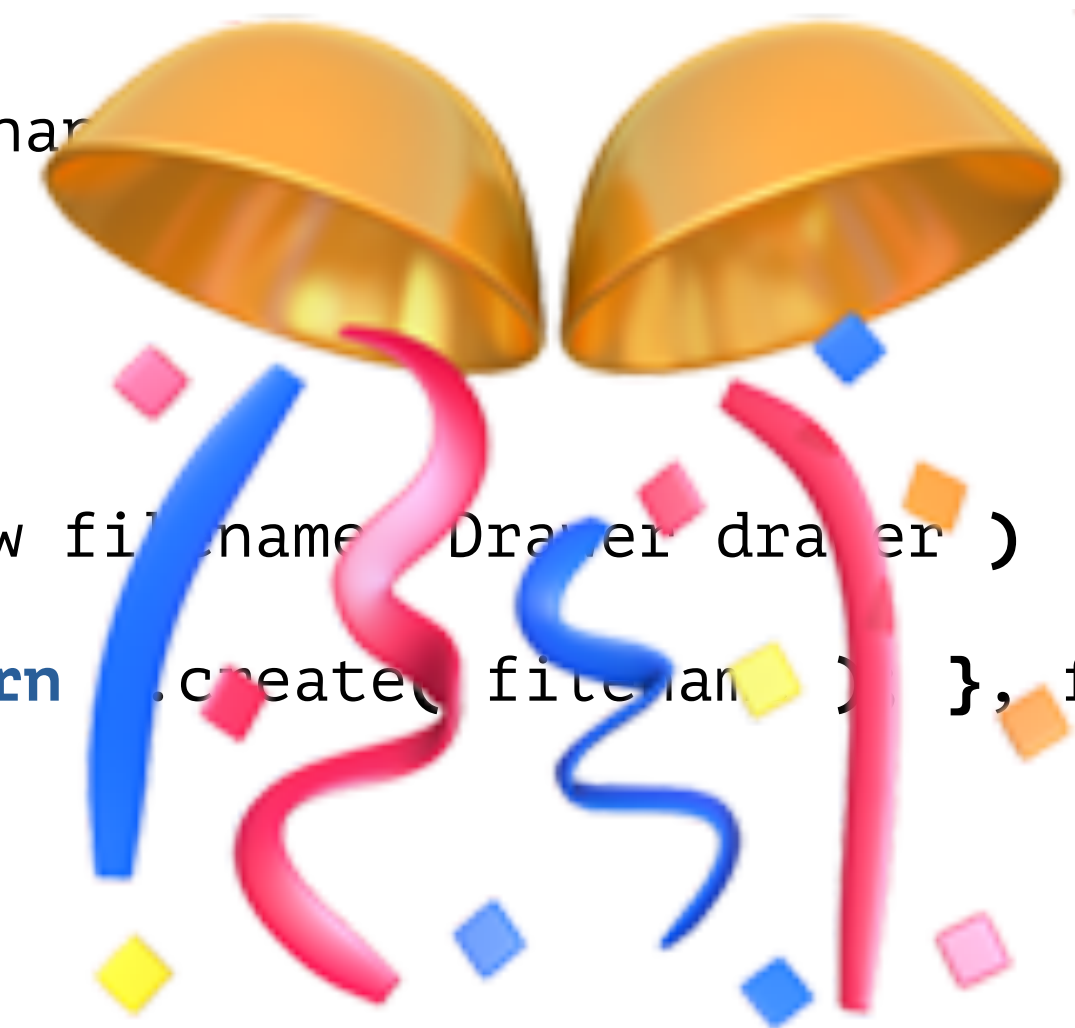
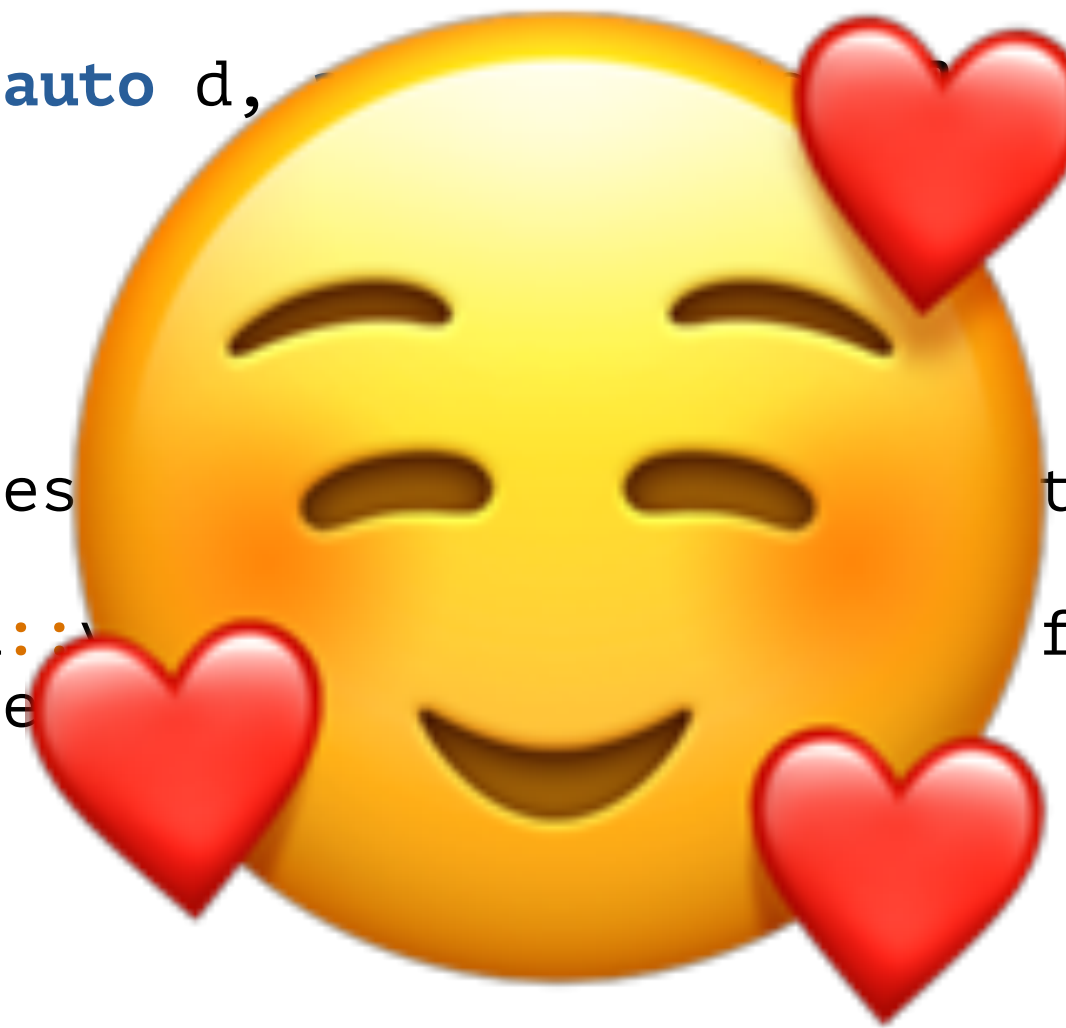
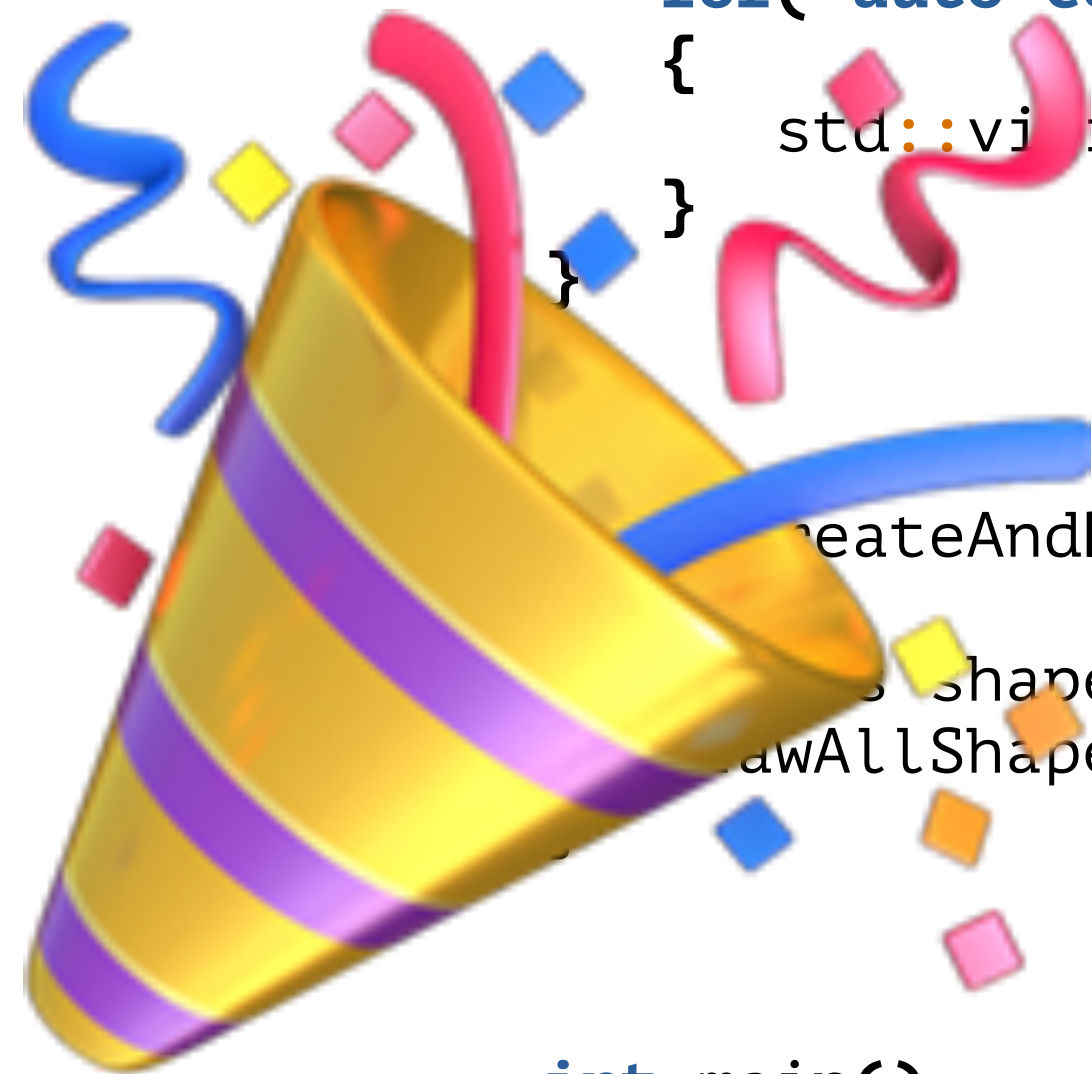
```
{
```

```
  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 );
}
```

These two functions belong to me...

# 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, ...)  
};
```

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

```
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 );  
    }  
}
```

# 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>;
```

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

```
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

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

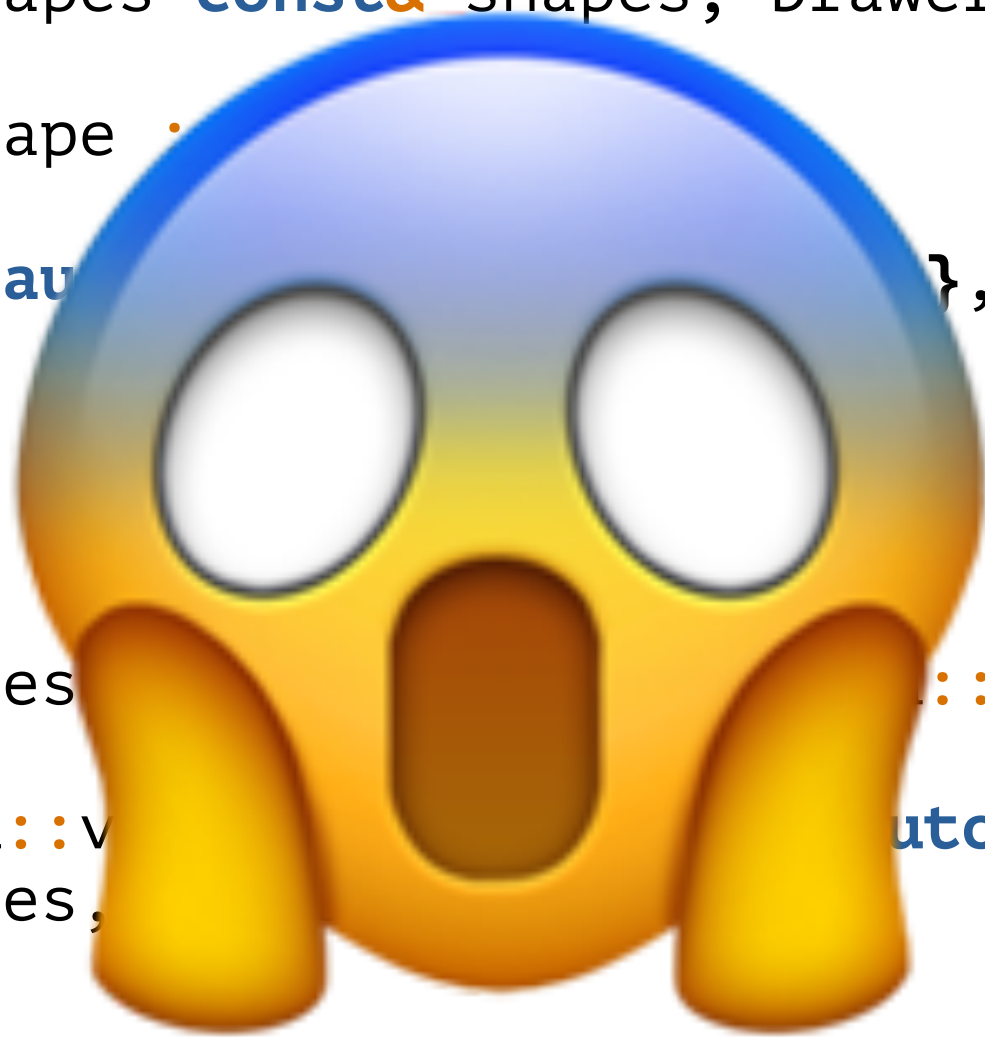
# A Truly Modern C++ Solution: std::variant

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

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

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

```
void createAndDrawShapes( ShapesFactory f, std::string_view filename, Drawer drawer )  
{  
    Shapes shapes = std::variant<ShapesFactory>{ f };  
    drawAllShapes( shapes, drawer );  
}
```



My Code

Your Code

Architectural  
Boundary

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

This is an architectural disaster, a total failure!





*std::variant*

*One feature to fool them all,  
one feature to blind them,  
one feature to bring them all,  
and with dependencies bind them.*

*(Ancient folklore)*

# 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

Your Code

Architectural  
Boundary

```
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

Your Code

Architectural  
Boundary

```
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

Your Code

Architectural  
Boundary

```
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

```
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 (?)

---

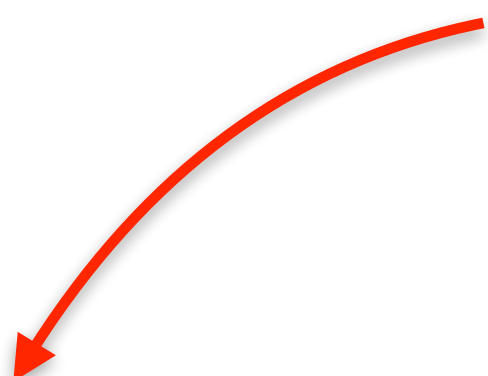
```
double getSide() const noexcept;
// ... getCenter(), getRotation(), ...

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

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 );
}
```

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



My Code

Your Code

Architectural  
Boundary

80



# Templates to the Rescue (?)

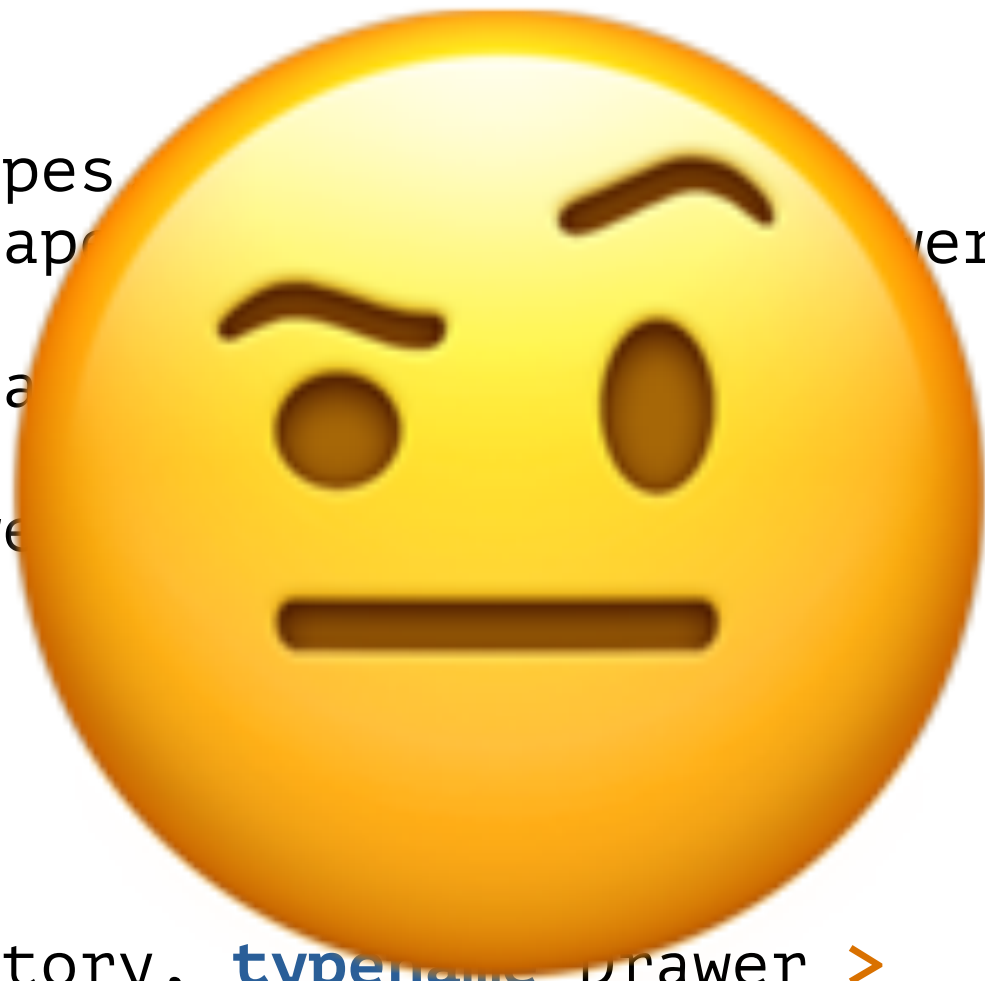
---

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

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

template< typename Shapes
void drawAllShapes( Shapes shapes, Drawer 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 );
}
```



There is no silver bullet!



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



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



# std::variant vs. Virtual Functions

---

<i>std::variant</i>	<i>Virtual Functions</i>
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

# Architecture/Design matters!



The design plays a much more central role in the success of a project than any feature could ever do.



Have you ever heard of a project that failed because some feature could not be used?





Do you happen to work in a project with a complete architectural document, including ADRs?



In C++, features and language details  
are at the center of attention.



6 (?) out of 48 talks at Meeting C++  
2024 are related to Software Design.

**Meeting C++  
2024**

But... `std::variant` is so much faster. I still want to use it...



Well, then use it, but create  
an **Architectural Decision  
Record (ADR)**




Does this mean we have to  
go back to object-oriented  
programming?



# Do We Have to Use the Fallen Paradigm?

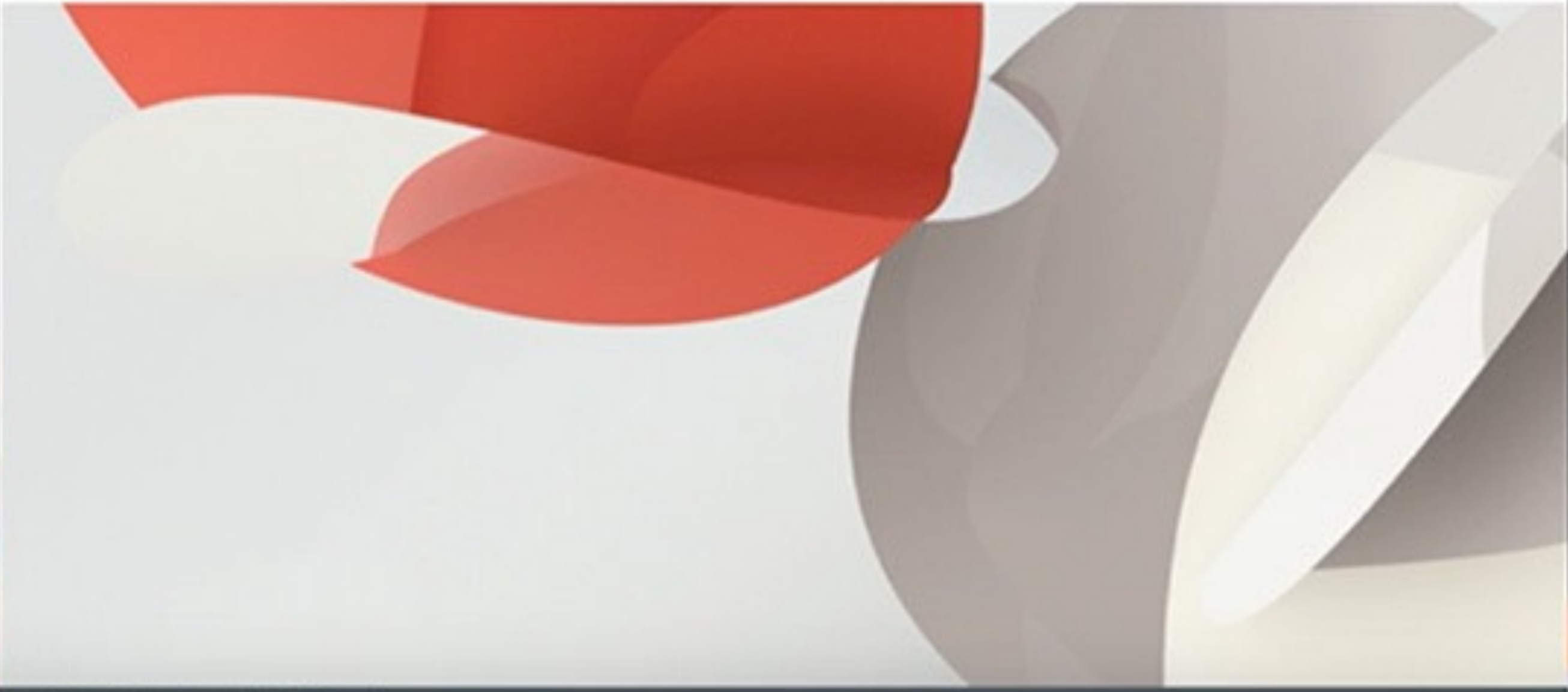
---

GoingNative 2013 Inheritance Is The Base Class of Evil




## Inheritance Is The Base Class of Evil

[Sean Parent](#) | Principal Scientist



© 2013 Adobe Systems Incorporated. All Rights Reserved.

0:37 / 24:19



# Value-Based Object-Oriented Programming

**Meeting C++ 2023**  
Prog C++ - Ivan Čukić

think-cell

PROG C++

Prog C++ is a broad genre of C++ code. The style was an emergence of psychedelic developers who abandoned standard C with classes traditions in favour of instrumentation and compositional techniques more frequently associated with generic, functional, or value-based object oriented coding practices.

INTRODUCTION  
WRAPS  
SWAPS  
STATES  
ERRORS  
VALUES  
SAFETY

Look here: value-based object-oriented coding practices

Meeting C++ 2023  
Ivan Čukić [kdab.com](http://kdab.com), [cukic.co](http://cukic.co)

Meeting C++ 2023  
Bloomberg think-cell

4:07 / 1:50:35



# Interlude: A new Era of C++

The image is a screenshot of a video player. The video content is a presentation slide from 'Meeting C++ 2023'. The slide title is 'Prog C++ - Ivan Čukić'. The main text on the slide reads: 'Prog C++ is a broad genre of C++ code. The style was an emergence of psychedelic developers who abandoned standard C with classes traditions in favour of instrumentation and compositional techniques more frequently associated with generic, functional, or value-based object oriented coding practices.' The slide also features a list of terms: 'INTRODUCTION', 'WRAPS', 'SWAPS', 'STATES', 'ERRORS', 'VALUES', and 'SAFETY'. At the bottom of the slide, it says 'Meeting C++ 2023' and 'Ivan Čukić | kdab.com, cukic.co'. The video player interface shows the speaker, Ivan Čukić, at a podium. The video title is 'Meeting C++ 2023' and the duration is 4:07 / 1:50:35. The video player controls include play, volume, and full screen buttons.

**Meeting C++ 2023**  
Prog C++ - Ivan Čukić

think-cell

PROG C++

Prog C++ is a broad genre of C++ code. The style was an emergence of psychedelic developers who abandoned standard C with classes traditions in favour of instrumentation and compositional techniques more frequently associated with generic, functional, or value-based object oriented coding practices.

INTRODUCTION  
WRAPS  
SWAPS  
STATES  
ERRORS  
VALUES  
SAFETY

Meeting C++ 2023  
Ivan Čukić | kdab.com, cukic.co

4:07 / 1:50:35

Are you tired of the term  
**Modern C++?**

Do you also feel that after  
13 years modern is not  
modern anymore?

Do you also feel that the term is misused for any C++ code?

**Modern C++ is not about features, not about standards, it's about a philosophy.**

We should rename  
**Modern C++!**

# What about **Progressive C++?**



Ivan Čukić

(Creator of the term “Progressive C++”)



# A Value-Based Object-Oriented Solution

---

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

    virtual void draw() const = 0;
    // ... several other virtual functions, including 'clone()'
};

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 Value-Based Object-Oriented Solution

---

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

    virtual void draw() const = 0;
    // ... several other virtual functions, including 'clone()'
};

class Circle : public ShapeConcept
{
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 Value-Based Object-Oriented Solution

---

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

    virtual void draw() const = 0;
    // ... several other virtual functions, including 'clone()'
};

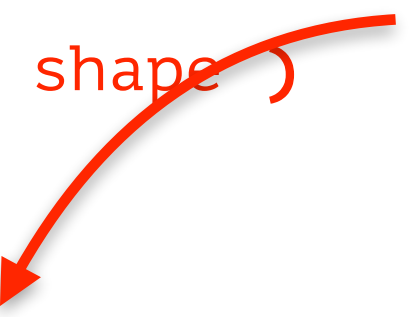
template< typename ConcreteShape >
class ShapeModel : public ShapeConcept
{
public:
    explicit ShapeModel( ConcreteShape shape )
        : shape_{shape}
    {}

    void draw() const override { /*...*/ }
    // ... several other virtual functions, including 'clone()'

private:
    ConcreteShape shape_;
};

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

How does a model draw its shape?



# A Value-Based Object-Oriented Solution

---

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

    virtual void draw() const = 0;
    // ... several other virtual functions, including 'clone()'
};

template< typename ConcreteShape, typename DrawStrategy >
class ShapeModel : public ShapeConcept
{
public:
    explicit ShapeModel( ConcreteShape shape, DrawStrategy drawer )
        : shape_{shape}
        , drawer_{drawer}
    {}

    void draw() const override { drawer_(shape_); }
    // ... several other virtual functions, including 'clone()'

private:
    ConcreteShape shape_;
    DrawStrategy drawer_;
};

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

# A Value-Based Object-Oriented Solution

---

```
    }, drawer_{drawer_}
};

void draw() const override { drawer_(shape_); }
// ... several other virtual functions, including 'clone()'

private:
    ConcreteShape shape_;
    DrawStrategy drawer_;
};

class Circle : public ShapeConcept
{
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 Value-Based Object-Oriented Solution

---

```
    }, drawer_{drawer_}
    {}

    void draw() const override { drawer_(shape_); }
    // ... several other virtual functions, including 'clone()'

private:
    ConcreteShape shape_;
    DrawStrategy drawer_;
};

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

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

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

# A Value-Based Object-Oriented Solution

---

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

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

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

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

    virtual void draw() const = 0;
    // ... several other virtual functions, including 'clone()'
};

template< typename ConcreteShape, typename DrawStrategy >
class ShapeModel : public ShapeConcept
{
public:
    explicit ShapeModel( ConcreteShape shape, DrawStrategy drawer )
```

# A Value-Based Object-Oriented Solution

```
};  
  
// ... remaining data members  
  
class ShapeConcept  
{  
public:  
    virtual ~ShapeConcept() = default;  
  
    virtual void draw() const = 0;  
    // ... several other virtual functions, including 'clone()'  
};
```

```
template< typename ConcreteShape, typename DrawStrategy >  
class ShapeModel : public ShapeConcept  
{  
public:  
    explicit ShapeModel( ConcreteShape shape, DrawStrategy drawer )  
        : shape_{shape}  
        , drawer_{drawer}  
    {}  
  
    void draw() const override { drawer_(shape_); }  
    // ... several other virtual functions, including 'clone()'  
  
private:  
    ConcreteShape shape_;  
    DrawStrategy drawer_;  
};
```

The ShapeModel wraps the polymorphic drawing behavior around any concrete shape type



# A Value-Based Object-Oriented Solution

---

```
class Shape
{
public:
    // ...

private:
    class ShapeConcept
    {
    public:
        virtual ~ShapeConcept() = default;

        virtual void draw() const = 0;
        // ... several other virtual functions, including 'clone()'
    };

    template< typename ConcreteShape, typename DrawStrategy >
    class ShapeModel : public ShapeConcept
    {
    public:
        explicit ShapeModel( ConcreteShape shape, DrawStrategy drawer )
            : shape_{shape}
            , drawer_{drawer}
        {}

        void draw() const override { drawer_(shape_); }
        // ... several other virtual functions, including 'clone()'

    private:
        ConcreteShape shape_;
        DrawStrategy drawer_;
    };
};
```

# A Value-Based Object-Oriented Solution

---

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

    virtual void draw() const = 0;
    // ... several other virtual functions, including 'clone()'
};

template< typename ConcreteShape, typename DrawStrategy >
class ShapeModel : public ShapeConcept
{
public:
    explicit ShapeModel( ConcreteShape shape, DrawStrategy drawer )
        : shape_{shape}
        , drawer_{drawer}
    {}

    void draw() const override { drawer_(shape_); }
    // ... several other virtual functions, including 'clone()'

private:
    ConcreteShape shape_;
    DrawStrategy drawer_;
};

std::unique_ptr<ShapeConcept> pimpl_;
};
```

# A Value-Based Object-Oriented Solution

---

```
class Shape
{
public:
    // ...

private:
    class ShapeConcept
    {
public:
        virtual ~ShapeConcept() = default;

        virtual void draw() const = 0;
        // ... several other virtual functions, including 'clone()'
    };

    template< typename ConcreteShape, typename DrawStrategy >
    class ShapeModel : public ShapeConcept
    {
public:
        explicit ShapeModel( ConcreteShape shape, DrawStrategy drawer )
            : shape_{shape}
            , drawer_{drawer}
        {}

        void draw() const override { drawer_(shape_); }
        // ... several other virtual functions, including 'clone()'

private:
        ConcreteShape shape_;
        DrawStrategy drawer_;
    };
};
```

# A Value-Based Object-Oriented Solution

---

```
class Shape
{
public:
    template< typename ConcreteShape, typename DrawStrategy >
    Shape( ConcreteShape shape, DrawStrategy drawer )
        : pimpl_{ std::make_unique<ShapeModel<ConcreteShape,DrawStrategy>>(shape, drawer) }
    {}

    // ...

```

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

        virtual void draw() const = 0;
        // ... several other virtual functions, including 'clone()'
    };

```

```
template< typename ConcreteShape, typename DrawStrategy >
class ShapeModel : public ShapeConcept
{
public:
    explicit ShapeModel( ConcreteShape shape, DrawStrategy drawer )
        : shape {shape}

```

# A Value-Based Object-Oriented Solution

---

```
class Shape
{
public:
    template< typename ConcreteShape, typename DrawStrategy >
    Shape( ConcreteShape shape, DrawStrategy drawer )
        : pimpl_{ std::make_unique<ShapeModel<ConcreteShape,DrawStrategy>>(shape, drawer) }
    {}

    void draw() const { pimpl_->draw(); }

    // ...

private:
    class ShapeConcept
    {
public:
        virtual ~ShapeConcept() = default;

        virtual void draw() const = 0;
        // ... several other virtual functions, including 'clone()'
    };

    template< typename ConcreteShape, typename DrawStrategy >
    class ShapeModel : public ShapeConcept
    {
public:
        explicit ShapeModel( ConcreteShape shape, DrawStrategy drawer )
            : shape {shape}
```

# A Value-Based Object-Oriented Solution

---

```
class Shape
{
public:
    template< typename ConcreteShape, typename DrawStrategy >
    Shape( ConcreteShape shape, DrawStrategy drawer )
        : pimpl_{ std::make_unique<ShapeModel<ConcreteShape,DrawStrategy>>(shape, drawer) }
    {}

    void draw() const { pimpl_->draw(); }

    Shape( Shape const& );
    Shape( Shape&& );
    ~Shape() = default;
    Shape& operator=( Shape const& );
    Shape& operator=( Shape&& );

private:
    class ShapeConcept
    {
public:
        virtual ~ShapeConcept() = default;

        virtual void draw() const = 0;
        // ... several other virtual functions, including 'clone()'
    };

    template< typename ConcreteShape, typename DrawStrategy >
    class ShapeModel : public ShapeConcept
    {
public:
        explicit ShapeModel( ConcreteShape shape, DrawStrategy drawer )
            : shape {shape}
```

The copy operations are based on the virtual clone() function of the ShapeConcept abstraction

# A Value-Based Object-Oriented Solution

---

```
class Shape
{
public:
    template< typename ConcreteShape, typename DrawStrategy >
    Shape( ConcreteShape shape, DrawStrategy drawer )
        : pimpl_{ std::make_unique<ShapeModel<ConcreteShape,DrawStrategy>>(shape, drawer) }
    {}

    void draw() const { pimpl_->draw(); }

    Shape( Shape const& );
    Shape( Shape&& );
    ~Shape() = default;
    Shape& operator=( Shape const& );
    Shape& operator=( Shape&& );

private:
    class ShapeConcept
    {
public:
        virtual ~ShapeConcept() = default;

        virtual void draw() const = 0;
        // ... several other virtual functions, including 'clone()'
    };

    template< typename ConcreteShape, typename DrawStrategy >
    class ShapeModel : public ShapeConcept
    {
public:
        explicit ShapeModel( ConcreteShape shape, DrawStrategy drawer )
            : shape {shape}
```

## The Shape abstraction ...

- ... can do the same as the Shape base class.
- ... is faster due to fewer indirections.
- ... is a value (i.e. can be copied, moved, ...).
- ... simplifies the surrounding code.

## This is Type Erasure.

# A Value-Based Object-Oriented Solution

---

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

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

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

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

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

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



# A Value-Based Object-Oriented Solution

```
// ... several other virtual functions, including 'clone()'

private:
    ConcreteShape shape_;
    DrawStrategy drawer_;
};

std::unique_ptr<ShapeConcept> pimpl_;
};

using Shapes = std::vector<Shape>;

using ShapesFactory = std::function<Shapes(std::string_view)>;

void drawAllShapes( Shapes const& shapes )
{
    for( auto const& shape : shapes )
    {
        shape.draw();
    }
}

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

class OpenGLDrawer
{
```

A vector of values, not pointers



# A Value-Based Object-Oriented Solution

```
// ... several other virtual functions, including 'clone()'

private:
    ConcreteShape shape_;
    DrawStrategy drawer_;
};

std::unique_ptr<ShapeConcept> pimpl_;
};

using Shapes = std::vector<Shape>;

using ShapesFactory = std::function<Shapes(std::string_view)>;

void drawAllShapes( Shapes const& shapes )
{
    for( auto const& shape : shapes )
    {
        shape.draw();
    }
}

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

class OpenGLDrawer
{
```

Another Type Erasure abstraction



# A Value-Based Object-Oriented Solution

---

```
std::unique_ptr<ShapeConcept> pimpl_;\n};\n\nusing Shapes = std::vector<Shape>;\n\nusing ShapesFactory = std::function<Shapes(std::string_view)>;\n\nvoid drawAllShapes( Shapes const& shapes )\n{\n    for( auto const& shape : shapes )\n    {\n        shape.draw(); ← Value syntax, not pointer syntax\n    }\n}\n\nvoid createAndDrawShapes( ShapesFactory const& factory, std::string_view filename )\n{\n    Shapes shapes = factory( filename );\n    drawAllShapes( shapes );\n}\n\nclass OpenGLDrawer\n{\npublic:\n    explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}\n\n    void operator()( Circle const& circle ) const;\n\n    void operator()( Square const& square ) const;\n};
```

# A Value-Based Object-Oriented Solution

---

```
    std::unique_ptr<ShapeConcept> pimpl_;\n};\n\nusing Shapes = std::vector<Shape>;\n\nusing ShapesFactory = std::function<Shapes(std::string_view)>;\n\nvoid drawAllShapes( Shapes const& shapes )\n{\n    for( auto const& shape : shapes )\n    {\n        shape.draw();\n    }\n}\n\nvoid createAndDrawShapes( ShapesFactory const& factory, std::string_view filename )\n{\n    Shapes shapes = factory( filename );\n    drawAllShapes( shapes );\n}\n\nclass OpenGLDrawer\n{\npublic:\n    explicit OpenGLDrawer( /*... color, texture, transparency, ...*/ ) {}\n\n    void operator()( Circle const& circle ) const;\n\n    void operator()( Square const& square ) const;\n};
```

# A Value-Based Object-Oriented Solution

---

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

```
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, ...)  
};
```

No inheritance, but value-semantics.  
The same OpenGLDrawer as in the std::variant solution.

```
class YourShapesFactory  
{  
public:  
  Shapes operator()( std::string_view filename ) const  
  {  
    Shapes shapes{};  
    std::string shape{};  
  
    std::istringstream shape_file{ filename };  
  }
```

# A Value-Based Object-Oriented Solution

---

```
class YourShapesFactory
{
public:
    Shapes operator()( std::string_view filename ) const
    {
        Shapes shapes{};
        std::string shape{};

        std::istringstream shape_file{ filename };

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

        return shapes;
    }
};
```

Again, no inheritance, but value-semantics.

Creating values, storing values ...  
No dynamic memory allocation in our code.

# A Value-Based Object-Oriented Solution

---

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

return shapes;
};
```

```
int main()
{
    YourShapesFactory factory{};

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

And one more time, no pointer, no allocation, but value-semantics.



# A Value-Based Object-Oriented Solution

---

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

return shapes;
}
};

int main()
{
    YourShapesFactory factory{};

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



# A Value-Based Object-Oriented Solution

---

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

```
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, ...)
};
```

```
class YourShapesFactory
{
public:
    Shapes operator()( std::string_view filename ) const
    {
        Shapes shapes{};
        std::string shape{};

        std::istringstream shape_file{ filename };

        while( iss >> shape )
        {
```

# A Value-Based Object-Oriented Solution

---

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

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, ...)
};

class YourShapesFactory
{
public:
    Shapes operator()( std::string_view filename ) const
    {
        Shapes shapes{};
        std::string shape{};
    }
}
```

# A Value-Based Object-Oriented Solution

---


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

My Code

Architectural  
Boundary

---

Your Code

```
class Rectangle  Adding new shape types is easily possible
{
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
};
```

```
class OpenGLDrawer
{
```

# A Value-Based Object-Oriented Solution

---

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

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

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, ...)
};

class YourShapesFactory
{
public:
    Shapes operator()( std::string_view filename ) const
    {
        Shapes shapes{};
        std::string shape{};

        std::istringstream shape_file{ filename };

        while( iss >> shape )
```

# A Value-Based Object-Oriented Solution

---

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

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

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; ← Easy extension in your code only 😊

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

class YourShapesFactory
{
public:
    Shapes operator()( std::string_view filename ) const
    {
        Shapes shapes{};
        std::string shape{};

        std::istringstream shape_file{ filename };
    }
};
```

# A Value-Based Object-Oriented Solution

---

```
class YourShapesFactory
{
public:
    Shapes operator()( std::string_view filename ) const
    {
        Shapes shapes{};
        std::string shape{};

        std::istringstream shape_file{ filename };

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

        return shapes;
    }
};
```

# A Value-Based Object-Oriented Solution

---

```
class YourShapesFactory
{
public:
    Shapes operator()( std::string_view filename ) const
    {
        Shapes shapes{};
        std::string shape{};

        std::istringstream shape_file{ filename };

        while( iss >> shape )
        {
            if( shape == "circle" ) {
                double radius{};
                shape_file >> radius /* >> color, texture, transparency, ... */;
                shapes.emplace_back( Circle{radius}, OpenGLDrawer{ /*...*/ } );
            }
            else if( shape == "square" ) {
                double side{};
                iss >> side >> /* >> color, texture, transparency, ... */;
                shapes.emplace_back( Square{side}, OpenGLDrawer{ /*...*/ } );
            }
            else if( shape == "rectangle" ) {
                double width{}, height{};
                iss >> width >> height >> /* >> color, texture, transparency, ... */;
                shapes.emplace_back( Rectangle{width,height}, OpenGLDrawer{ /*...*/ } );
            }
            else {
                break;
            }
        }
    }
}
```

Again, easy extension in your code only 😊

# A Value-Based Object-Oriented Solution

---

```
class YourShapesFactory
{
public:
    Shapes operator()( std::string_view filename ) const
    {
        Shapes shapes{};
        std::string shape{};

        std::istringstream shape_file{ filename };

        while( iss >> shape )
        {
            if( shape == "circle" ) {
                double radius{};
                shape_file >> radius /* >> color, texture, transparency, ... */;
                shapes.emplace_back( Circle{radius}, OpenGLDrawer{ /*...*/ } );
            }
            else if( shape == "square" ) {
                double side{};
                iss >> side >> /* >> color, texture, transparency, ... */;
                shapes.emplace_back( Square{side}, OpenGLDrawer{ /*...*/ } );
            }
            else if( shape == "rectangle" ) {
                double width{}, height{};
                iss >> width >> height >> /* >> color, texture, transparency, ... */;
                shapes.emplace_back( Rectangle{width,height}, OpenGLDrawer{ /*...*/ } );
            }
            else {
                break;
            }
        }
    }
}
```



# A Value-Based Object-Oriented Solution

---

This solution is amazing:

- Changes/extensions work as expected
- The architectural boundaries are adhered to
- Fewer pointers
- Fewer virtual functions (i.e. only one indirection instead of two)
- Less inheritance
- Less manual memory management
- Better performance



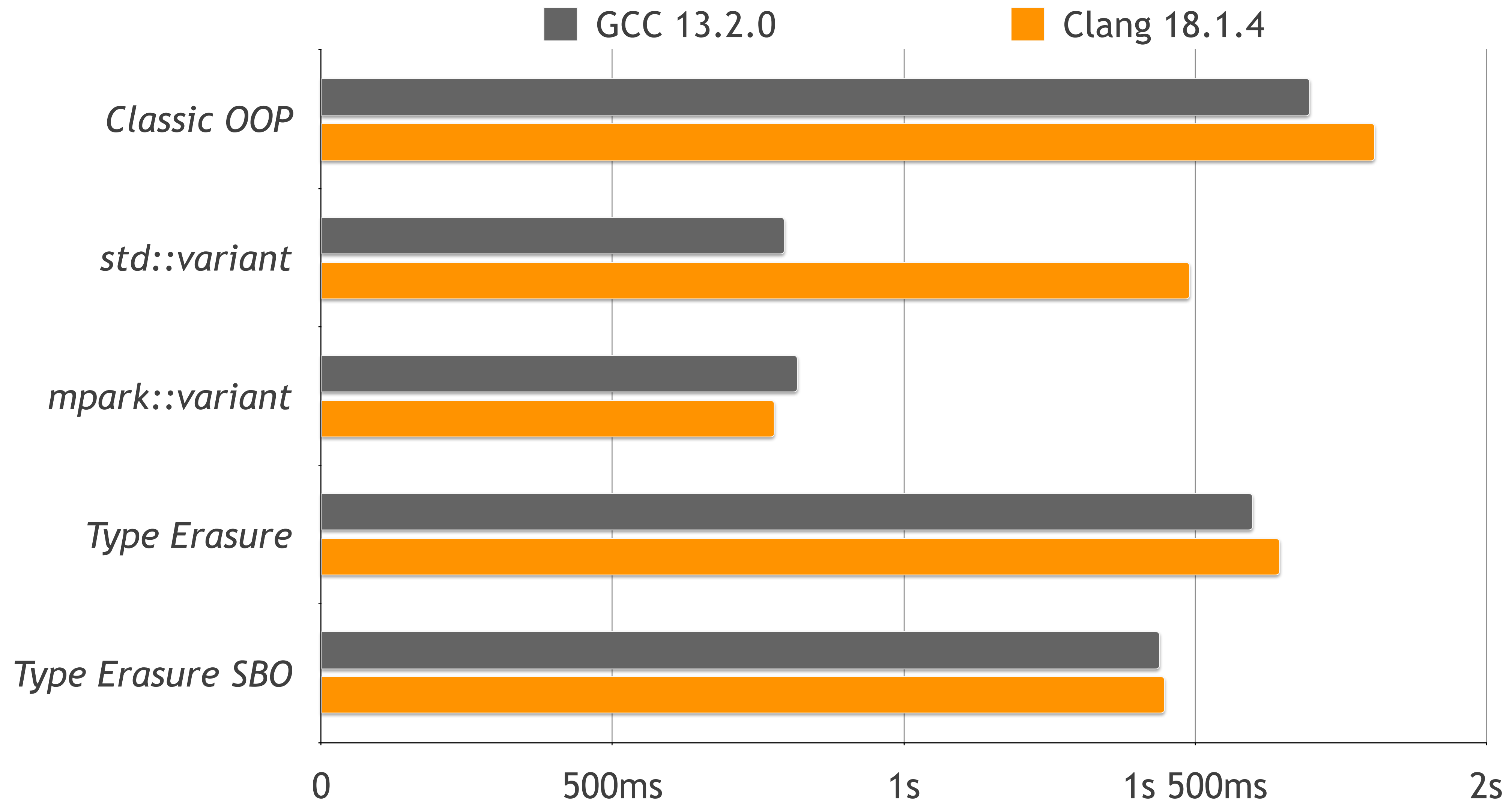
# Performance Comparison

---

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



# Performance Comparison



So, should we now use only  
value-based object-oriented  
programming?



No, of course not.  
Please also use functional  
programming solutions.



So, then, when should we use functional programming or object-oriented programming?



Of course the answer is:

**It depends**



# std::variant vs. Type Erasure

---

<i>std::variant</i>	<i>Type Erasure</i>
Functional programming	Object-oriented programming
Fixed set of types	Open set of types
Open set of operations	Closed set of operations
Best used in lower levels of the architecture (implementation details)	Best used in higher levels of the architecture



# The Two Sides of the Same Coin

---

This side: Object-Oriented  
Programming with Type Erasure



The other side: Functional  
Programming with `std::variant`

An ornate, gold-colored picture frame with intricate carvings and scrollwork, surrounding a white central area. The frame has a textured, metallic appearance and is set against a white background with faint, light-colored starburst patterns.

First Rule of Software Design:

**Think about the  
architectural  
properties of a  
solution first!**

# Think About the Architectural Properties First

---



The image shows a video player interface. The main content is a presentation slide with the following text:

**The Next Big Thing**  
Prepared for Meeting C++ 2018

Andrei Alexandrescu, Ph.D.  
andrei@erdani.com

November 15, 2018

On the right side, there is a vertical black box containing the following text:

Meeting C++ 2018  
Opening Keynote  
Andrei Alexandrescu  
The next big Thing

Below this box is a small video thumbnail showing a man in a black shirt standing on a stage. At the bottom of the video player, there is a control bar with the following elements from left to right: a play button, a volume icon, a progress bar showing 0:02 / 1:54:34, a pause button, a full screen button, a settings gear icon, a share icon, a comment icon, and a close icon.



Second Rule of Software Design:

**Consider both  
advantages and  
disadvantages of  
a solution!**

There is no silver bullet.

There are always pros and cons.

It always depends.

Thank you!

Two silver bullets are shown against a light gray background. The bullet on the left is positioned horizontally and has the words "THE ONE" engraved on its side. The bullet on the right is also horizontal but angled downwards and has the word "SOLUTION" engraved on its side. Both bullets have a metallic, reflective finish.

# There Is No Silver Bullet

Klaus Iglberger, Meeting C++ 2024

[klaus.iglberger@gmx.de](mailto:klaus.iglberger@gmx.de)

# TODOs

---

- 🕒 “Don’t Throw Coins, but think about the architectural properties of a solution”