Embrace No-Paradigm Programming

Klaus Iglberger, CppEurope 2020
klaus.iglberger@gmx.de
C++ Trainer since 2016

Author of the blaze C++ math library

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

Regular presenter at C++ conferences

Email: klaus.iglberger@gmx.de
What is C++?
What is C++?

Chandler Carruth and Titus Winters
{chandlerc,titus}@google.com
@chandlerc1024 and @TitusWinters
What is C++?
What kind of programming language is C++?
UNIX operating systems, and contain a rich set of data, structured control flow, machine-independent code. It also has also been called a “portable assembly language” because it has some low-level constructs such as pointers and addresses, has some low-level constructs such as weakly typed.

C was developed in the 1970s at Bell Labs. It was originally designed for the DEC PDP-11—whose operating system, C compiler, and Unix programs were all written in C. In 1988, an ANSI standard was issued, and was revised in 1999. C was the de facto standard for microcomputer programming in the 1980s and 1990s.

C++

C++, an object-oriented language founded on C, was developed at Bell Laboratories in the 1980s. In addition to being compatible with C, C++ provides classes, polymorphism, exception handling, templates, and it provides more robust type checking than C does. It also provides an extensive and powerful standard library.

C#

C# is a general-purpose, object-oriented language and programming environment developed by Microsoft with syntax similar to C, C++, and Java, and it provides extensive tools that aid development on Microsoft platforms.

Cobol
LearnCpp.com is a free website devoted to teaching you how to program in C++. Whether you've had any prior programming experience or not, the tutorials on this site will walk you through all the steps to write, compile, and debug your C++ programs, all with plenty of examples.

Becoming an expert won't happen overnight, but with a little patience, you'll get there. And LearnCpp.com will show you the way.

Having trouble remembering where you saw something? Not sure where to find something? Use our site index to find what you're looking for!

### Make Your Own Website Now

Go to Wix.com to Discover How to Develop an Online Presence for Your Business or Brand.

<table>
<thead>
<tr>
<th>Chapter 0</th>
<th>Introduction / Getting Started</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Introduction to these tutorials</td>
</tr>
<tr>
<td>0.2</td>
<td>Introduction to programming languages</td>
</tr>
<tr>
<td>0.3</td>
<td>Introduction to C/C++</td>
</tr>
<tr>
<td>0.4</td>
<td>Introduction to C++ development</td>
</tr>
<tr>
<td>0.5</td>
<td>Introduction to the compiler, linker, and libraries</td>
</tr>
<tr>
<td>0.6</td>
<td>Installing an Integrated Development Environment (IDE)</td>
</tr>
<tr>
<td>0.7</td>
<td>Compiling your first program</td>
</tr>
<tr>
<td>0.8</td>
<td>A few common C++ problems</td>
</tr>
<tr>
<td>0.9</td>
<td>Configuring your compiler: Build configurations</td>
</tr>
<tr>
<td>0.10</td>
<td>Configuring your compiler: Compiler extensions</td>
</tr>
<tr>
<td>0.11</td>
<td>Configuring your compiler: Warning and error levels</td>
</tr>
<tr>
<td>0.12</td>
<td>Configuring your compiler: Choosing a language standard</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 1</th>
<th>C++ Basics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Statements and the structure of a program</td>
</tr>
<tr>
<td>1.2</td>
<td>Comments</td>
</tr>
<tr>
<td>1.3</td>
<td>Introduction to variables</td>
</tr>
<tr>
<td>1.4</td>
<td>Variable assignment and initialization</td>
</tr>
<tr>
<td>1.5</td>
<td>Introduction to iostream: cout, cin, and endl</td>
</tr>
<tr>
<td>1.6</td>
<td>Uninitialized variables and undefined behavior</td>
</tr>
<tr>
<td>1.7</td>
<td>Keywords and naming identifiers</td>
</tr>
<tr>
<td>1.8</td>
<td>Introduction to literals and operators</td>
</tr>
<tr>
<td>1.9</td>
<td>Introduction to expressions</td>
</tr>
<tr>
<td>1.10</td>
<td>Developing your first program</td>
</tr>
<tr>
<td>1.x</td>
<td>Chapter 1 summary and quiz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2</th>
<th>C++ Basics: Functions and Files</th>
</tr>
</thead>
</table>
“... Overall you can get up and running pretty quickly with an object-oriented programming language like C++. ...”

Brad Larson, Episode 234 of CppCast, February, 13th, 2020
Inheritance Is The Base Class of Evil
Sean Parent | Principal Scientist
Why Are So Many Developers Hating on Object-Oriented Programming?

21 Aug 2019 12:00pm, by David Cassel
Can a browser engine be successful with data-oriented design?
If you want to improve the code quality in your organization, replace all of your coding guidelines with one goal:

No Raw Loops
// Solution 4a

auto solve() -> int {
    vector v = { 2, 1, 3, 5, 4 };  
    auto p = minmax_element(begin(v), end(v));
    return *p.second - *p.first;
}
Functional C++
@KevlinHenney
Functional C++ for Fun and Profit
Phil Nash
C++

From Wikipedia, the free encyclopedia

“CXX” redirects here. For the Roman numerals, see 120 (number).

C++ (/ ˈsɛpləg ˈplæs /) is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or “C with Classes”. The language has expanded significantly over time, and modern C++ has object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Oracle, and IBM, so it is available on many platforms.[8]

C++ was designed with a bias toward system programming and embedded, resource-constrained software and large systems, with performance, efficiency, and flexibility of use as its design highlights.[7] C++ has also been found useful in many other contexts, with key strengths being software infrastructure and resource-constrained applications,[8], including desktop applications, servers (e.g. e-commerce, Web search, or SQL servers), and performance-critical applications (e.g. telephone switches or space probes).[8]

C++ is standardized by the International Organization for Standardization (ISO), with the latest standard version ratified and published by ISO in December 2017 as ISO/IEC 14882:2017 (informally known as C++17).[9] The C++ programming language was initially standardized in 1998 as ISO/IEC 14882:1998, which was then amended by the C++03, C++11 and C++14 standards. The current C++17 standard supersedes these with new features and an enlarged standard library. Before the initial standardization in 1998, C++ was developed by Danish computer scientist Bjarne Stroustrup at Bell Labs since 1979 as an extension of the C language; he wanted an efficient and flexible language similar to C that also provided high-level features for program organization.[10] C++20 is the next planned standard, keeping with the current trend of a new version every three years.[11]
What kind of programming language is C++?
Procedural programming
Object-oriented programming
Functional programming
Generic programming
Multiparadigm programming
Let’s see ...
Disclaimer

- C++ is too complex to give a complete answer
- I consider only solutions with dynamic polymorphism
- I consider only the state of the art of C++
- There will be a lot of code ...
- ... but it’s not about the details, it’s about the design
- I deliberately choose a simple problem ...
Our Toy Problem: Drawing Shapes
enum ShapeType
{
    circle,
    square
};

class Shape
{
public:
    explicit Shape( ShapeType t )
    : type{ t }
    {}

    virtual ~Shape() = default;

    ShapeType getType() const noexcept;

private:
    ShapeType type;
};

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

    virtual ~Circle() = default;

    // A Procedural Solution

enum ShapeType {
    circle,
    square
};

class Shape {
public:
    explicit Shape( ShapeType t ) : type{ t } {}

    virtual ~Shape() = default;

    ShapeType getType() const noexcept;
private:
    ShapeType type;
};

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

    virtual ~Circle() = default;

};

A Procedural Solution
enum ShapeType
{
    circle,
    square
};

class Shape
{
    public:
        explicit Shape( ShapeType t )
        , type{ t }
        {}

    virtual ~Shape() = default;

    ShapeType getType() const noexcept;

    private:
        ShapeType type;
};

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

    virtual ~Circle() = default;

    A Procedural Solution
private:
    ShapeType type;
};

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

        virtual ~Circle() = default;
        double getRadius() const noexcept;
        // ... getCenter(), getRotation(), ...

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

void translate( Circle&, Vector3D const& );
void rotate( Circle&, Quaternion const& );
void draw( Circle const& );

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

```cpp
void translate( Circle&, Vector3D const& );
void rotate( Circle&, Quaternion const& );
void draw( Circle const& );

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

        virtual ~Square() = default;
        double getSide() const noexcept;
        // ... getCenter(), getRotation(), ...

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

void translate( Square&, Vector3D const& );
void rotate( Square&, Quaternion const& );
void draw( Square const& );

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
    for( auto const& s : shapes )
    {
        switch ( s->type )
        {
```
A Procedural Solution

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

void translate( Square&, Vector3D const& );
void rotate( Square&, Quaternion const& );
void draw( Square const& );

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
  for( auto const& s : shapes )
  {
    switch ( s->type )
    {
      case circle:
        DrawCircle( *static_cast<Circle const*>( s.get() ) );
        break;
      case square:
        DrawSquare( *static_cast<Square const*>( s.get() ) );
        break;
      default:
        break;
    }
  }
}

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

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

```
for( auto const& s : shapes )
{
    switch ( s->type )
    {
    case circle:
        DrawCircle( *static_cast<Circle const*>( s.get() ) );
        break;
    case square:
        DrawSquare( *static_cast<Square const*>( s.get() ) );
        break;
    }
}

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

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

    // Drawing all shapes
    draw( shapes );
}
Design Evaluation

<table>
<thead>
<tr>
<th>Enum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th>Addition of shapes (OCP)</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enum</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th>Enum</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = very bad, ..., 9 = very good</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The SOLID Principles

**Single Responsibility Principle**

**Open-Closed Principle**

**Liskov Substitution Principle**

**Interface Segregation Principle**

**Dependency Inversion Principle**
## Design Evaluation

<table>
<thead>
<tr>
<th>Enum</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>very bad, ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>very bad, ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>very bad, ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>very bad, ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>very bad, ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>very bad, ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>very bad, ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>very bad, ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>very good</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th>Enum</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*1 = very bad, ..., 9 = very good*
Design Evaluation

<table>
<thead>
<tr>
<th>Enum</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th>Enum</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th>Enum</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>7</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th>Enum</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
Performance Results

Benchmarks!

MacBook Pro 13-inch, MacOS 10.14.6 (Mojave), Intel Core i7, 2.7Ghz, 16 GByte, Compilation flags:  -std=c++17 -O3 -DNDEBUG
100 random shapes, 2.5M updates each, normalized to the enum solution
Performance Results

GCC-9.2  |  Clang-9
---|---
Enum  |  1

MacBook Pro 13-inch, MacOS 10.14.6 (Mojave), Intel Core i7, 2.7Ghz, 16 GByte, Compilation flags: -std=c++17 -O3 -DNDEBUG
100 random shapes, 2.5M updates each, normalized to the enum solution
# Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enum</strong></td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
class Shape
{
public:
    Shape() = default;
    virtual ~Shape() = default;

    virtual void translate( Vector3D const& ) = 0;
    virtual void rotate( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};

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

    virtual ~Circle() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;

private:
    double radius;
};

An Object-Oriented Solution
class Shape
{
  public:
    Shape() = default;
    virtual ~Shape() = default;

    virtual void translate( Vector3D const& ) = 0;
    virtual void rotate( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};

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

    virtual ~Circle() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;

  private:
    double radius;
};
An Object-Oriented Solution

```cpp
virtual void translate(const Vector3D& v) = 0;
virtual void rotate(const Quaternion& q) = 0;
virtual void draw() const = 0;

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

  virtual ~Circle() = default;

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

  void translate(const Vector3D& v) override;
  void rotate(const Quaternion& q) override;
  void draw() const override;

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

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

  // ... }
```
An Object-Oriented Solution

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

class Square : public Shape
{
public:
    explicit Square( double s )
        : side{ s }
        , // ... Remaining data members
    {}
    virtual ~Square() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;

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

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
    for( auto const& s : shapes )
    {
        s->draw()
    }
An Object-Oriented Solution

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

virtual ~Square() = default;

double getSide() const noexcept;

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

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

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

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

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

    // Drawing all shapes
    draw( shapes );
}
```
void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
    for( auto const& s : shapes )
    {
        s->draw()
    }
}

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

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

    // Drawing all shapes
    draw( shapes );
}
Design Evaluation

<table>
<thead>
<tr>
<th>Enum</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th>Enum</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOP</td>
<td>8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 = very bad, ..., 9 = very good</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>00</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>00</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

*1 = very bad, ..., 9 = very good*
Performance Results

GCC-9.2

Clang-9

Enum

OO

Relative Runtime

MacBook Pro 13-inch, MacOS 10.14.6 (Mojave), Intel Core i7, 2.7Ghz, 16 GByte, Compilation flags: -std=c++17 -O3 -DNDEBUG

100 random shapes, 2.5M updates each, normalized to the enum solution
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>00</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
The Visitor Design Pattern

Client

Visitor

virtual visit( ConcreteElementA ) = 0
virtual visit( ConcreteElementB ) = 0

ConcreteVisitorA

virtual visit( ConcreteElementA )
virtual visit( ConcreteElementB )

ConcreteVisitorB

virtual visit( ConcreteElementA )
virtual visit( ConcreteElementB )

ConcreteElementA

virtual accept( Visitor v )
operationA()

v->visit( this )

ConcreteElementB

virtual accept( Visitor v )
operationB()

v->visit( this )

ObjectStructure
The Visitor Design Pattern

Client

Visitor

virtual visit( Circle ) = 0
virtual visit( Square ) = 0

ConcreteVisitorA

virtual visit( Circle )
virtual visit( Square )

ConcreteVisitorB

virtual visit( Circle )
virtual visit( Square )

ObjectStructure

Shape

virtual accept( Visitor ) = 0

Circle

virtual accept( Visitor v )
operationA()

v->visit( this )

Square

virtual accept( Visitor v )
operationB()

v->visit( this )
The Visitor Design Pattern

Client

Visitor

virtual visit( Circle ) = 0
virtual visit( Square ) = 0

ObjectStructure

Visitor

Shape

virtual accept( Visitor ) = 0

Rotate

virtual visit( Circle )
virtual visit( Square )

Draw

virtual visit( Circle )
virtual visit( Square )

Circle

virtual accept( Visitor v )
operationA()

Square

virtual accept( Visitor v )
operationB()

v->visit( this )

v->visit( this )
class Circle;
class Square;

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

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

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

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

class Circle : public Shape
{
public:
    explicit Circle( double rad ) :
        radius{ rad }
    , // ... Remaining data members
{}
A Visitor-Based Solution

class Circle;
class Square;

class Visitor
{
  public:
    virtual ~Visitor() = default;
    virtual void visit( Circle const& ) const = 0;
    virtual void visit( Square const& ) const = 0;
};

class Shape
{
  public:
    Shape() = default;
    virtual ~Shape() = default;
    virtual void accept( Visitor const& ) = 0;
};

class Circle : public Shape
{
  public:
    explicit Circle( double rad )
      : radius{ rad }
        , // ... Remaining data members
    {}
A Visitor-Based Solution

class Circle;
class Square;

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

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

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

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

class Circle : public Shape
{
    public:
        explicit Circle( double rad )
            : radius{ rad }
            , // ... Remaining data members
    }
A Visitor-Based Solution

```cpp
virtual ~Shape() = default;

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

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

virtual ~Circle() = default;

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

void accept( Visitor const& ) override;

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

class Square : public Shape
{
public:
  explicit Square( double s )
    : side{ s }
  , // ... Remaining data members
  {}
A Visitor-Based Solution

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

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

    virtual ~Square() = default;

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

    void accept( Visitor const& ) override;

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

class Draw : public Visitor
{
public:
    void visit( Circle const& ) const override;
    void visit( Square const& ) const override;
};
virtual ~Square() = default;

double getSide() const noexcept;

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

void accept( Visitor const& ) override;

private:
    double side;

    // ... Remaining data members

};

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

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

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

    // Creating some shapes
    Shapes shapes;
    shapes.push_back( std::make_unique<Circle>( 2.0 ) );
    shapes.push_back( std::make_unique<Square>( 1.5 ) );
void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
    for( auto const& s : shapes )
    {
        s->accept( Draw{} )
    }
}

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

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

    // Drawing all shapes
    draw( shapes );
}
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enum</strong></td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td><strong>OO</strong></td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td><strong>Visitor</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enum</strong></td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td><strong>OO</strong></td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td><strong>Visitor</strong></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
Performance Results

Enum
OO
Visitor

Relative Runtime

GCC-9.2
Clang-9

MacBook Pro 13-inch, MacOS 10.14.6 (Mojave), Intel Core i7, 2,7Ghz, 16 GByte, Compilation flags: -std=c++17 -O3 -DNDEBUG
100 random shapes, 2.5M updates each, normalized to the enum solution
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
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;
};
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;
}
A “Modern C++” Solution

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

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

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

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

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

using Shape = std::variant<Circle, Square>;
```
public:
explicit Square( double s )
: side{ s }
{ }

double getSide() const noexcept;

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

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

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

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

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

    // Creating some shapes
    Shapes shapes;
    shapes.push_back( Circle{ 2.0 } );
```cpp
using Shape = std::variant<Circle, Square>;

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

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

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

    // Drawing all shapes
    draw( shapes );
}
```
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enum</strong></td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td><strong>OO</strong></td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td><strong>Visitor</strong></td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><strong>std::variant</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
# Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>std::variant</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>std::variant</td>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>std::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>std::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
Performance Results

Relative Runtime

GCC-9.2  Clang-9

Enum  OO  Visitor  std::variant

MacBook Pro 13-inch, MacOS 10.14.6 (Mojave), Intel Core i7, 2.7Ghz, 16 GByte, Compilation flags: -std=c++17 -O3 -DNDEBUG

100 random shapes, 2.5M updates each, normalized to the enum solution
Performance Analysis

template< typename T, typename V >
constexpr auto make_func() {
    return + []( const char* b, V v ) {
        const auto& x = *reinterpret_cast<const T*>(b);
        v(x);
    };
}

template< typename... Ts, typename V >
void foo( std::size_t i, const char* b, V v ) {
    static constexpr std::array<void(*)(const char*, V v ), sizeof...(Ts)> table = { make_func<Ts,V>()... };
    table[i](b,v);
}
mpark::variant

MPark.Variant

Introduction

MPark.Variant is an implementation of C++17 std::variant for C++11/14/17.

- Based on my implementation of std::variant for libc++
- Continuously tested against libc++'s std::variant test suite.
Performance Results

<table>
<thead>
<tr>
<th>Method</th>
<th>GCC-9.2</th>
<th>Clang-9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visitor</td>
<td></td>
<td>4.5</td>
</tr>
<tr>
<td>std::variant</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>mpark::variant</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MacBook Pro 13-inch, MacOS 10.14.6 (Mojave), Intel Core i7, 2.7Ghz, 16 GByte, Compilation flags: -std=c++17 -O3 -DNDEBUG
100 random shapes, 2.5M updates each, normalized to the enum solution
Performance Analysis

// ...

switch (v.index()) {
    case B + 0: return MPARK_DISPATCH(B + 0);
    case B + 1: return MPARK_DISPATCH(B + 1);
    case B + 2: return MPARK_DISPATCH(B + 2);
    case B + 3: return MPARK_DISPATCH(B + 3);
    case B + 4: return MPARK_DISPATCH(B + 4);
    case B + 5: return MPARK_DISPATCH(B + 5);
    case B + 6: return MPARK_DISPATCH(B + 6);
    case B + 7: return MPARK_DISPATCH(B + 7);
    case B + 8: return MPARK_DISPATCH(B + 8);
    case B + 9: return MPARK_DISPATCH(B + 9);
    case B + 10: return MPARK_DISPATCH(B + 10);
    case B + 11: return MPARK_DISPATCH(B + 11);
    case B + 12: return MPARK_DISPATCH(B + 12);
    case B + 13: return MPARK_DISPATCH(B + 13);
    case B + 14: return MPARK_DISPATCH(B + 14);
    case B + 15: return MPARK_DISPATCH(B + 15);
    case B + 16: return MPARK_DISPATCH(B + 16);
    case B + 17: return MPARK_DISPATCH(B + 17);
    case B + 18: return MPARK_DISPATCH(B + 18);
    case B + 19: return MPARK_DISPATCH(B + 19);
    case B + 20: return MPARK_DISPATCH(B + 20);
    case B + 21: return MPARK_DISPATCH(B + 21);
    case B + 22: return MPARK_DISPATCH(B + 22);
    case B + 23: return MPARK_DISPATCH(B + 23);
    case B + 24: return MPARK_DISPATCH(B + 24);
    case B + 25: return MPARK_DISPATCH(B + 25);
    case B + 26: return MPARK_DISPATCH(B + 26);
    case B + 27: return MPARK_DISPATCH(B + 27);
}
Performance Analysis

case B + 8: return MPARK_DISPATCH(B + 8);
case B + 9: return MPARK_DISPATCH(B + 9);
case B + 10: return MPARK_DISPATCH(B + 10);
case B + 11: return MPARK_DISPATCH(B + 11);
case B + 12: return MPARK_DISPATCH(B + 12);
case B + 13: return MPARK_DISPATCH(B + 13);
case B + 14: return MPARK_DISPATCH(B + 14);
case B + 15: return MPARK_DISPATCH(B + 15);
case B + 16: return MPARK_DISPATCH(B + 16);
case B + 17: return MPARK_DISPATCH(B + 17);
case B + 18: return MPARK_DISPATCH(B + 18);
case B + 19: return MPARK_DISPATCH(B + 19);
case B + 20: return MPARK_DISPATCH(B + 20);
case B + 21: return MPARK_DISPATCH(B + 21);
case B + 22: return MPARK_DISPATCH(B + 22);
case B + 23: return MPARK_DISPATCH(B + 23);
case B + 24: return MPARK_DISPATCH(B + 24);
case B + 25: return MPARK_DISPATCH(B + 25);
case B + 26: return MPARK_DISPATCH(B + 26);
case B + 27: return MPARK_DISPATCH(B + 27);
case B + 28: return MPARK_DISPATCH(B + 28);
case B + 29: return MPARK_DISPATCH(B + 29);
case B + 30: return MPARK_DISPATCH(B + 30);
case B + 31: return MPARK_DISPATCH(B + 31);
default: return MPARK_DEFAULT(B + 32);
}

// ...
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
What kind of programming language is C++?
### std::variant<Type...> VS INHERITANCE

<table>
<thead>
<tr>
<th>INHERITANCE</th>
<th>VARIANT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open/Closed to new alternatives</td>
<td>Closed to new alternatives</td>
</tr>
<tr>
<td>Closed to new operations</td>
<td>Open to new operations</td>
</tr>
<tr>
<td>Multi-level</td>
<td>Single level</td>
</tr>
<tr>
<td>OO</td>
<td>Functional</td>
</tr>
<tr>
<td>Pointer semantics</td>
<td>Value semantics</td>
</tr>
<tr>
<td>Design forced by the implementation details</td>
<td>Many design choices possible</td>
</tr>
<tr>
<td>Forces dynamic memory allocations</td>
<td>No dynamic memory allocations</td>
</tr>
</tbody>
</table>
C++’s evolution priorities

Historical

Add things
Fix things
Simplify

A future worth considering?

De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable ("Simplifying C++" #6 of N)
What Do We Mean By Great?

- Easier to write?
- Easier to maintain?
- More optimizable?

We’ll touch a bit on each of these things, but focus on the parts that let the optimizer work.

This is not a “Best Practices” talk per se, it’s a talk to make you think more about your classes.
The Strategy Design Pattern

Context
context()

Strategy
virtual algorithm() = 0

ConcreteStrategyA
virtual algorithm()

ConcreteStrategyB
virtual algorithm()
The Strategy Design Pattern

Shape

draw()

DrawStrategy

virtual draw() = 0

DrawStrategyA

virtual draw()

DrawStrategyB

virtual draw()
A Strategy-Based Solution

```cpp
class Circle;
class Square;

class DrawStrategy
{
public:
    virtual ~DrawStrategy() {}
    virtual void draw(const Circle& circle) const = 0;
    virtual void draw(const Square& square) const = 0;
};

class Shape
{
public:
    Shape() = default;
    virtual ~Shape() = default;
    virtual void translate(Vector3D const& ) = 0;
    virtual void rotate(Quaternion const& ) = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle(double rad, std::unique_ptr<DrawStrategy> ds)
        : radius{ rad }
        // Remaining data members
    
    virtual void translate(Vector3D const& ) override;
    virtual void rotate(Quaternion const& ) override;
    virtual void draw() const override;
};
```

class Circle;
class Square;

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

  virtual void draw( const Circle& circle ) const = 0;
  virtual void draw( const Square& square ) const = 0;
};

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

  virtual void translate( Vector3D const& ) = 0;
  virtual void rotate( Quaternion const& ) = 0;
  virtual void draw() const = 0;
};

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

  virtual void draw() const override;
};
A Strategy-Based Solution

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

    virtual void draw( const Circle& circle ) const = 0;
    virtual void draw( const Square& square ) const = 0;
};

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

    virtual void translate( Vector3D const& ) = 0;
    virtual void rotate( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};

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

    virtual ~Circle() = default;

A Strategy-Based Solution

```cpp
virtual void draw() const = 0;
};

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

    virtual ~Circle() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;

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

class Square : public Shape
{
    public:
        explicit Square( double s, std::unique_ptr<DrawStrategy> ds )
```
A Strategy-Based Solution

```cpp
class Square : public Shape
{
public:
    explicit Square( double s, std::unique_ptr<DrawStrategy> ds )
        : side{ s }
        , // Remaining data members
        , drawing{ std::move(ds) }
    {}

    virtual ~Square() = default;

double getSide() const noexcept;
// Remaining functions...

void translate( Vector3D const& ) override;
void rotate( Quaternion const& ) override;
void draw() const override;

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

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
    for( auto const& s : shapes )
    {
        s->draw();
    }
```
A Strategy-Based Solution

```cpp
void translate( Vector3D const& ) override;
void rotate( Quaternion const& ) override;
void draw() const override;

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

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

class OpenGLStrategy : public DrawStrategy
{
public:
    virtual ~OpenGLStrategy() {}

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

int main()
{
    using Shapes = std::vector<std::unique_ptr<Shape>>;
    using
```
A Strategy-Based Solution

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

class OpenGLStrategy : public DrawStrategy
{
    public:
        virtual ~OpenGLStrategy() {};

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

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

    // Creating some shapes
    Shapes shapes;
    shapes.push_back( std::make_unique<Circle>( 2.0,
                                                std::make_unique<OpenGLStrategy>() ) );
    shapes.push_back( std::make_unique<Square>( 1.5,
                                                std::make_unique<OpenGLStrategy>() ) );
    shapes.push_back( std::make_unique<Circle>( 4.2,
                                                std::make_unique<OpenGLStrategy>() ) );
```
A Strategy-Based Solution

```cpp
class OpenGLStrategy : public DrawStrategy
{
public:
    virtual ~OpenGLStrategy() {}

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

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

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

    // Drawing all shapes
    draw(shapes);
}
```
A Strategy-Based Solution

class Circle;
class Square;

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

    virtual void draw( const Circle& circle ) const = 0;
    virtual void draw( const Square& square ) const = 0;
};

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

    virtual void translate( Vector3D const& ) = 0;
    virtual void rotate( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle( double rad, std::unique_ptr<DrawStrategy> ds )
        : radius{ rad }
    {
        // Remaining data members
    }
};
A Strategy-Based Solution

class Circle;
class Square;

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

    virtual void draw( const Circle& circle ) const = 0;
};

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

    virtual void draw( const Square& square ) const = 0;
};

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

    virtual void translate( Vector3D const& ) = 0;
    virtual void rotate( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};
class Shape
{
public:
    Shape() = default;
    virtual ~Shape() = default;
    virtual void translate( Vector3D const& ) = 0;
    virtual void rotate( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};

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

    virtual ~Circle() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;

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

class Square : public Shape
{
public:
    // ...

    virtual void rotate( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};
A Strategy-Based Solution

```cpp
std::unique_ptr<DrawCircleStrategy> drawing;
};

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

    virtual ~Square() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;

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

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
    for( auto const& s : shapes )
    {
        s->draw();
    }
```
A Strategy-Based Solution

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

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

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

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

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

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

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

```cpp
public:
    virtual ~OpenGLStrategy() {}

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

class OpenGLSquareStrategy : public DrawSquareStrategy
{
    public:
        virtual ~OpenGLStrategy() {}
        void draw( Square const& square ) const override;
};
```

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

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

    // Drawing all shapes
    draw( shapes );
}
```
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
# Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th>Method</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
Performance Results

Enum
OO
Visitor
std::variant
mpark::variant
Strategy

Relative Runtime

GCC-9.2
Clang-9

MacBook Pro 13-inch, MacOS 10.14.6 (Mojave), Intel Core i7, 2.7Ghz, 16 GByte, Compilation flags: -std=c++17 -O3 -DNDEBUG
100 random shapes, 2.5M updates each, normalized to the enum solution
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
A “Modern C++” Solution

```cpp
class Circle;
class Square;

using DrawCircleStrategy = std::function<void(Circle const&)>;
using DrawSquareStrategy = std::function<void(Square const&)>;

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

    virtual void translate(Vector3D const&) = 0;
    virtual void rotate(Quaternion const&) = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle(double rad, DrawCircleStrategy ds)
        : radius{ rad }, // ... Remaining data members
          drawing{ std::move(ds) }
    {}

    virtual ~Circle() = default;

double setRadius() const noexcept;
```
A “Modern C++” Solution

class Circle;
class Square;

using DrawCircleStrategy = std::function<void(Circle const&)>;
using DrawSquareStrategy = std::function<void(Square const&)>;

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

        virtual void translate( Vector3D const& ) = 0;
        virtual void rotate( Quaternion const& ) = 0;
        virtual void draw() const = 0;
};

class Circle : public Shape
{
    public:
        explicit Circle( double rad, DrawCircleStrategy ds )
            : radius{ rad }
            , // ... Remaining data members
            , drawing{ std::move(ds) }
            {}

        virtual ~Circle() = default;

        double setRadius() const noexcept;

        // ...
A “Modern C++” Solution

class Circle;
class Square;

using DrawCircleStrategy = std::function<void(Circle const&)>;
using DrawSquareStrategy = std::function<void(Square const&)>;

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

        virtual void translate( Vector3D const& ) = 0;
        virtual void rotate( Quaternion const& ) = 0;
        virtual void draw() const = 0;
};

class Circle : public Shape
{
    public:
        explicit Circle( double rad, DrawCircleStrategy ds )
            : radius{ rad }
            , // ... Remaining data members
            , drawing{ std::move(ds) }
        {}

        virtual ~Circle() = default;

double setRadius() const noexcept;

A “Modern C++” Solution

```cpp
class Shape
{
public:
    Shape() = default;
    virtual ~Shape() = default;
    virtual void translate( Vector3D const& ) = 0;
    virtual void rotate( Quaternion const& ) = 0;
    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    explicit Circle( double rad, DrawCircleStrategy ds )
        : radius{ rad }
        , // ... Remaining data members
        , drawing{ std::move(ds) }
    {}

    virtual ~Circle() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;

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

class Square : public Shape
{
public:
};
```
A “Modern C++” Solution

```cpp
void translate( Vector3D const& ) override;
void rotate( Quaternion const& ) override;
void draw() const override;
private:
  double radius;
  // ... Remaining data members
  DrawCircleStrategy drawing;
};

class Square : public Shape
{
  public:
  explicit Square( double s, DrawSquareStrategy ds )
      : side{ s }
      , // ... Remaining data members
      , drawing{ std::move(ds) }
  {}

  virtual ~Square() = default;

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

  void translate( Vector3D const& ) override;
  void rotate( Quaternion const& ) override;
  void draw() const override;

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

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
  for( auto const& s : shapes )
  {
  
  
```

```
```
A “Modern C++” Solution

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

class OpenGLCircleStrategy
{
    public:
        void operator()( Circle const& circle ) const;
    };

class OpenGLSquareStrategy
{
    public:
        void operator()( Square const& square ) const;
    };

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

    // Creating some shapes
    Shapes shapes;
    shapes.push_back( std::make_unique<Circle>( 2.0 , OpenGLCircleStrategy{} ) );
    shapes.push_back( std::make_unique<Square>( 1.5 , OpenGLSquareStrategy{} ) );
    shapes.push_back( std::make_unique<Circle>( 4.2 ) );
}
```
int main()
{
    using Shapes = std::vector<std::unique_ptr<Shape>>;

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

    // Drawing all shapes
    draw( shapes );
}
A “Modern C++” Solution

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

class OpenGLCircleStrategy
{
    public:
        void operator()( Circle const& circle ) const;
};

class OpenGLSquareStrategy
{
    public:
        void operator()( Square const& square ) const;
};

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

    // Creating some shapes
    Shapes shapes;
    shapes.push_back( std::make_unique<Circle>( 2.0 , OpenGLCircleStrategy{} ) );
    shapes.push_back( std::make_unique<Square>( 1.5 , OpenGLSquareStrategy{} ) );
    shapes.push_back( std::make_unique<Circle>( 4.2 ) );
}
```
A “Modern C++” Solution

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

void draw( Circle const& circle ) const;
void draw( Square const& square ) const;

struct Draw
{
    template< typename T >
    void operator()( T const& drawable ) const {
        draw( drawable );
    }
};

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

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

    // Drawing all shapes
    draw( shapes );
}
```
A “Modern C++” Solution

```cpp
struct Draw
{
    template<typename T>
    void operator()(T const& drawable) const
    {
        draw(drawable);
    }
};

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

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

    // Drawing all shapes
    draw(shapes);
}
```
void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
    for( auto const& s : shapes )
    {
        s->draw()
    }
}

void draw( Circle const& circle ) const;
void draw( Square const& square ) const;

struct Draw
{
    template< typename T >
    void operator()( T const& drawable ) const {
        draw( drawable );
    }
};

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

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

    // Drawing all shapes
}
A “Modern C++” Solution

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

void draw( Circle const& circle ) const;
void draw( Square const& square ) const;

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

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

    // Drawing all shapes
}
```
A “Modern C++” Solution

class Circle;
class Square;

using DrawCircleStrategy = std::function<void(Circle const&)>;
using DrawSquareStrategy = std::function<void(Square const&)>;

struct Draw
{
    template< typename T >
    void operator()( T const& drawable ) const {
        draw( drawable );
    }
};

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

        virtual void translate( Vector3D const& ) = 0;
        virtual void rotate( Quaternion const& ) = 0;
        virtual void draw() const = 0;
};

class Circle : public Shape
{
    public:
        explicit Circle( double rad, DrawCircleStrategy ds = Draw{} )
class Circle : public Shape
{
public:
    explicit Circle( double rad, DrawCircleStrategy ds = Draw{} )
        : radius{ rad }
        , // ... Remaining data members
        drawing{ std::move(ds) }
    {}

    virtual ~Circle() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;

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

class Square : public Shape
{
public:
    explicit Square( double s, DrawSquareStrategy ds = Draw{} )
    { }

    virtual ~Square() = default;

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

    void translate( Vector3D const& ) override;
    void rotate( Quaternion const& ) override;
    void draw() const override;

private:
    double sideLength;
    // ... Remaining data members
    DrawSquareStrategy drawing;
};
A “Modern C++” Solution

```cpp
class Square : public Shape
{
    public:
        explicit Square( double s, const DrawSquareStrategy ds = Draw{} )
            : side{ s }
            , // ... Remaining data members
            , drawing{ std::move(ds) }
            {}

    virtual ~Square() = default;

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

    void translate( const Vector3D& ) override;
    void rotate( const Quaternion& ) override;
    void draw() const override;

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

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

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

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

    // Drawing all shapes
    draw( shapes );
}
# Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*1 = very bad, ..., 9 = very good*
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enum</strong></td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td><strong>OO</strong></td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td><strong>Visitor</strong></td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><strong>mpark::variant</strong></td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td><strong>Strategy</strong></td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td><strong>std::function</strong></td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ... , 9 = very good
Performance Results

MacBook Pro 13-inch, MacOS 10.14.6 (Mojave), Intel Core i7, 2,7Ghz, 16 GByte, Compilation flags: -std=c++17 -O3 -DNDEBUG
100 random shapes, 2.5M updates each, normalized to the enum solution
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
A Type-Erased Solution

class Circle
{
    public:
        explicit Circle( double rad )
            : radius{ rad }
            , // ... Remaining data members
            {}
        double getRadius() const noexcept;
        // ... getCenter(), getRotation(), ...

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

void translate( Circle const&, Vector3D const& );
void rotate( Circle const&, Quaternion const& );
void draw( Circle const& );

class Square
{
    public:
        explicit Square( double s )
            : side{ s }
            , // ... Remaining data members
            {}
        double getSide() const noexcept;
        // ... Remaining data members
};
A Type-Erased Solution

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

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

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

void translate( Circle const&, Vector3D const& );
void rotate( Circle const&, Quaternion const& );
void draw( Circle const& );

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

    double getSide() const noexcept;
A Type-Erased Solution

```cpp
void translate( Circle const&, Vector3D const& );
void rotate( Circle const&, Quaternion const& );
void draw( Circle const& );

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

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

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

void translate( Square const&, Vector3D const& );
void rotate( Square const&, Quaternion const& );
void draw( Square const& );

class Shape
{
private:
    struct Concept
    {
        virtual ~Concept() {};
        virtual void do_translate( Vector3D const& ) const = 0;
    }

    // ... Remaining data members
};
```
A Type-Erased Solution

void translate( Square const&, Vector3D const& );
void rotate( Square const&, Quaternion const& );
void draw( Square const& );

class Shape
{
  private:
    struct Concept
    {
      virtual ~Concept() {}
      virtual void do_translate( Vector3D const& v ) const = 0;
      virtual void do_rotate( Quaternion const& q ) const = 0;
      virtual void do_draw() const = 0;
      // ...
    };

template< typename T >
struct Model : Concept
{
  Model( T const& value )
    : object{ value }
  {}

  void do_translate( Vector3D const& v ) const override
  {
    translate( object, v );
  }

  void do_rotate( Quaternion const& q ) const override
  {
    rotate( object, q );
  }
}
A Type-Erased Solution

```cpp
};

template< typename T >
struct Model : Concept
{
    Model( T const& value )
    : object{ value }
    {}

    void do_translate( Vector3D const& v ) const override
    {
        translate( object, v );
    }

    void do_rotate( Quaternion const& q ) const override
    {
        rotate( object, q );
    }

    void do_draw() const override
    {
        draw( object );
    }

    // ...

    T object;
};

std::unique_ptr<Concept> pimpl;

friend void translate( Shape& shape, Vector3D const& v )
{
```

A Type-Erased Solution

void translate( Square const&, Vector3D const& );
void rotate( Square const&, Quaternion const& );
void draw( Square const& );

class Shape
{
    private:
        struct Concept
        {
            virtual ~Concept() {};
            virtual void do_translate( Vector3D const& v ) const = 0;
            virtual void do_rotate( Quaternion const& q ) const = 0;
            virtual void do_draw() const = 0;
            // ...
        };

        template< typename T >
        struct Model : Concept
        {
            Model( T const& value )
                : object{ value }
            {}

            void do_translate( Vector3D const& v ) const override
            {
                translate( object, v );
            }

            void do_rotate( Quaternion const& q ) const override
            {
                rotate( object, q );
            }
        };

    private:
        double side;
        // ... Remaining data members
};
A Type-Erased Solution

// ...

T object;
};

std::unique_ptr<Concept> pimpl;

friend void translate( Shape& shape, Vector3D const& v )
{
    shape.pimpl->do_translate( v );
}

friend void rotate( Shape& shape, Quaternion const& q )
{
    shape.pimpl->do_rotate( q );
}

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

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

    // Special member functions
    Shape( Shape const& s );
    Shape( Shape&& s );
    Shape& operator=( Shape const& s );
    Shape& operator=( Shape&& s );
A Type-Erased Solution

    // ...
    
    T object;
    }

    std::unique_ptr<Concept> pimpl;

friend void translate( Shape& shape, Vector3D const& v )
{
    shape.pimpl->do_translate( v );
}

friend void rotate( Shape& shape, Quaternion const& q )
{
    shape.pimpl->do_rotate( q );
}

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

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

    // Special member functions
    Shape( Shape const& s );
    Shape( Shape&& s );
    Shape& operator=( Shape const& s );
    Shape& operator=( Shape&& s );
A Type-Erased Solution

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

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

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

    // ...
};

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

int main()
{
    // Usage...
}
```
A Type-Erased Solution

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

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

    // ...
};

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

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

    // Creating some shapes
    Shapes shapes;
    shapes.push_back( Circle( 2.0 ) );
    shapes.push_back( Square( 1.5 ) );
    shapes.push_back( Circle( 4.2 ) );

    // Drawing all shapes
    draw( shapes );
}
```
A Type-Erased Solution

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

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

    // Creating some shapes
    Shapes shapes;
    shapes.push_back( Circle( 2.0 ) );
    shapes.push_back( Square( 1.5 ) );
    shapes.push_back( Circle( 4.2 ) );

    // Drawing all shapes
    draw( shapes );
}
```
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td>9</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th>Method</th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
# Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
Performance Results

- Enum
- OO
- Visitor
- std::variant
- mpark::variant
- Strategy
- std::function
- Type Erasure

Relative Runtime

MacBook Pro 13-inch, MacOS 10.14.6 (Mojave), Intel Core i7, 2.7Ghz, 16 GByte, Compilation flags: -std=c++17 -O3 -DNDEBUG
100 random shapes, 2.5M updates each, normalized to the enum solution
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
# Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>OO</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Strategy</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
# Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
# Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
## Design Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Addition of shapes (OCP)</th>
<th>Addition of operations (OCP)</th>
<th>Separation of Concerns (SRP)</th>
<th>Ease of Use</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpark::variant</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>std::function</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Type Erasure</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

1 = very bad, ..., 9 = very good
Procedural programming
Object-oriented programming
Functional programming
Generic programming
Multiparadigm programming
No-paradigm programming
C++20: C++ at 40
stability and evolution

Bjarne Stroustrup
Morgan Stanley, Columbia University
www.stroustrup.com
Tell Them ...

- Understand the virtues of “Modern C++”
  - Reduce the use of pointers and inheritance hierarchies
  - Prefer value semantics
  - Keep your code simple

- Learn about the different programming paradigms
  - Learn about their advantages and weaknesses
  - Pick the good ideas

- There is no “one-fits-all” solution

- Don’t shackle yourself with thinking about paradigms ...
Embrace No-Paradigm Programming

Klaus Iglberger, CppEurope 2020
klaus.iglberger@gmx.de