Object Oriented Software Design

More operators, Automatic conversion, Inheritance, Overloading, Virtual destructor

Giuseppe Lipari
`http://retis.sssup.it/~lipari`

Scuola Superiore Sant'Anna – Pisa

December 10, 2010

Outline

Code reuse

- In C++ (like in all OO programming), one of the goals is to re-use existing code
- There are two ways of accomplishing this goal: composition and inheritance
  - Composition consists defining the object to reuse inside the new object
  - Composition can also expressed by relating different objects with pointers each other
  - Inheritance consists in enhancing an existing class with new more specific code
  - Until now you've seen only composition

Syntax

```
class A {
   int i;
protected:
   int j;
public:
   A() : i(0),j(0) {};
   ~A() {};
   int get() const {return i;}
   int f() const {return j;}
};

class B : public A {
  int i;
public:
  B() : A(), i(0) {};
  ~B() {};
  void set(int a) {j = a; i+= j}
  int g() const {return i;}
};
```

B is derived from A

Constructor

Inherited member

Use of Inheritance

- Now we can use B as a special version of A

```
int main()
{
   B b;
   cout << b.get() << "\n";     // calls A::get();
   b.set(10);
   cout << b.g() << "\n";
   b.g();
   A *a = &b;              // Automatic type conversion
   a->f();
  B *p = new A;
}
```

Constructor call order

- see ord-constr/
- Watch out for the order in which things are done inside a constructor ...
- Of course, destructors are called in reverse order

Redefinition and name hiding

- Of course, we can re-define some function member

```
class A {
   int i;
protected:
   int j;
public:
   A() : i(0),j(0) {};
   ~A() {};
   int get() const
     {return i;}
   int f() const
     {return j;}
};
```

```
class B : public A {
  int i;
public:
  B() : A(), i(0) {};
  ~B() {};
  int get() const
    {return i;}
  void set(int a)
    {j = a; i+= j}
  int f() const
    {return i;}
};
```

```
int main()
{
  B b;
  cout << b.get() << "\n";
  b.set(10);
  cout << b.f() << "\n";
}
```

Overloading and hiding

- There is no overloading across classes

```
class A {
   ...
public:
   int f(int, double);
}

class B : public A {
   ...
public:
   void f(double);
}
```

```
int main()
{
    B b;
    b.f(2,3.0);     // ERROR!
}
```

- `A::f()` has been hidden by `B::f()`
- either you redefine exactly the base version, or you will hide all the base members with the same name

Scoping

- Suppose that B refines function `f()`, and that `B::f()` wants to invoke `A::f()`

```
class A {
public:
    int f(int i);
};

class B : public A {
public:
    int f(int i) { return A::f(i) + 1;}
};
```

Not everything is inherited

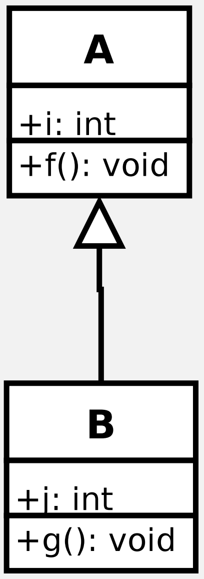
- What is not inherited
  - constructors
  - assignment operator
  - destructor
- Default constructor, copy constructor and assignment are automatically synthesized, if the programmer does not provide its own
  - when writing these functions, remember to call corresponding function in the base class!

Example

```
class A {
    int i;
public:
    A(int ii) : i(ii) {};
    A(const A&a) : i(a.i) {}
    A &operator=(const A&a) {i = a.i;}
};

class B : public A {
    int j;
public:
    B(int ii) : A(ii), j(ii+1) {};
    B(const B& b) : A(b), j(b.j) {}
    B &operator=(const B& b) {
        A::operator=(b); j = b.j;
    }
};
```

- Wherever you can use A, you can use B ...
- an object of class B ***isA*** subtype of A
- This is called *up-casting*

Graphical representation



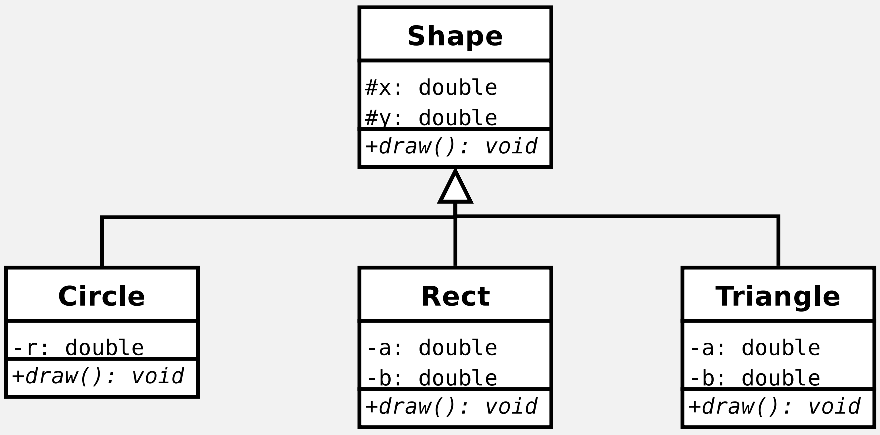
- This is UML
- If we have a reference to B, we can cast implicitly to a reference to A
- a reference to A cannot be cast implicitly to B (*downcast*)

Upcasting and downcasting

- Upcasting is a fundamental activity in OO programming (and it is safe)
- Downcasting is not safe at all
    - the compiler will issue an error when you try to implicitly downcast
- To better understand upcasting, we need to introduce virtual functions

Virtual functions

- Let's introduce virtual functions with an example



Implementation

```
class Shape {
protected:
  double x,y;
public:
  Shape(double x1, double y2);
  virtual void draw() = 0;
};

class Circle : public Shape {
  double r;
public:
  Circle(double x1, double y1,
         double r);
  virtual void draw();
};
```

```
class Rect : public Shape {
  double a, b;
public:
  Rect(double x1, double y1,
       double a1, double b1);
  virtual void draw();
};

class Triangle : public Shape {
  double a, b;
public:
  Triangle(double x1, double y1,
           double a1, double b1);
  virtual void draw();
};
```

We would like to collect shapes

- Let's make a vector of shapes

```
vector<Shapes *> shapes;

shapes.push_back(new Circle(2,3,10));
shapes.push_back(new Rect(10,10,5,4));
shapes.push_back(new Triangle(0,0,3,2));

// now we want to draw all the shapes ...

for (int i=0; i<3; ++i) shapes[i]->draw();
```

- We would like that the right draw function is called
- However, the problem is that Shapes::draw() is called
- The solution is to make draw virtual

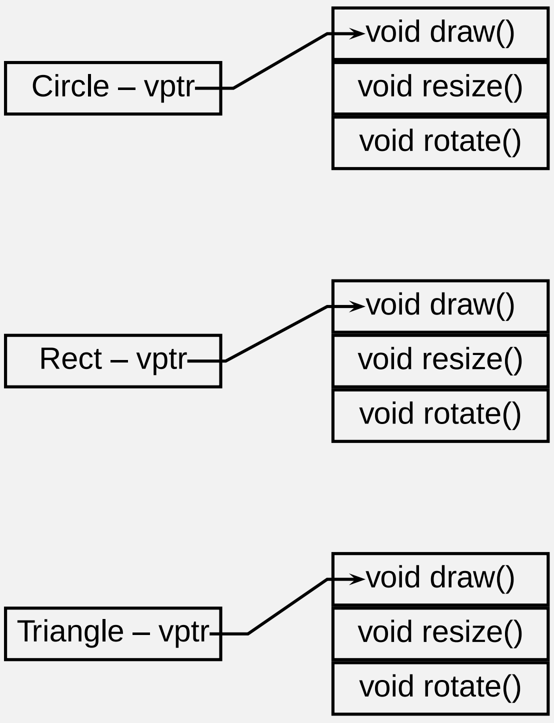
Virtual functions

```
class Shape {
protected:
  double x,y;
public:
  Shape(double xx, double yy);
  void move(double x, double y);
  virtual void draw();
  virtual void resize(double scale);
  virtual void rotate(double degree);
};

class Circle : public Shape {
  double r;
public:
  Circle(double x, double y,
         double r);
  void draw();
  void resize(double scale);
  void rotate(double degree);
};
```

- `move()` is a regular function
- `draw()`, `resize()` and `rotate()` are virtual
- see shapes/

Virtual table

- When you put the virtual keyword before a function declaration, the compiler builds a vtable for each class



Calling a virtual function

- When the compiler sees a call to a virtual function, it performs a late binding, or dynamic binding
  - each object of a class derived from `Shape` has a `vptr` as first element.
    - It is like a hidden member variable
- The virtual function call is translated into
  - get the `vptr` (first element of object)
  - move to the right position into the vtable (depending on which virtual function we are calling)
  - call the function

Equivalent in C

- It is easy to replicate this behavior in C
  - it suffices to use array of *pointers to functions*
  - However, in C this has to be done explicitly
  - It is not nice code, and it is easy to introduce bugs
- In C++, it is automatic
  - it is quite efficient,
  - if you look at the generated assembler code, it is just two additional assembler instructions with respect to a regular function call

Examples

- See shapes/
- See virtual/

Overloading and overriding

- When you override a virtual function, you cannot change the return value
  - Simply because the compiler will not know which function to actually call
- There is only one exception to the previous rule:
  - if the base class virtual method returns a pointer or a reference to an object of the base class . . .
  - . . . the derived class can change the return value to a pointer or reference of the derived class

Overload and override

- Examples

Correct

```
class A {
public:
   virtual A& f();
   int g();
};

class B: public A {
public:
   virtual B& f();
   double g();
};
```

Wrong

```
class A {
public:
   virtual A& f();
};

class C: public A {
public:
   virtual int f();
};
```

Private inheritance

- A base class can be inherited as private, instead of public:
- All the members of the base class become private

```
class A {
protected:
  void f();
public:
  int g();
};

class B : public A {
public:
  int h();
};

int main() {
  B b1;
  b1.f();    // NO
  b1.g();   // OK
}
```

```
class C : private A {
public:
  int h();
};

int main() {
  C c1;
  c1.f();   // NO
  c1.g();    // NO
}
```

Destructors

- What happens if we try to destruct an object through a pointer to the base class?

```
class A {
public:
  A();
  ~A();
};

class B : public A {
public:
  B();
  ~B();
};

int main() {
  A *p;
  p = new B;
  // ...
  delete p;
}
```

Virtual destructor

- This is a big mistake!
  - The destructor of the base class is called, which “destroys” only part of the object
  - You will soon end up with a segmentation fault (or illegal access), or memory corruption
- To solve the problem, we have to declare a virtual destructor
  - If the destructors are virtual, they are called in the correct order
  - See

Restrictions

- Never call a virtual function inside a destructor!
  - Can you explain why?
- You can not call a virtual function inside a constructor
  - in fact, in the constructor, the object is only half-built, so you could end up making a wrong thing
  - during construction, the object is not yet ready! The constructor should only build the object
- Same thing for the destructor
  - during destruction, the object is half destroyed, so you will probably call the wrong function

Restrictions

- Example

```
class Base {
  string name;
public:
  Base(const string &n) : name(n) {}
  virtual string getName() { return name; }
  virtual ~Base() { cout << getName() << endl;}
};
```

```
class Derived : public Base {
  string name2;
public:
  Derived(const string &n) : Base(n), name(n + "2") {}
  virtual string getName() {return name2;}
  virtual ~Derived() {}
};
```

Pure virtual functions

- A virtual function is pure if no implementation is provided
- Example:

```
class Abs {
public:
  virtual int fun() = 0;
  virtual ~Abs();
};
class Derived public Abs {
public:
  Derived();
  virtual int fun();
  virtual ~Derived();
};
```

This is a pure virtual function. No object of Abs can be instantiated.

One of the derived classes must *finalize* the function to be able to instantiate the object.

Interface classes

- If a class only provides pure virtual functions, it is an *interface class*
  - an interface class is useful when we want to specify that a certain class *conforms* to an interface
  - Unlike Java, there is no special keyword to indicate an interface class
  - more examples in section multiple inheritance

What happens?

- Consider the following code snippet

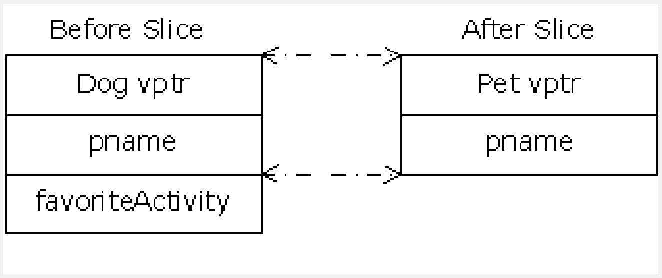
```
class Employee {
  // ...
  Employee& operator=(const Employee& e);
  Employee(const Employee& e);
};

class Manager : public Employee {
  // ...
};

void f(const Manager& m)
{
  Employee e;
  e = m;
}
```

Slicing

- Only the “`Employee` part” of `m` is copied from `m` to `e`.
  - The assignment operator of `Employee` does not know anything about managers!
- This is called “object slicing” and it can be a source of errors and various problems
- Solution: pay attention to the assignment operator!

Another example

- If you upcast to an object instead of a pointer or reference, something will happen that may surprise you: the object is “sliced” until all that remains is the subobject that corresponds to the destination type of your cast.
- Consider the code in inheritance/slicing/slicing.cpp
- any calls to describe() will cause an object the size of Pet to be pushed on the stack
- the compiler copies only the Pet portion of the object and slices the derived portion off of the object, like this:



slicing cont.

- What happens to the virtual function call?
- The compiler is smart, and understand what is going on!
  - the compiler knows the precise type of the object because the derived object has been forced to become a base object.
  - When passing by value, the copy-constructor for a Pet object is used, which initializes the VPTR to the Pet VTABLE and copies only the `Pet` parts of the object.
  - There’s no explicit copy-constructor here, so the compiler synthesizes one.
  - Under all interpretations, the object truly becomes a Pet during slicing.

Type conversion

- In C and C++, if the compiler sees an expression or function call using a type that isn't quite the one it needs, it can often perform an automatic type conversion from the type it has to the type it wants.
- In C++, you can achieve this same effect for user-defined types by defining automatic type conversion functions.
- These functions come in two flavors:
  - a particular type of constructor and
  - an overloaded operator.

Type conversion via constructor

- If you define a constructor that takes as its single argument an object (or reference) of another type, that constructor allows the compiler to perform an automatic type conversion.
- For example,

```
class One {
public:
  One() {}
};

class Two {
public:
  Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
  f(one); // Wants a Two, has a One
}
```

Another example

```
class AA {
  int ii;
public:
  AA(int i) : ii(i) {}
  void print() { cout << ii << endl;}
};
void fun(AA  x) {
  x.print();
}
int main()
{
   fun(5);
}
```

- The integer is “converted” into an object of class AA

Preventing implicit conversion

- To prevent implicit conversion, we can declare the constructor to be `explicit`

```
class AA {
  int ii;
public:
  explicit AA(int i) : ii(i) {}
  void print() { cout << ii << endl;}
};
void fun(AA  x) {
  x.print();
}
int main()
{
   fun(5); // error, no implicit conversion
   fun(AA(5)); // ok, conversion is explicit
}
```

Type conversion through operator

- This is a very special kind of operator:

```
class Three {
  int i;
public:
  Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
  int x;
public:
  Four(int xx) : x(xx) {}
  operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
  Four four(1);
  g(four);
  g(1);  // Calls Three(1,0)
}
```

Differences

- With the constructor technique, the destination class is performing the conversion;
  - creating a single-argument constructor always defines an automatic type conversion (even if it's got more than one argument, if the rest of the arguments are defaulted)
  - you can turn it off using `explicit`
- With operators, the source class performs the conversion.
  - The value of the constructor technique is that you can add a new conversion path to an existing system as you're creating a new class.
  - In addition, there is no way to use a constructor conversion from a user-defined type to a built-in type, this is possible only with operator overloading

An useful example

- How to convert a “String” class to `char *`

```
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
  string s;
public:
  Stringc(const string& str = "") : s(str) {}
  operator const char*() const {
    return s.c_str();
  }
};

int main() {
  Stringc s1("hello"), s2("there");
  strcmp(s1, s2); // Standard C function
  strspn(s1, s2); // Any string function!
}
```

Problems with conversion

- The first problems happens when someone uses **both** constructor and operator

```
class Orange; // Class declaration

class Apple {
public:
  operator Orange() const; // Convert Apple to Orange
};

class Orange {
public:
  Orange(Apple); // Convert Apple to Orange
};

void f(Orange) {}

int main() {
  Apple a;
//! f(a); // Error: ambiguous conversion
}
```

- `Just don’t do it`

Multiple conversions

- This is somewhat called *fan-out*

```
class Orange {};
class Pear {};

class Apple {
public:
  operator Orange() const;
  operator Pear() const;
};

// Overloaded eat():
void eat(Orange);
void eat(Pear);

int main() {
  Apple c;
//! eat(c);
  // Error: Apple -> Orange or Apple -> Pear ???
}
```

- Again: `don't do it`, don't put yourself into troubled waters

Brain teaser

```
class Fi {};

class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

class Fo {
  int i;
public:
  Fo(int x = 0) : i(x) {}
  operator Fee() const { return Fee(i); }
};

int main() {
  Fo fo;
  Fee fee = fo;
}
```

- How many functions calls in the instruction `Fee fee = fo;`?

Virtual operator overloading

- You can make operators virtual just like other member functions
- However, implementing virtual operators may often become confusing
  - because you may be operating on two objects, both with unknown types
- For example, consider a system that deals with matrices, vectors and scalar values, all three of which are derived from class Math
  - We want to make the `operator*` a virtual function, so that we can transparently call the correct function when multiplying two objects
  - However, the actual virtual function that is called depends on the type of the left operand of the `operator*`
  - How to make it depend also on the right operand?
- see multiple_dispatch/

An example

```
class Matrix : public Math {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this);
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Matrix" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Matrix" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Matrix" << endl;
    return *this;
  }
};
```

All cases

- Basically, we build a matrix of cases:
- when the left operand is a Matrix, the right operand can be:
  - a Matrix
  - a Scalar
  - a Vector
- same for Vector and Scalar

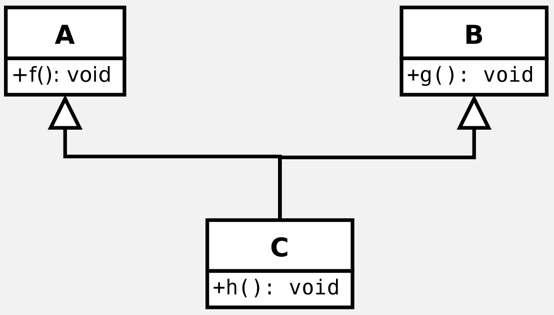
	Matrix	Scalar	Vector
Matrix			
Scalar			
Vector			

Multiple dispatch

- This technique is called *multiple dispatch*
  - The first dispatch is cause by the virtual operator*, which depends on the left operand (rows in the matrix)
  - the second dispatch depends on the right operand, and it is performed by a second virtual function **multiply**.
- This technique is not so common, but may be useful in some cases.

Multiple inheritance

- A class can be derived from 2 or more base classes



- C inherits the members of A and B

Multiple inheritance

- Syntax

```
class A {
 public:
   void f();
};

class B {
public:
   void f();
};

class C : public A, public B
{
    ...
};
```

- If both A and B define two functions with the same name, there is an ambiguity
  - it can be solved with the scope operator

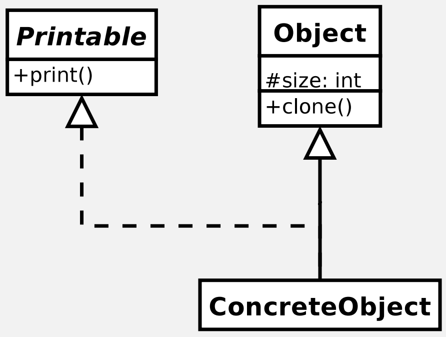
```
C c1;

c1.A::f();
c1.B::f();
```

Why multiple inheritance?

- Is multiple inheritance really needed?
  - There are contrasts in the OO research community
  - Many OO languages do not support multiple inheritance
  - Some languages support the concept of “Interface” (e.g. Java)
- Multiple inheritance can bring several problems both to the programmers and to language designers
- Therefore, the much simpler *interface inheritance* is used (that mimics Java interfaces)

Interface inheritance

- It is called interface inheritance when an onjecy derives from a base class and from an *interface class*
- A simple example

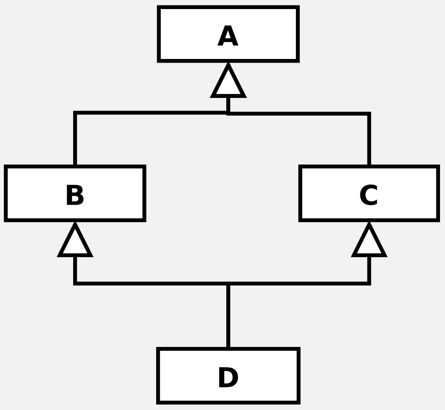


Interface and implementation inheritance

- In *interface inheritance*
  - The base class is abstract (only contains the interface)
  - For each method there is only one final implementation in the derived classes
  - It is possible to always understand which function is called
- Implementation inheritance is the one normally used by C++
  - the base class provides some implementation
  - when inheriting from a base class, the derived class inherits its implementation (and not only the interface)

The diamond problem

- What happens if class D inherits from two classes, B and C which both inherith from A?
- This may be a problem in object oriented programming with multiple inheritance!

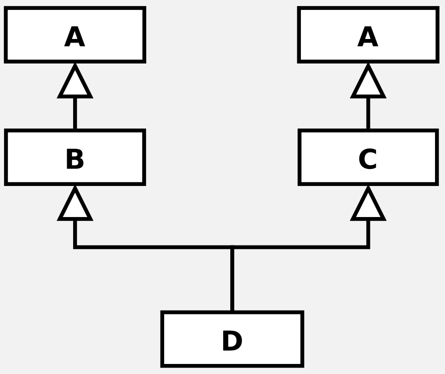


- Problem:
  - If a method in D calls a method defined in A (and does not override the method),
  - and B and C have overridden that method differently,
  - from which class does D inherit the method: B, or C?
  - In C++ this is solved by using keyword "virtual" when inheriting from a class

Virtual base class

- If you do not use virtual inheritance

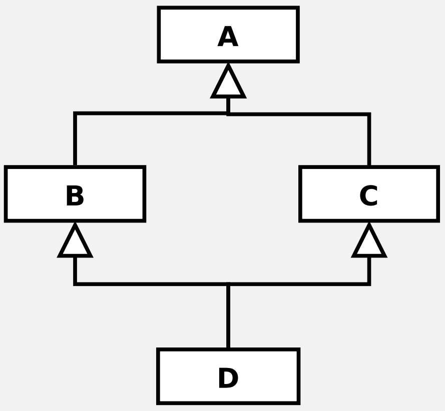
```
class A {...};
class B : public A {...};
class C : public A {...};
class D : public B, public C
{
  ...
};
```



- With public inheritance the base class is duplicated
- To use one of the methods of A, we have to specify which “path” we want to follow with the scope operator
- Cannot upcast!
- see minher/duplicate.cpp

Virtual base class

```
class A {...};
class B : virtual public A {...};
class C : virtual public A {...};
class D : public B, public C {...};
```



- With virtual public inheritance the base class is inherited only once
- see minher/vbase.cpp for an example

Initializing virtual base

- The strangest thing in the previous code is the initializer for Top in the Bottom constructor.
- Normally one doesn't worry about initializing subobjects beyond direct base classes, since all classes take care of initializing their own bases.
- There are, however, multiple paths from Bottom to Top,
  - who is responsible for performing the initialization?
- For this reason, the most derived class must initialize a virtual base.
- But what about the expressions in the Left and Right constructors that also initialize Top?
  - they are ignored when a Bottom object is created
  - The compiler takes care of all this for you

When inheritance is used

- Inheritance should be used when we have a *isA* relation between objects
  - you can say that a circle is a kind of shape
  - you can say that a rect is a shape
- What if the derived class contains some special function that is useful only for that class?
  - Suppose that we need to compute the diagonal of a rectangle

isA vs. *isLikeA*

- If we put function diagonal() only in Rect, we cannot call it with a pointer to shape
  - in fact, diagonal() is not part of the interface of shape
- If we put function diagonal() in Shape, it is inherited by Triangle and Circle
  - diagonal() does not make sense for a Circle
  - we should raise an error when diagonal is called on a Circle
- What to do?

The fat interface

- one solution is to put the function in the Shape interface
  - it will return an error for the other classes like Triangle and Circle
- another solution is to put it only in Rect and then make a downcasting when necessary
  - see diagonal/ for the two solutions
- This is a problem of inheritance! Anyway, the second one it probably better

Downcasting

- One way to downcast is to use the dynamic_cast construct

```
class Shape { ... };

class Circle : public Shape { ... };

void f(Shape *s)
{
  Circle *c;

  c = dynamic_cast<Circle *>(s);
  if (c == 0) {
    // s does not point to a circle
  }
  else {
    // s (and c) points to a circle
  }
}
```

Dynamic cast

- The dynamic_cast() is solved at run-time, by looking inside the structure of the object
- This feature is called *run-time type identification* (RTTI)
- In some compiler, it can be disabled at compile time

Liskov Substitution Principle

Functions that use pointers of references to base classes must be able to use objects of derived classes without knowing it.

Barbara Liskov, “Data Abstraction and Hierarchy,” SIGPLAN Notices, 23,5 (May, 1988).

- The importance of this principle becomes obvious when you consider the conse quences of violating it. If there is a function which does not conform to the LSP, then that function uses a pointer or reference to a base class, but must know about all the derivatives of that base class.

Example of violations of LSP

- One of the most glaring violations of this principle is the use of C++ Run-Time Type Information (RTTI) to select a function based upon the type of an object.

```
void DrawShape(const Shape& s)
{
  Square *q;
  Circle *c;

  if (q = dynamic_cast<Square *>(s))
    DrawSquare(q);
  else if (c = dynamic_cast<Circle *>(s))
    DrawCircle(c);
}
```

- Clearly the DrawShape function is badly formed. It must know about every possible derivative of the Shape class, and it must be changed whenever new derivatives of Shape are created. Indeed, many view the structure of this function as anathema to Object Oriented Design.

Other examples of violation

- there are other, far more subtle, ways of violating the LSP

```
class Rectangle
{
  public:
    void   SetWidth(double w) {itsWidth=w;}
    void   SetHeight(double h) {itsHeight=w;}
    double GetHeight() const   {return itsHeight;}
    double GetWidth() const    {return itsWidth;}
  private:
    double itsWidth;
    double itsHeight;
};
```

- Now suppose we want to introduce a Square
  - A square is a particular case of a rectangle, so it seems natural to derive class Square from class rectangle
  - Do you see problems with this reasoning?

Problems?

- Square will inherit the SetWidth and SetHeight functions.
- These functions are utterly inappropriate for a Square!
  - since the width and height of a square are identical.
- This should be a significant clue that there is a problem with the design.

Fixing it

- Suppose we write the code so that when we set the height the with changes as well, and viceversa.
- We have to do the Rectangle members virtual, otherwise it does not work!

Fixing it – the code

```
class Rectangle
{
  public:
    virtual void SetWidth(double w)  {itsWidth=w;}
    virtual void SetHeight(double h) {itsHeight=h;}
    double       GetHeight() const   {return itsHeight;}
    double       GetWidth() const    {return itsWidth;}
  private:
    double itsHeight;
    double itsWidth;
};
class Square : public Rectangle
{
  public:
    virtual void SetWidth(double w);
    virtual void SetHeight(double h);
};
void Square::SetWidth(double w)
{
  Rectangle::SetWidth(w);
  Rectangle::SetHeight(w);
}
void Square::SetHeight(double h)
{
```

```
  Rectangle::SetWidth(h);
```

The real problem

- We changed the interface, not only the behavior!

```
void g(Rectangle& r)
{
  r.SetWidth(5);
  r.SetHeight(4);
  assert(r.GetWidth() * r.GetHeight()) == 20);
}
```

- The code above was written by a programmer that did not know about squares
- what happens if you pass it a pointer to a Square object?
  - the programmer made the (at that time correct) assumption that modifying the height does not change the width of a rectangle.

Good design is not obvious

- The previous design violates the LSP
- A Square is not the same as a Rectangle for some pieces of code
  - from the behavioral point of view, they are not equivalent (one cannot be used in place of the other)
  - The *behavior* is what is important in software!
  - See the paper (downloadable from the web site).