Part 3 — C++ Class Inheritance

Class Inheritance

An object oriented programming paradigm is to derive new classes from existing base-class(es). The derived class inherits data members and methods from the base-class(es).

This allows for:

- Code and data reuse
- Code adaptation, either by keeping base-class implementation or overriding it with new functionality

Single inheritance is when we inherit from one base-class, and *Multiple inheritance* is when we inherit from more than one base-class.

Class Inheritance

The usual model is an "*is-a*"" relationship:

- "a Rectangle is a Shape"
- "a Square is a Shape"
- "an Ellipse is a Shape"
- "a Circle is a Shape"

We say "*B is derived from A*", "*B is a sub-class of A*", "*A is a superclass of B*", and write $A \leftarrow B$.

This type relation defines type hierarchies which can have multiple levels:

```
 _______
|       |<---- Rectangle
| Shape |<---- Square
|       |<---- Ellipse
|_______|<---- Circle
```

Class Inheritance

Should we make Ellipse a super-class of Circle, because mathematically, the set of circles is a subset of ellipses?

Mathematical set inclusion doesn't imply an "is-a" relation, whereby base-class methods must be implemented by all derived classes and object keep their identity.

For instance, Ellipses may have a class method `stretchX` which stretches in the x direction. However, if applied to a circle, this would distort it and as a result would become an ellipse (i.e. lose their identity).

This — in a nutshell — is the notorious "circle-ellipse" problem

Class Inheritance Example

```cpp
// Base class
class Shape {
public:
    int color;      // All shapes have a color
    // And an area, default implementation is 0
    float area() const { return 0; }
};

// First specialized shape: axis aligned rectangle
class Rectangle : public Shape {
public:
    Rectangle(int l_, int r_, int t_, int b_) { ... }
    // overrides Shape::area()
    float area() const { return (r-l) * (b-t); }
private:
    // describes Rectangle (left,right,top,bottom), also inherits color
    int l, r, t, b;
};
```

Rectangles have the following data members:

```cpp
int color;
int l, r, t, b;
```

Inheritance Types

A derived class inherits all data members and methods from base-class(es), but we can control the access of these data members and methods both inside and outside the derived class:

- public: public members of the base class are public in the derived class, protected remain protected
- protected: public and protected members of the base class are protected in the derived class
- private: public and protected members of the base class are private in the derived class

Note: private members of the base class are always inaccessible to the derived class!

> **Best Practice:** By default, use public inheritance unless you require restricted access.

Inheritance Types - Example

```cpp
class Base {
public:
    int x;
protected:
    int y;
private:
    int z;
};

class PublicDerived : public Base {
    // x is public
    // y is protected
    // z is not accessible
};
class ProtectedDerived : protected Base {
    // x is protected
    // y is protected
    // z is not accessible
};
class PrivateDerived : private Base {
    // x is private
    // y is private
    // z is not accessible
};
```

Inheritance Types - Example

```cpp
class X {
public:
    int a;      // visible to all: users of X,
    void fa();  // X itself, and derived classes
protected:
    int b;      // visible to derived classes and X,
    void fb();  // but not to users of class X!
private:
    int c;      // only visible to member functions
    void fc();  // of X
};

// Y "is an" X
class Y : public X {
    void foo() {
        a = 0; fa(); /* OK */
        b = 0; fb(); /* OK */
        c = 0; fc(); /* NOT OK */
    }
    int d;
};

int main() {
    X x;
    Y y;
    x.a = 0;    // OK
    y.a = 0;    // OK
    x.b = 0;    // NOT OK
    x.c = 0;    // NOT OK
}
```

Sub-Class Memory Layout

When using single inheritance (one base-class), data members are added at the end of the base-class data. Using the previous example for `X`, `Y`:

```
X object layout:    | int a  (4 bytes)
                    | int b  (4 bytes)
                    | int c  (4 bytes)

Y object layout:    | int a  (4 bytes)  \
                    | int b  (4 bytes)    X part
                    | int c  (4 bytes)  /
                    | int d  (4 bytes)  - Y part
```

This layout scheme will allow us to treat Y objects as X objects, without code changes because the X parts in Y objects is stored at the same address relative to the first byte of the object.

Thus, code that works with a pointer/reference to an X object also works with a pointer/reference to a Y object.

Assignment Across Class Hierarchy

Suppose we have Y inherits data members and methods from X

```
class Y : public X ...
```

Public inheritance gives us an "*is-a*" relationship with public and protected X members visible in Y.

```
X a; Y b;
```

Are the assignments `a = b;` or `b = a;` meaningful? How would we implement Y assignment operator and copy constructor?

Similarly,

```
X *pa; Y *pb;
```

Are the assignments `pa = pb;` or `pb = pa;` meaningful?

Object Assignment

```cpp
class Y : public X { ... };
X a;
Y b;
a = b;       // ???
```

This is ok, because Ys are Xs — but the object is sliced:

- X assignment operator is called with reference to b
- X part of b is copied to a, Y part is not
- We are losing information

What about

```cpp
b = a;
```

This is **NOT OK!**

Y can contain more data than X, so how can we fill in the rest of the missing details?

Pointer Assignment

```
class Y : public X { ... };
X a, *pa;
Y b, *pb;

pa = &b;            // Ok?
pa = pb;            // Ok?
```

YES! pa now points to b, or *pb respectively. Note that the information about Y is not available when accessing *pa, because pa points to an X object.

What about

`pb = &a;` or `pb = pa;`

NO! *pb is an object of type Y. Again, where would the additional data come from?

Reusing Base-Class Operators

Base-class operators need to be called explicitly when you provide your own implementation in the derived class!

```cpp
struct X {
    X() { x = 0; }
    int x;
};
struct Y : public X {
    Y() { y = 0; }
    // Copy-construct X part and Y part
    Y(const Y &rhs) : X(rhs), y(rhs.y) {}

    Y &operator=(const Y &rhs) {
        if (&rhs == this) { return *this; }
        X::operator=(rhs);      // Call X's AO
        y = rhs.y;              // Assign Y's part
        return *this;
    }
    int y;
};
X a, *pa;  Y b;
a = b;                  // a.x = b.x. b.y is not copied (object slicing)
pa = &b;                // Ok, *pa is object of type X, Y part is invisible
```

Inheritance and Constructors

```cpp
class X {
public:
    X() = default;
    X(int _a) { ... }
};
class Y : public X {
public:
    Y() {  /* X() is called here */  }
    Y(int b) : X(b) { ...}  // explicit X(int) call
};
```

The derived class constructor calls the base-class constructor first to initialize base-class members. If a constructor is not defined, a default derived class constructor will be provided which calls the base-class constructor and constructs non-POD members of Y.

Base-class constructors, copy constructors, and assignment operators are called by the default derived class operators.

Inheritance and Constructors

If your derived classes simply add methods and you wish to make the base class constructor accessible through inheritance, use the `using` keyword to bring it into scope:

```cpp
class Parent {
    Parent() = default;
    Parent(int = 13, int = 42);
};

class Child : public Parent {
    using Parent::Parent;
    // The set of inherited constructors is
    // 1. Parent(const Parent&)
    // 2. Parent(Parent&&)
    // 3. Parent(int, int)

    // Child has the following constructors:
    // 1. Child()
    // 2. Child(const D2&)
    // 3. Child(D2&&)
    // 4. Child(int, int) <- inherited
};
```

Inheritance and Destructors

```cpp
struct X {
    X() { p = new int[100]; }
    ~X() { delete [] p; }
    int *p;
};
struct Y : public X {
    Y() {   /* X() is called here */
        q = new int[200];
    }
    ~Y() { delete [] q; /* ~X() called here */ }
    int *q;
};
```

Destructors are called in reverse order of constructor calls. The derived class destructor `Y()` calls based-class destructor.

- `Y() only deals with resources allocated in Y`!`
- `X()` is called at the end; takes care of the rest

Virtual Functions

Suppose we have a class `Graphics` which contains a list of pointers to objects to be drawn: `Circle`, `Rectangle`, `...`

How can we use inheritance to draw the respective shapes from the object pointers?

First solution: Objects contain an `id` to identify their type `...`

```cpp
class Shape {
public:
    int type_id;
    int color;
};

enum { CIRCLE, RECTANGLE, TRIANGLE, ... };

class Circle: public Shape {
    int x, y, r;
public:
    Circle() { x = y = r = 0; type_id = CIRCLE; }
    void draw(Screen *s) const { ... }
};
```

Virtual Functions

```cpp
class Graphics {
public:
    void draw() {
        for (int i = 0; i < n_objs; ++i) {
            Shape *p = objs[i];
            switch(p->type_id) {
            case CIRCLE:
                // cast: make the compiler believe that p actually points to a Circle.
                // p does point to a circle (type_id matches), so its safe
                static_cast<Circle*>(p)->draw(screen);
                break;
            case RECTANGLE:
                static_cast<Rectangle*>(p)->draw(screen);
                break;
            // ...
            }
        }
    }
    Shape *objs[];           // array of pointers to Shapes
    int n_objs;              // number of objects
    Screen *screen;          // where to draw shapes
};
```

Problem: Slow, need to change code when adding new shapes, hard to maintain

Polymorphism via Virtual Functions

Goal: Given a base-class pointer, execute member functions in the current object context

```
Shape *p = new Circle;
p->draw();              // Call Circle::draw()?
Shape *q = new Rectangle;
q->draw();              // Call Rectangle::draw()?
```

Polymorphism: Same function name, different action dependent on object type.

Requires that objects "*know*" their type, because the only information the runtime system has is the object data the base-class pointer points to.

Solution: Virtual functions

Polymorphism via Virtual Functions

Second Solution: Use virtual functions

```cpp
class Shape {
public:
    int color;
    // =0: marks abstract methods
    // derived classes must implement them (otherwise compiler error)
    virtual void draw(Screen *s) const = 0;
    virtual float area() const = 0;
};

class Circle : public Shape {
public:
    // Draws circle, implements virtual function
    void draw(Screen *s) const { ... }
};
```

The keyword `virtual` indicates that the methods in sub-classes are accessible via the base-class pointers, through *late binding* (runtime polymorphism).

Polymorphism via Virtual Functions

In the presence of virtual functions the compiler adds an extra data member to the class: the so-called "*virtual function table pointer*" (VFTP) which points to a block of memory that contains information about the class type including a table of virtual function addresses.

Sample memory layout of an object with at least one function declared virtual (by itself or an ancestor):

```
   Object              Virtual Function Table (VFT)

|  vftp     |----->| address of virtual function 0, say draw |
|  obj data |      | address of virtual function 1, say area |
|  ...      |      | address of virtual function 2 ...       |
                   | ...                                     |
                   | other class type information            |
```

There is one VFT for each class type in your program. The runtime system initializes them before `main()` is executed.

VFTs give the runtime system access to type information, such as virtual function addresses and base-class types.

This is called **RTTI** — RunTime Type Information

Invoking Virtual Functions

Recall that in C, we can store function addresses in pointers:

```c
void foo(int x) {}
void (*p)(int);
p = &foo;
(*p)(5);        // calls foo(5)
```

If `draw()` is virtual and p is a `Shape*`, when calling

```
p->draw(screen);
```

the runtime system looks up the function to call from the VFT accessible via p.

This allows us to iterate through the `Shape*` array and call the right draw function for each actual shape in turn.

Invoking Virtual Functions

Suppose `draw()` is the first virtual function in Shape, and p points to a Circle. Then

```
p->draw(screen);
```

does the following (equivalent C code):

```
    (*(p->vftp[0]))(p, screen);
       **********: address of Circle::draw
```

where `(*pointer-to-function)(params)` calls a function given a pointer to it, and p is passed as `this` pointer to give the function access to obj. members.

Likewise, if `area()` is the second virtual function,

```
p->area();
```

is equivalent to the following C code:

```
(*(p->vftp[1]))(p);
```

Circle and Rectangle Memory Layout

```
 Circle VFT              Shape *p = new Circle;  Shape *q = new Circle;
-------------- <----------------+------------           |
 Circle::draw    \              v            \          v
--------------    \          --------------   \   --------------
 Circle::area      ----------     vftp         ---     vftp
--------------               --------------       --------------
     ...                          color                color
--------------               --------------       --------------
                                   x                    x
                             --------------       --------------
                                   y                    y
                             --------------       --------------
                                   r                    r
                             --------------       --------------


Rectangle VFT
-------------- <-----------------------------   Shape *r = new Rectangle;
Rectangle:draw   \  Shape *t = new Rectangle;\        |
--------------    \          --------------   \   --------------
Rectangle::area    ----------     vftp         ---   vftp
--------------               --------------       --------------
     ...                          color              color
--------------               --------------       --------------
                                   l                  l
                             ---------------      --------------
                                   r                  r
                             --------------       --------------
                                  ...                ...
                             --------------       --------------
```

Circle and Rectangle Memory Layout

Virtual function overhead:

- **Space**: One more pointer per object, and one VFT per class
- **Time**: Table access, indirect function call

Because using virtual function creates runtime costs it is an optional feature in C++. If you don't use virtual functions, you don't pay anything.

Advantages:

- Simplifies code, extensible design
- Code independent of number of classes in the system, can be put in library

Virtual Destructors

```cpp
// virtual_destructor_example.cpp
#include <iostream>
class X {
public:
    X() { std::cout << "X CONST" << std::endl; }
    // THIS IS WRONG! WHY?
    ~X() { std::cout << "X DEST" << std::endl; }
    virtual void foo() { }
};
class Y : public X {
public:
    Y() { std::cout << "Y CONST" << std::endl; }
    ~Y() { std::cout << "Y DEST" << std::endl; }
    void foo() {  }
};

int main() {
    X *px = new Y;
    px->foo();
    delete px;
}
```

Virtual Destructors

Solution: Make the destructor virtual! Then,

```
delete px;
```

will call `~Y()`.

> **Rule:** If a class contains virtual functions, it must declare its destructor virtual as well. g++ checks this using -Wall.

Example

```cpp
class Shape {
public:
    virtual void draw(Screen *s) = 0;   // abstract
    virtual ~Shape() { }                // do nothing
};

class Circle    : public Shape { ... };
class Rectangle : public Shape { ... };
class Square    : public Shape { ... };

int main() {
    const int N = 3;
    Shape *A[N];        // allocate shapes (constructor arguments omitted)
    A[0] = new Circle;
    A[1] = new Rectangle;
    A[2] = new Square;
    for (int i=0; i < N; ++i) {     // draw all shapes using their respective draw functions
        A[i]->draw(screen);
    }
    for (int i=0; i < N; ++i) {     // delete all shapes using their respective destructors
        delete A[i];
    }
    // Important: the destructor loop above is needed, because pointers
    // are POD and when arrays go out of scope, element destructors are
    // not called!
    // When in doubt, simply ask yourself when typing new ... where the
    // corresponding delete is
}
```

Override

When overriding methods from a base class, there are several conditions that must be met:

- The base class function must be virtual
- The base and derived function names must be identical (except for destructors)
- The parameter types must be identical
- The `const`ness must be identical
- The return types and exception specifications must be compatible

If any of those conditions are not met, the method will not be overridden, and most of the time the compiler won't even give you a warning about it.

C++11 provides a solution: declaring the method as `override`.

Override — Example

```cpp
class Base {
public:
    virtual void f1() const;
    virtual void f1(int x);
    virtual void f3();
    void f4();
};

class Derived : public Base {
public:
    void f1() override;         // error: different constness
    void f1(long x) override;   // error: different parameter type
    void f3() override;         // OK
    void f4() override;         // error: base not virtual
};
```

Without using `override` the code would compile fine, and **maybe** the compiler will provide some warnings. Using `override` the code will fail to compile generating errors in the three cases.

Final

Another new feature in C++11 is the `final` keyword: it specifies that a virtual function cannot be overridden in a derived class or that a class cannot be inherited from.

```cpp
struct A {
    virtual void foo();
    void bar() final;                  // error: non-virtual function cannot be final
};

struct B : A {
    void foo() override final;         // OK: B::foo is final
};

struct C final : B {                   // Ok: C is final
    void foo() override;               // error: foo cannot be overridden
                                       // as it's final in B
};

struct D : C {                         // error: C is final
    // ...
};
```

Inheritance Tips

Use polymorphism, it's powerful and makes your code extensible and more readable. Its runtime overhead is small, but a pointer is added to each such object you create. So this may be problematic if you allocate many small objects.

Declare destructors virtual in the presence of other virtual member functions. `g++` will remind you using `-Wall`!

Base-class copy constructors are not automatically called in derived class copy constructors you provide. Use initializer list

```
: X(rhs)
```

In the derived class assignment operator call base-class X operator explicitly:

```
X::operator=(rhs);
```

Inheritance Tips

Beware: virtual function calls in base-class constructors call base-class functions! Derived class functions can't be called this way, because the derived class object hasn't been fully constructed yet!

```cpp
struct X {
    X() { foo(); }      // you might expect this to call derived
                        // class foo() because foo is virtual - NO!
    virtual void foo() { }
};

struct Y : public X {
    Y() { foo(); }
    // here, first X::foo() is called, then Y::foo(), although it is virtual!
    // Even calling foo() here may be problematic, if
    //  at that time the construction of Y isn't complete yet
    void foo() {  }
};
```

Lesson: Don't call virtual functions in constructors!