Part 3: Class Inheritance

Contents [DOCUMENT NOT FINALIZED YET]

Class Inheritance

Object Oriented Programming Paradigm

Derive new class from existing base-class(es)

Inherits data members and methods from base-class(es)

- Code/data reuse
- Code adaption
  - either keep base-class implementation
  - or override it with new functionality

Single inheritance: inherit from one base-class

Multiple inheritance: inherit from more than one base-class

- C#/Java and many other modern programming languages only support restricted forms (interfaces)

Inheritance Example

Sub-class/derived class specializes super-class/base-class

Usually models "is-a" relationships. E.g.

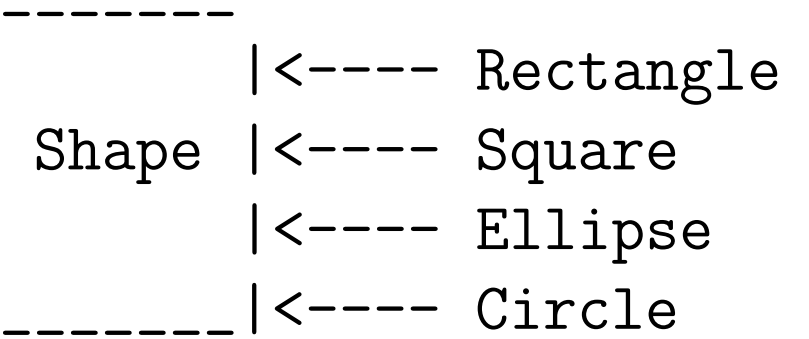
- "a Rectangle is a Shape"
- "a Square is a Shape"
- "an Ellipse is a Shape"
- "a Circle is a Shape"

We say

"B is derived from A", "B is a sub-class of A", "A is a super-class of B"

and write $A \leftarrow B$

This type relation defines type hierarchies which can have multiple levels. E.g.:



We could be tempted to make Ellipse a super-class of Circle, because mathematically, the set of circles is a subset of ellipses

However, mathematical set inclusion doesn't imply an "is-a" relation in the software-engineering sense, whereby base-class methods must be implemented by all derived classes and objects keep their identity

For instance, consider method `stretchX` which stretches an object in x direction. While certainly applicable to ellipses, it could distort circles which as a result become ellipses, i.e. lose their identity

This — in a nutshell — is the notorious "circle–ellipse" problem, which is discussed in detail at

`http://en.wikipedia.org/wiki/Circle-ellipse_problem`

Have a look if you are interested to learn more about that subject and ways to avoid problems similar to the one mentioned above

```
// base-class
class Shape
{
public:
  // all shapes have a color
  int color;

  // and area function
  // default implementation: return 0
  float area() const { return 0; }
};

// first specialized shape: axis aligned rectangle
class Rectangle : public Shape
{
public:
  // constructor: 4 values define a rectangle ...
  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;
};
```

Because Rectangle is derived from Shape, it inherits Shape's data:

A Rectangle has the following data members:

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

```
// second specialized shape
class Circle : public Shape
{
public:
  // constructor: circle defined by center and radius
  Circle(int x_, int y_, int r_) { ... }
  // overrides Shape::area()
  float area() const { return r * r * M_PI; }

private:
  // describes a Circle, also inherits color
  int x, y, r;
};
```

Circles have the following data members:

```
int color;
int x, y, r;
```

Inheritance Types

A derived class inherits all data members and methods from base-class(es)

Access permissions depend on qualifiers

```
class Y : public X { ... };
          ******
```

- Y "is-an" X design pattern. Y can be treated as an X
- Sub-class Y can access public and protected members of X, but cannot access private members of X
- Most common usage

```
class Y : protected X { ... };
          *********
```

- Y "is-implemented-in-terms-of" X design pattern
- public members of X become protected in Y
- This means that users of Y will not have access to any X components
- Rarely used

```
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

Example - using X,Y from the previous page:

```
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
```

As we will see in a moment, this layout scheme will allow us to treat Y objects as X objects without code changes because the X part in Y objects is stored at the same addresse 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

Assignments Across Class Hierarchy

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

Y inherits data members and methods from X

Public inheritance: "is-a" relationship public and protected X members visible in Y

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

Assignments: a = b; or b = a; meaningful?

How to implement Y assignment operator and copy constructor?

Similarly:

X *pa; Y *pb;

Assignments: pa = pb; or pb = pa; meaningful?

Object Assignment

```
class Y : public X {...};

X a;
Y b;

a = b;   ?
```

OK, because Ys are Xs — but object is sliced:

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

```
b = a;   ?
```

NOT OK !

Y can contain more data than X

How to fill the rest?

Example: you can't create a circle from a shape object. It's underspecified

Pointer Assignment

```
class Y : public X {...};

X a, *pa;
Y b, *pb;

pa = &b;  or   pa = pb;    OK?
```

YES

pa now points to b, or *pb respectively

Information about Y is not available when accessing *pa because pa points to an X object

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

NO

*pb is object of type Y

Again, where would the additional data come from?

Reusing Base-Class Operators

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

```
struct X
{
  X() { x = 0; }
  int x;
};

struct Y : public X
{
  Y() { y = 0; }

  Y(const Y &rhs)
     : X(rhs), y(rhs.y)  // copy-construct X part and Y part
  { // ^---- base-class name
  }

  Y &operator=(const Y &rhs)
  {
    if (&rhs == this) return *this; // guard against self-assignment
    X::operator=(rhs);     // call X AO, assigns X part
    y = rhs.y;             // assign Y-part
    return *this;
  }

  int y;
};

X a, *pa;
Y b;
a = b;      // a.x = b.x; b.y not copied (object slicing)
pa = &b;    // OK, *pa is object of type X. Y part invisible
```

Contructors/Destructors and Inheritance

```
class X
{
public:
  X(int a_=0) { ... }
};

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 contructor 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 Destructors

```
struct X
{
  X()  { p = new int[100]; }
  ~X() { delete [] p; }
  int *p;
};

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

Destructors are called in reverse order of constructor calls

Derived class destructor Y() calls base-class destructor

Y() only deals with resources allocated in Y!

X() is called at the end: takes care of the rest

Virtual Functions

Inheritance at work: graphics example

Class Graphics contains a list of pointers to objects to be drawn: Circles, Rectangles, ...

First solution: Objects contain an id to identify their type

```
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 { ... }
};
```

```
class Graphics
{
public:

  void draw()  // draw all objects
  {
    for (int i=0; i < n_objs; ++i) {
      Shape *p = objs[i];
      switch(p->type_id) {
        case CIRCLE:
        static_cast<Circle*>(p)->draw(screen);
//      **********************
// cast: make the compiler believe that p actually
// points to a Circle. C-equivalent:  (Circle*)p
// More about casts later. In this case p is actually
// pointing to a Circle, because the type_id matches.
// So the cast is safe.
        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
};
```

Problems: 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?
```

It would be nice if this calls Circle::draw and Rectangle::draw, respectively, even though p and q point to Shapes

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 pointers point to

Solution: Virtual Functions

Graphics 2

Second solution: use virtual functions

```
// abstract base-class
// (because it contains abstract functions)
class Shape
{
public:
  int color;
  // =0: marks abstract methods
  // derived classes must implement them
  virtual void  draw(Screen *s) const = 0;
  virtual float area() const          = 0;
};

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

Keyword virtual indicates that the methods in sub-classes is accessible via base-class pointers

This is a design choice you have in C++

In Java all methods are virtual

```
class Graphics
{
public:

  void draw() {          // draw all objects
    for (int i=0; i < n_objs; ++i) {
      objs[i]->draw(screen);
    }
  }
  Shape *objs[];  // array of pointers to Shapes
  int n_objs;     // number of objects
  Screen *screen;
};
```

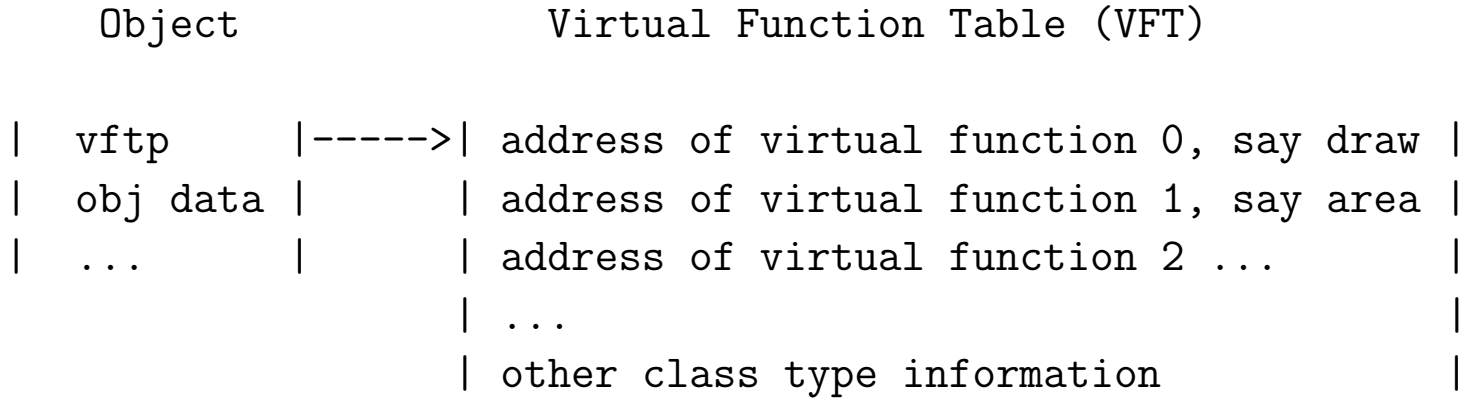
No type_id, no switch. Faster and easy to maintain

If the type of *objs[i] is known at runtime, the correct draw function can be called

HOW can this be implemented by the compiler writer?

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):



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)

Polymorphism in C++: invoking virtual functions

Did you know that you can store function addresses in pointers in 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

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

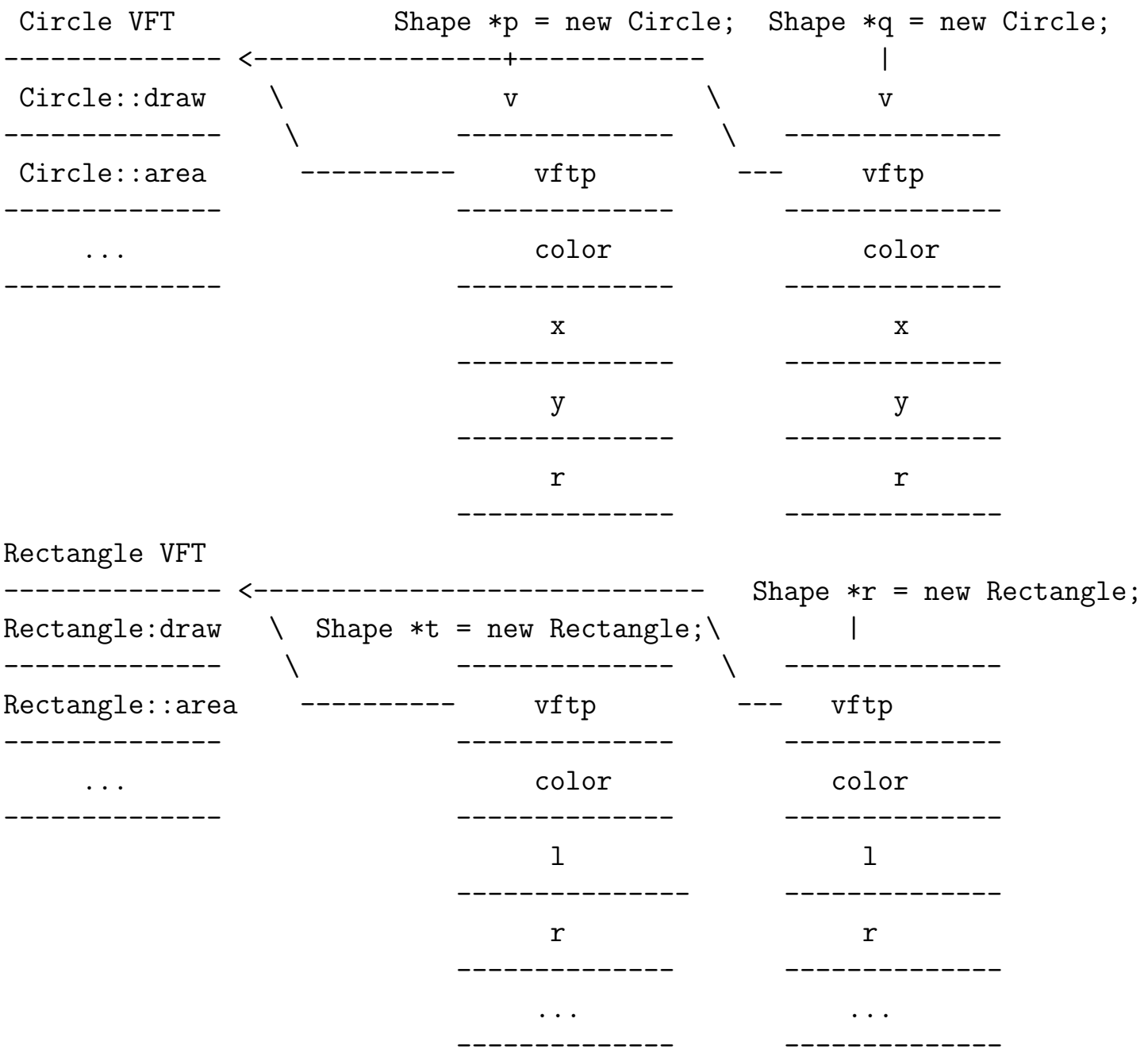
Likewise, if area() is the second virtual function

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

is equivalent to the following C code:

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

Circle and Rectangle Memory Layout:



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

```
class X
{
public:
  X()  { ... }
  ~X() { ... }   // THIS IS WRONG! WHY?
  virtual void foo() { ... }
};

class Y : public X
{
public:
  Y()  { ... }
  ~Y() { ... }
  virtual void foo() { ... }
};

X *px = new Y; // calls Y() - which calls X() first - OK

px->foo(); // calls Y::foo() - OK - polymorphism at work

delete px; // calls ~X(), but not ~Y(). PROBLEM if Y has
           // allocated resources - they won't be released
```

Solution: make 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

Complete Example

```
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 { ... };

{
  const int N = 3;
  Shape *A[N];

  // allocate shapes (constructor arguments omitted)
  A[0] = new Circle; A[1] = new Rectangle; A[2] = new Square;

  // draw all shapes using their respective draw functions
  for (int i=0; i < N; ++i) {
    A[i]->draw(screen);
  }

  // delete all shapes using their respective destructors
  for (int i=0; i < N; ++i) {
    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 and final

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 constness 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 overriden, 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:

```
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

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

```
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

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 explicitely: X::operator=(rhs);

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

```
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