Back to Basics:
Designing Classes (part 1 of 2)

KLAUS IGLBERGER
C++ Trainer/Consultant

Author of the blaze C++ math library

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

Chair of the CppCon B2B and SD tracks

Email: klaus.iglberger@gmx.de

Klaus Iglberger
Content

Back to Basics: Class Design (Part 1)
- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)
- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility
The Challenge of Class Design

Back to Basics: Class Design (Part 1)
- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)
- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility
What is the root source of all problems in software development?

Change
The Challenge of Class Design

The truth in our industry:

Software must be adaptable to frequent changes
The Challenge of Class Design

The truth in our industry:

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

Dependencies
The Challenge of Class Design

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

**Guideline:** Design classes for easy change.

**Guideline:** Design classes for easy extensions.
Design Guidelines

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility
Design for Readability

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility
Guideline: Spent time to find good names for all entities.

```cpp
template<
class T,
    std::size_t N
> struct array;
```

What does 'N' represent?
Guideline: Spent time to find good names for all entities.

```cpp
template<typename T, std::size_t Size>
struct array;
```

Now it’s clear!
Guideline: Spent time to find good names for all entities.

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

Container or numerical vector?
Design for Readability

**Guideline:** Spent time to find good names for all entities.

```cpp
#include <memory>

// ... (Code snippet)

// ... (Code snippet)
```

Action or query?
Design for Readability

Naming is Hard: Let's Do Better

Kate Gregory
kate@gregcons.com
www.gregcons.com/kateblog
@gregcons
Design for Readability

**Guideline:** Spent time to find good names for all entities.

"Naming requires Empathy."
(Kate Gregory, Naming is Hard: Let’s Do Better, CppCon 2019)
Design for Change and Extension

Back to Basics: Class Design (Part 1)
- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)
- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility
"I’m tired of this example, but I don’t know any better one."

(Lukas Bergdoll, MUC++ organizer)
Designing the Shape Hierarchy

```cpp
class Shape {
public:
    virtual void draw() = 0;
    virtual void serialize() = 0;
};

class Circle : public Shape {
public:
    virtual void draw() = 0;
};

class Square : public Shape {
public:
    virtual void draw() = 0;
};

class OpenGLCircle : public Circle {
public:
    virtual void draw() = 0;
};

class MetalCircle : public Circle {
public:
    virtual void draw() = 0;
};

class OpenGLSquare : public Square {
public:
    virtual void draw() = 0;
};

class MetalSquare : public Square {
public:
    virtual void draw() = 0;
};

class OpenGLLittleEndianSquare : public OpenGLSquare {
public:
    virtual void serialize() = 0;
};

class OpenGLBigEndianSquare : public OpenGLSquare {
public:
    virtual void serialize() = 0;
};
```
Designing the Shape Hierarchy

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

Circle
- virtual draw()
- virtual serialize()

Square
- virtual draw()

OpenGLSquare
- virtual serialize()

MetalSquare
- virtual draw()

OpenGLLittleEndianCircle
- virtual draw()
- virtual serialize()

OpenGLBigEndianCircle
- virtual draw()
- virtual serialize()

MetalLittleEndianCircle
- virtual draw()
- virtual serialize()

MetalBigEndianCircle
- virtual draw()
- virtual serialize()

OpenGLLittleEndianSquare
- virtual serialize()

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

- ... many derived classes;
- ... ridiculous class names;
- ... deep inheritance hierarchies;
- ... duplication between similar implementations (DRY);
- ... (almost) impossible extensions (OCP);
- ... impeded maintenance.
Designing the Shape Hierarchy

Guideline: Resist the urge to put everything into one class. Separate concerns!

Guideline: If you use OO programming, use it properly.
Guideline: Design classes for easy change.

Guideline: Design classes for easy extensions.
Designing the Shape Hierarchy
”Inheritance is Rarely the Answer.
Delegate to Services: Has-A Trumps Is-A.”
(Andrew Hunt, David Thomas, The Pragmatic Programmer)
The Solution: Design Principles and Patterns
The Solution: Design Principles and Patterns

- Single-Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Don’t Repeat Yourself (DRY)
The Single-Responsibility Principle (SRP)

“Everything should do just one thing.”

(Common Knowledge?)
The Single-Responsibility Principle (SRP)

“The Single-Responsibility Principle advices to separate concerns to isolate and simplify change.”

(Klaus Iglberger)

The SRP is also known as

- Separation of Concerns
- High cohesion / low coupling
- Orthogonality
The Open-Closed Principle (OCP)

“The Open-Closed Principle advices to prefer design that simplifies the extension by types or operations.”

(Klaus Iglberger)
Don’t Repeat Yourself (DRY)

“The DRY Principle advises to reduce duplication in order to simplify change.”

(Klaus Iglberger)
The Solution: Design Principles and Patterns

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

A design pattern ...

- ... has a name;
- ... carries an intent;
- ... aims at reducing dependencies;
- ... provides some sort of abstraction;
- ... has proven to work over the years.
The Strategy Design Pattern

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

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

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

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

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

class Circle;

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

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

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

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

class Circle;

class DrawCircleStrategy {
    public:
        virtual ~DrawCircleStrategy() {}
        virtual void draw( Circle const& circle, /*...*/ ) const = 0;
    }

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

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

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

class Circle;

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

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

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

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

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

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

    void serialize( /*...*/ ) const override;
    // ...

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

class Square;
```

Dependency Injection
A Strategy-Based Solution

// ...

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

class Square;

class DrawSquareStrategy
{
  public:
    virtual ~DrawSquareStrategy() {}
    virtual void draw(Square const& square, /*...*/ ) const = 0;
};

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

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

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

    void serialize( /*...*/ ) const override;
    // ...

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

class OpenGLCircleStrategy : public DrawCircleStrategy
{
A Strategy-Based Solution

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

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

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

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

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

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

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

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

    void draw( Square const& square ) const override; // Draw a square
};

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

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

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

By means of the Strategy design pattern we have ...

- ... extracted implementation details (SRP);
- ... created the opportunity for easy change;
- ... created the opportunity for easy extension (OCP);
- ... reduced duplication (DRY);
- ... limited the depth of the inheritance hierarchy;
- ... simplified maintainance.
A Strategy-Based Solution — Guidelines

**Guideline:** Design classes for easy change.

**Guideline:** Design classes for easy extensions.

**Guideline:** Don’t guess! If you expect change, prefer design that makes this change easy. If you don’t expect any change, learn from the next change.
A Strategy-Based Solution

The guidelines make sense, but still you complain …

"That’s the style of the 90s and early 2000s, not Modern C++!”

(You)

And you are correct. Today we favor a value-semantics style…
A Strategy-Based Solution

class Circle;

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

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

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

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

  void serialize( /*...*/ ) const override;
    // ...

private:
  double radius;
    // ... Remaining data members
  DrawCircleStrategy drawing;
};
A Strategy-Based Solution

```cpp
template< typename DrawStrategy >
class Circle : public Shape
{
public:
    Circle( double rad )
        : radius{ rad }
        , // ... Remaining data members
{}

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

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

void serialize( /*...*/ ) const override;
// ... 

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

It’s still the same intent: separation of concerns (SRP)
A Strategy-Based Solution — Guidelines

Guideline: Design classes for easy change.

Guideline: Design classes for easy extensions.
Our Second Toy Problem: Persistence Systems

class PersistenceInterface
{
public:
    PersistenceInterface();
    virtual ~PersistenceInterface();

    virtual bool write( const Blob& blob ) = 0;
    virtual bool write( const Blob& blob, WriteCallback callback ) = 0;
    virtual bool read ( Blob& blob, uint timeout ) = 0;
    virtual bool read ( Blob& blob, ReadCallback callback, uint timeout ) = 0;
    // ...
};
Our Second Toy Problem: Persistence Systems

```cpp
class PersistenceInterface
{
public:
    PersistenceInterface();

    virtual ~PersistenceInterface();

    virtual bool write( const Blob& blob ) = 0;
    virtual bool write( const Blob& blob, WriteCallback callback ) = 0;
    virtual bool read ( Blob& blob, uint timeout ) = 0;
    virtual bool read ( Blob& blob, ReadCallback callback, uint timeout ) = 0;
    // ...
};
```

The virtual functions may pose a problem in the future ...

- ... because they represent the interface to callers;
- ... because they represent the interface for deriving classes;
- ... don’t separate concerns;
- ... potentially introduces a lot of duplication;
- ... make changes harder (and sometimes impossible).
The Template Method Design Pattern

```
AbstractClass

templateMethod()
virtual primitiveOperation1() = 0
virtual primitiveOperation2() = 0

ConcreteClass

virtual primitiveOperation1()
virtual primitiveOperation2()
```

The templateMethod() represents a sequence of operations that cannot be changed.

Only some steps in the sequence of operations can be changed in deriving classes.
The Template Method-Based Solution

class PersistenceInterface
{
public:
  PersistenceInterface();

  virtual ~PersistenceInterface();

  bool write( const Blob& blob );
  bool write( const Blob& blob, WriteCallback callback );
  bool read( Blob& blob, uint timeout );
  bool read( Blob& blob, ReadCallback callback, uint timeout );
  // ...

private:
  virtual bool doWrite( const Blob& blob ) = 0;
  virtual bool doWrite( const Blob& blob, WriteCallback callback ) = 0;
  virtual bool doRead( Blob& blob, uint timeout ) = 0;
  virtual bool doRead( Blob& blob, ReadCallback callback, uint timeout ) = 0;
  // ...
};

No virtual function in the public interface (except for the destructor).
In C++ we call this the Non-Virtual Interface Idiom (NVI)
The Template Method-Based Solution

```cpp
bool PersistenceInterface::write( const Blob& blob )
{
    LOG_INFO( "PersistenceInterface::write( Blob ), name = " << blob.name() << ": starting..." );

    if ( blob.name().empty() )
    {
        LOG_ERROR( "PersistenceInterface::write( Blob ): Attempt to" 
                    " write unnamed blob failed" );
        return false;
    }

    const auto start = std::chrono::high_resolution_clock::now();
    const bool success = doWrite( blob );
    const uint32_t time = std::chrono::high_resolution_clock::now() - start;

    LOG_INFO( "PersistenceInterface::write( Blob ), name = " << blob.name() << ": Writing blob of size " << blob.size() << 
              " bytes " << ( success ? "succeeded" : "failed" ) << " in" 
              " duration = " << time.count() << "ms" );

    return success;
}
```
The Template Method-Based Solution

class PersistenceInterface
{
public:
    PersistenceInterface();
    virtual ~PersistenceInterface();
    bool write(const Blob& blob);
    // ...

private:
    virtual bool doWrite(const Blob& blob) = 0;
    // ...
};
The Template Method-Based Solution

```cpp
class PersistenceInterface
{
public:
    PersistenceInterface();
    virtual ~PersistenceInterface();
    bool write( const Blob& blob );
    // ...

private:
    virtual bool prepareWrite() = 0;
    virtual bool doWrite( const Blob& blob ) = 0;
    // ...
};
```

By means of the Non-Virtual Interface Idiom (NVI) we have ...

- ... separated concerns and simplified change (SRP);
- ... enabled internal changes with no impact on callers;
- ... reduced duplication (DRY).
A Template Method-Based Solution — Guidelines

Guideline: Design classes for easy change.

Guideline: Design classes for easy extensions.
Design for Change and Extension
Design for Change and Extension

**Guideline:** Classes should be ...

- ... concise and focused on one purpose (SRP)
- ... developed with extensibility in mind (OCP)
- ... split into smaller pieces to favor reuse (DRY)
Design for Testability

Back to Basics: Class Design (Part 1)
- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)
- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility
Design for Testability

template< typename Type, size_t Capacity >
class FixedVector
{
    public:
        // ...
    
    private:
        // ...

    void destroy( Type* first, Type* last )
    {
        for( ; first != last; ++first )
            first->~Type();
    }

    size_t size_;  
    std::byte raw_[Capacity*sizeof(Type)];
};
Unit testing c++. How to test private members?

I would like to make unit tests for my C++ application.

What is the correct form to test private members of a class? Make a friend class which will test the private members, use a derived class, or some other trick?

Which technique does the testing APIs use?
Design for Testability

In C++ you can always do #define private public, #define class struct and then nothing is private anymore! – BeniBela Jan 6 '13 at 20:13

A shame we can't downvote a comment. @BeniBela I hope you realize that your suggestion is extremely bad coding practice. Pretty funny though. – Steven Lu Jan 6 '13 at 20:22

But what is the correct way to test private members? They have to be tested, right? – Daniel Saad Jan 6 '13 at 20:26

@jimmy_keen I agree that unitests are to test 'contract'. However, you might have some parts of your code governed by (internal) 'contract' that you don't want to expose to users of your code. public and private is mainly for access control for consumers of your code, and not necessarily for separating contract-governed and not. – jdm Jan 6 '13 at 21:01

8 Answers

Typically, one only tests the public interface as discussed in the question's comments.

There are times however when it is helpful to test private or protected methods. For example, the implementation may have some non-trivial complexities that are hidden from users and that can be tested more precisely with access to non-public members. Often it's better to figure out a way to remove that complexity or figure out how to expose the relevant portions publicly, but not always.

One way to allow unit tests access to non-public members is via the friend construct.
Design for Testability

The choices to test private members:

- #define private public 😱
- Make the test a friend 😞
- Make the member public 😊
- Derive the test class from the tested class 😞
- Separate concerns 😩
  - Move the member into a private namespace ...
  - ... or into another class (as a separate service).
Design for Testability

This is the design favored by the C++ standard library:

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

template< class ForwardIt >
constexpr void destroy( ForwardIt first, ForwardIt last );
```
Design for Testability

**Guideline:** Resist the urge to put everything into one class.

**Guideline:** Design classes to be testable.
Implementation Guidelines

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility
Resource Management

Back to Basics: Class Design (Part 1)
- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)
- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility
class Widget
{
public:
    // ...

    Widget(); // Default constructor
    Widget( Widget const& other ); // Copy constructor
    Widget& operator=( Widget const& other ); // Copy assignment operator
    Widget( Widget&& other ) noexcept; // Move constructor
    Widget& operator=( Widget&& other ) noexcept; // Move assignment operator
    ~Widget(); // Destructor

private:

};
Resource Management

class Widget
{
    public:
    // ...
    Widget( );                   // Default constructor
    Widget( Widget const& other ); // Copy constructor
    Widget& operator=( Widget const& other ); // Copy assignment operator
    Widget( Widget&& other ) noexcept;      // Move constructor
    Widget& operator=( Widget&& other ) noexcept; // Move assignment operator
    ~Widget( );                      // Destructor
    // ...

    private:
    int i;                            // - i as a representative of a fundamental type
    std::string s;                    // - s as a representative of a class (user-defined) type

};
class Widget
{
    public:
    // ...

    // ...

    private:
    int i;
    std::string s;
};

Core Guideline C.20: If you can avoid defining default operations, do

The Rule of 0
class Widget
{
    public:
    // ...

    private:
    int i;
    std::string s;
    Resource* pr;  // - pr as representative of a possible resource
};

Core Guideline C.32: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning

Core Guideline C.33: If a class has an owning pointer member, define a destructor

~Widget() { delete pr; }

// ...

// - pr as representative of a possible resource
class Widget
{
    public:
    // ...

    private:
    int i;
    std::string s;
    Resource* pr;  // - pr as representative of a possible resource
};

Core Guideline C.32: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning

Core Guideline C.33: If a class has an owning pointer member, define a destructor

Core Guideline R.3: A raw pointer (a T*) is non-owning
Resource Management

```cpp
class Widget
{
  public:
    // ...

  private:
    int i;
    std::string s;
    std::unique_ptr<Resource> pr;
};
```

**Core Guideline C.32:** If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning.

**Core Guideline C.33:** If a class has an owning pointer member, define a destructor.

**Core Guideline R.3:** A raw pointer (a T*) is non-owning.
Resource Management

```cpp
class Widget
{
public:
    // ...

private:
    int i;
    std::string s;
    std::unique_ptr<Resource> pr;
};
```

Core Guideline R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)

// ...

// ...
C++’s most important idiom:

RAII

(Resource Acquisition Is Initialization)
Resource Management

Back to Basics: RAIi and the Rule of Zero

Arthur O'Dwyer
2019-09-17
Resource Management

class Widget
{
    public:
        // ...

private:
    int i;
    std::string s;
    std::unique_ptr<Resource> pr;
};

Core Guideline R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)

Guideline: Strive for the Rule of 0: Classes that don’t require an explicit destructor, explicit copy operations and explicit move operations are much (!) easier to handle.

    // ...

std::unique_ptr cannot be copied!
class Widget
{
 public:
  // ...

 Widget( Widget const& other );
 Widget& operator=( Widget const& other );
 // Widget( Widget&& other ) noexcept;  // not declared
 // Widget& operator=( Widget&& other ) noexcept;  // not declared

 // ...

 private:
  int i;
  std::string s;
  std::unique_ptr<Resource> pr;
};
class Widget
{
    public:
        // ...

    private:
        int i;
        std::string s;
        std::unique_ptr<Resource> pr;
};

Core Guideline C.21: If you define or =delete any default operation, define or =delete them all

Widget( Widget const& other );
Widget& operator=( Widget const& other );
Widget( Widget&& other ) noexcept = default;
Widget& operator=( Widget&& other ) noexcept = default;
~Widget() = default;
    // ...

The Rule of 5
Resource Management

class Widget
{
  public:
    // ...
  
  private:
    int i;
    std::string s;
    std::shared_ptr<Resource> pr;  // fundamentally changes the semantics!
};

Core Guideline C.21: If you define or =delete any default operation, define or =delete them all

Widget( Widget const& other );
Widget& operator=( Widget const& other );
Widget( Widget&& other ) noexcept = default;
Widget& operator=( Widget&& other ) noexcept = default;
~Widget() = default;
// ...

The Rule of 5
Resource Management

```cpp
class Widget
{
    public:
        // ...

    private:
        int i;
        std::string s;
        std::shared_ptr<Resource> pr;  // fundamentally changes the semantics!
};
```

Core Guideline C.21: If you define or `=delete` any default operation, define or `=delete` them all

Core Guideline C.20: If you can avoid defining default operations, do

The Rule of 5

The Rule of 0
Resource Management

**Guideline:** Strive for the Rule of 0, but if it cannot be achieved (e.g. because the class implements RAII itself), follow the Rule of 5.

**Guideline:** Design classes for easy change.
Back to Basics: The Special Member Functions

KLAUS IGLBERGER

Wednesday, October 27th, 7:45am MDT
Content

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility
Back to Basics: Designing Classes (part 1 of 2)

KLAUS IGLBERGER