

CS100 Recitation 10

GKxx

April 25, 2022

Contents

- 1 Copy and Swap
- 2 Prevent Copying: An Interesting Way
- 3 Resource-managing Classes
 - Surrogate
 - Reference-counting Handles

Contents

- 1 Copy and Swap
- 2 Prevent Copying: An Interesting Way
- 3 Resource-managing Classes
 - Surrogate
 - Reference-counting Handles

Swap of Vector

The `std::swap` (defined in `<algorithm>`):

```
template <typename T>
inline void swap(T &lhs, T &rhs) {
    T tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

- Swap is done by three copies.
- Inefficient on some special objects, like `Vector`.

Swap of Vector

Specialize the template function `std::swap`.

- Non-template > template-specialization > template.

```
namespace std {  
template <>  
inline void swap<Vector>(Vector &lhs, Vector &rhs) {  
    // What should we do here?  
}  
} // namespace std
```

It seems that `std::swap<Vector>` needs to access the private members.

Swap of Vector

By convention, we define a public member:

```
class Vector {  
    public:  
        void swap(Vector &other) noexcept {  
            using std::swap;  
            swap(m_size, other.m_size);  
            swap(m_capacity, other.m_capacity);  
            swap(m_data, other.m_data);  
        }  
        // other members  
};
```

Swap of Vector

Then we can let `std::swap<Vector>` call that member:

```
namespace std {  
template <>  
inline void swap<Vector>  
    (Vector &lhs, Vector &rhs) noexcept {  
    lhs.swap(rhs);  
}  
} // namespace std
```

Note that

- we are not adding any more things to `std`.
- in contrast to the default version, our swap functions are **exception-free**.

Copy and Swap

Surprisingly, we obtain a copy assignment operator that is both **self-assignment-safe** and **exception-safe**!

```
class Vector {  
public:  
    Vector &operator=(const Vector &other) {  
        auto temp = other;  
        swap(temp);  
        return *this;  
    }  
};
```


Contents

- 1 Copy and Swap
- 2 Prevent Copying: An Interesting Way
- 3 Resource-managing Classes
 - Surrogate
 - Reference-counting Handles

Prevent Copying

Make the compiler unable to synthesize the copying operations?

- If the class has an uncopyable base class.
- If the class has an uncopyable member.

Which one is better?

Prevent Copying

Make the compiler unable to synthesize the copying operations?

- If the class has an uncopyable base class.
- If the class has an uncopyable member.

Which one is better?

Empty Base Optimization (EBO).

Uncopyable Class

```
class Uncopyable {  
    Uncopyable(const Uncopyable &);  
    Uncopyable &operator=(const Uncopyable &);  
};  
class Widget : public Uncopyable {  
    // We don't define the copy operations.  
    // The compiler is unable to synthesize them,  
    // because the copy operations of the base class are  
    // inaccessible.  
};
```

Private Inheritance

Such definition causes problem: A reference or pointer to `Uncopyable` can be bound to objects of every such class!

Private Inheritance

Such definition causes problem: A reference or pointer to `Uncopyable` can be bound to objects of every such class!

- `private` inheritance: **The inheritance relationship is a secret.**
- Every operation that relies on such relationship cannot be performed, unless in the subclass or `friend` of the subclass.
 - upcasting and downcasting
 - Accessing base members
 - dynamic binding
 -

Private Inheritance

```
class Uncopyable {  
    Uncopyable(const Uncopyable &);  
    Uncopyable &operator=(const Uncopyable &);  
};  
class Widget : private Uncopyable {  
    // ...  
};
```

Private Inheritance

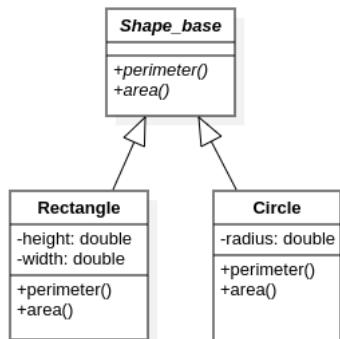
```
class Uncopyable {  
    Uncopyable(const Uncopyable &);  
    Uncopyable &operator=(const Uncopyable &);  
};  
class Widget : private Uncopyable {  
    // ...  
};
```

- This method is outdated from the perspective of C++11, but the way it uses inheritance is inspiring.

Contents

- 1 Copy and Swap
- 2 Prevent Copying: An Interesting Way
- 3 Resource-managing Classes
 - Surrogate
 - Reference-counting Handles

Shape



```
class Shape_base {
public:
    virtual double perimeter() const = 0;
    virtual double area() const = 0;
    virtual ~Shape_base() = default;
    Shape_base() = default;
};
```

Shape

```
class Rectangle : public Shape_base {  
    double height, width;  
public:  
    Rectangle(double h, double w);  
    virtual double perimeter() const override;  
    virtual double area() const override;  
};  
  
class Circle : public Shape_base {  
    double radius;  
public:  
    Circle(double r);  
    virtual double perimeter() const override;  
    virtual double area() const override;  
};
```

Problem

How can we define an array of shapes? (also `newed` arrays, containers, ...)

- `Shape_base shapes[100];` does not work.
 - Abstract base class.
 - Object slicing.
 - The sting of coworkers' derision.
- `Shape_base *shapes[100];` seems to work, but...
 - What happens when `shapes[i] = shapes[j];`?
 - The burden of memory management is on the user's part.

Virtual Copy Function

How can we copy an object correctly?

Virtual Copy Function

How can we copy an object correctly?

```
class Shape_base {  
    public:  
        virtual Shape_base *clone() const = 0;  
};  
class Rectangle : public Shape_base {  
    public:  
        virtual Shape_base *clone() const override  
        { return new Rectangle(height, width); }  
};  
class Circle : public Shape_base {  
    public:  
        virtual Shape_base *clone() const override  
        { return new Circle(radius); }  
};
```

Covariant Return-type

```
class Shape_base {  
    public:  
        virtual Shape_base *clone() const = 0;  
};  
class Rectangle : public Shape_base {  
    public:  
        virtual Rectangle *clone() const override  
            { return new Rectangle(height, width); }  
};  
class Circle : public Shape_base {  
    public:  
        virtual Circle *clone() const override  
            { return new Circle(radius); }  
};
```

Defining a Surrogate

Avoid manual memory management, while still keep the dynamic binding properties.

```
class Shape {  
    Shape_base *bp;  
public:  
    Shape_base() : bp(nullptr) {}  
    double perimeter() const {  
        return bp->perimeter();  
    }  
    double area() const {  
        return bp->area();  
    }  
};
```


Defining a Surrogate

- All other classes are **implementation details**, so all their members should be `private` (or `protected`).
- Declare `Shape` as a `friend` of them.
- Provide two interfaces `make_rectangle` and `make_circle`.
- Resource Acquisition Is Initialization, `RAII`.

Defining a Surrogate

- All other classes are **implementation details**, so all their members should be `private` (or `protected`).
- Declare Shape as a `friend` of them.
- Provide two interfaces `make_rectangle` and `make_circle`.
- `Resource Aquisition Is Initialization, RAII`.
- Since we allow default construction (so that we can define an array of Shape), we can provide an interface to tell whether `bp` is `nullptr`.

```
bool is_null() const { return !bp; }
```

Interfaces

```
class Shape {  
    friend Shape make_rectangle(double, double);  
    friend Shape make_circle(double);  
private:  
    Shape(Shape_base *p) : bp(p) {}  
};  
inline Shape make_rectangle(double h, double w)  
    { return new Rectangle(h, w); }  
inline Shape make_circle(double r)  
    { return new Circle(r); }
```

Make sure your surrogate is not influenced by outsider raw pointers!

Copy Control

Call the virtual clone function.

```
class Shape {  
public:  
    Shape(const Shape &other)  
        : bp(other.bp ? other.bp->clone() : nullptr) {}  
    Shape &operator=(const Shape &other) {  
        // Be careful with self-assignment!  
        auto p = other.bp ? other.bp->clone() : nullptr;  
        delete bp;  
        bp = p;  
        return *this;  
    }  
    ~Shape() { delete bp; }  
};
```

Homework Exercise

Use the copy-and-swap technique to define an assignment operator.

Use the Surrogate

Now we can use the shapes smoothly.

```
std::vector<Shape> shapeVec;  
for (int i = 0; i < n; ++i) {  
    if (some_condition(i))  
        shapeVec.push_back(make_rectangle(f(), g()));  
    else  
        shapeVec.push_back(make_circle(h()));  
}  
for (auto shape : shapeVec) {  
    std::cout << "perimeter == " << shape.perimeter()  
              << ", area == " << shape.area() << std::endl;  
}
```

The annoying pointers suddenly disappear!

Value Semantics and Reference Semantics

What will happen when we copy a surrogate object?

```
Shape a = somevalue(), b = somevalue();  
a = b;
```

- Value semantics: The object that b points to is copied. (The object is **unique**.)
- Reference semantics: a and b point to the same object. (The object is **shared**.)

Value Semantics and Reference Semantics

Pros and cons?

- Value semantics: always copy the object. Time- and space-costing.
- Reference semantics: avoid copying.
 - But if `b` is destroyed, should we destroy the object that `b` points to?

Value Semantics and Reference Semantics

Pros and cons?

- Value semantics: always copy the object. Time- and space-costing.
- Reference semantics: avoid copying.
 - But if `b` is destroyed, should we destroy the object that `b` points to?

We want both!

Contents

- 1 Copy and Swap
- 2 Prevent Copying: An Interesting Way
- 3 Resource-managing Classes
 - Surrogate
 - Reference-counting Handles

Reference-counting

We define a new kind of 'surrogate', named a **handle**.

- Allow an object to be shared by many handles, and **set a counter on it**.
- Increase the counter when a new handle is pointing to it.
- Decrease the counter when a handle no longer points to it.
- **When the counter is decreased to zero, delete the object!**
 - *"A man is dead when he is forgotten."*

Reference-counting

```
class Shape_base {  
    friend class Shape;  
    int use{1};  
    virtual double perimeter() const = 0;  
    virtual double area() const = 0;  
protected:  
    virtual ~Shape_base() = default;  
    Shape_base() = default;  
};
```

A Reference-counting Handle

```
class Shape {  
    Shape_base *bp;  
public:  
    double perimeter() const {  
        return bp->perimeter();  
    }  
    double area() const {  
        return bp->area();  
    }  
    bool is_null() const { return !bp; }  
private:  
    Shape(Shape_base *p) : bp(p) {}  
};
```

Copy Control

Copy ctor and dtor: (Be careful with null pointers!)

```
class Shape {  
public:  
    Shape(const Shape &other) : bp(other.bp) {  
        if (bp)  
            ++bp->use;  
    }  
    ~Shape() {  
        if (!--bp->use)  
            delete bp;  
    }  
};
```

Copy Control

Copy-assignment operator: Self-assignment-safe!!!

```
class Shape {  
public:  
    Shape &operator=(const Shape &other) {  
        if (other.bp)  
            ++other.bp->use;  
        if (bp && !--bp->use)  
            delete bp;  
        bp = other.bp;  
        return *this;  
    }  
};
```

Copy Control

This is **not** self-assignment-safe:

```
Shape &operator=(const Shape &other) {  
    if (bp && !--bp->use)  
        delete bp;  
    bp = other.bp;  
    if (other.bp)  
        ++other.bp->use;  
    return *this;  
}
```


Where is Copy?

It seems that we don't need the virtual `clone` functions at all!
But...

Where is Copy?

It seems that we don't need the virtual `clone` functions at all!
But... What if we allow some form of modification?

```
class Shape {  
public:  
    void some_modification(Type params) {  
        if (bp)  
            bp->some_modification(params);  
    }  
};
```

Where is Copy?

It seems that we don't need the virtual `clone` functions at all!
But... What if we allow some form of modification?

```
class Shape {  
public:  
    void some_modification(Type params) {  
        if (bp)  
            bp->some_modification(params);  
    }  
};
```

Suppose `Shape a = b;`. After modification on `a`, what if we still want `b` to hold the original object?

```
a.some_modification(/* ... */);
```

Copy on Write

Solution: We don't copy the object **until modification happens**.

- *Laziness is a virtue!*

```
class Shape {  
public:  
    void some_modification(Type params) {  
        if (bp) {  
            auto original = bp;  
            bp = original->clone();  
            if (!--original->use)  
                delete original;  
            bp->some_modification(params);  
        }  
    }  
};
```

Standard Library Support

Since C++11, the ideas of **surrogates** and **reference-counting handles** are supported in the standard library `<memory>` as **smart pointers**.

- `std::shared_ptr` is a reference-counting smart pointer.
- `std::weak_ptr` is a surrogate that keeps unique ownership of an object.
- `std::weak_ptr` might be used for some special purposes.

Reading Materials

- The ideas in this slides are from *Ruminations on C++* Chapter 5, 6, 7. Chapter 8 is related to Problem 3 in HW5. Chapter 9 and 10 talks about an interesting example.
- *Effective C++* Item 15, 17 talks about something else related.
- *C++ Primer* Chapter 12 (section 12.1) introduces smart pointers.
- To know about how to use smart pointers properly, see *Effective Modern C++* Item 18 - 22.