```
class point
private:
  float x,y;
public:
  point(float xx, float yy) {
    x=xx;
    y=yy;
  }
  void move (float dx, float dy) {
    x+=dx;
    y+=dy;
  }
};
```

default ctor

```
point () {
  x=y=0.0;
}
```

hidden call to ctor

```
point p1(3.0,4.2);
p1.move(0.3, 0.2);
```

Can also have accessor functions – maintains private data and public methods.

```
class point {
private:
   float x,y;
public:
   void setXY(float xx, float yy) {
      x = xx;
      Y = yy;
   }
   float getx() { return x; }
   float getY() { return y; }
   void move(...) {
      ...
   }
};
```

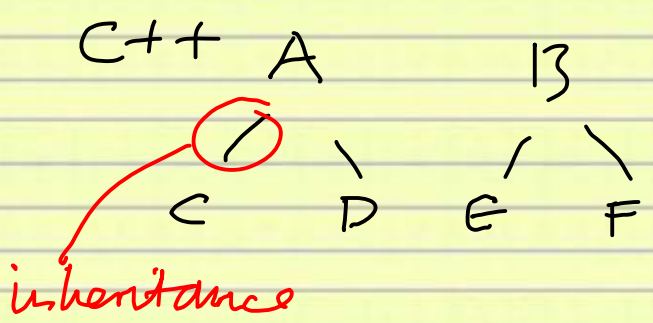
Use accessor functions

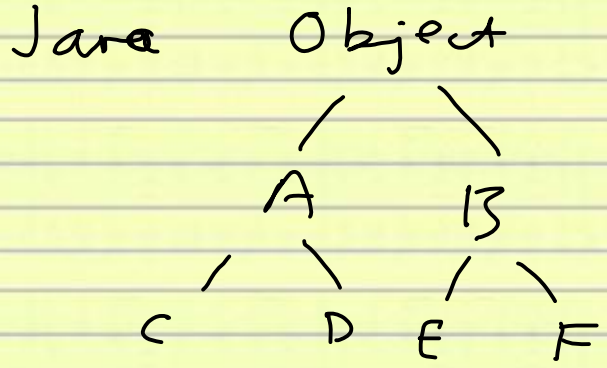
```
point p1;   <- x,y have unknown values, even though a default ctor is provided
p1.setXY(3.0, 4.7);
```

If no ctor is defined the default is provided:

```
point() {}   <- empty body
```

Inheritance : indicates a relationship between classes.
C++ uses a "forest" of classes whereas Java only allows a single tree.





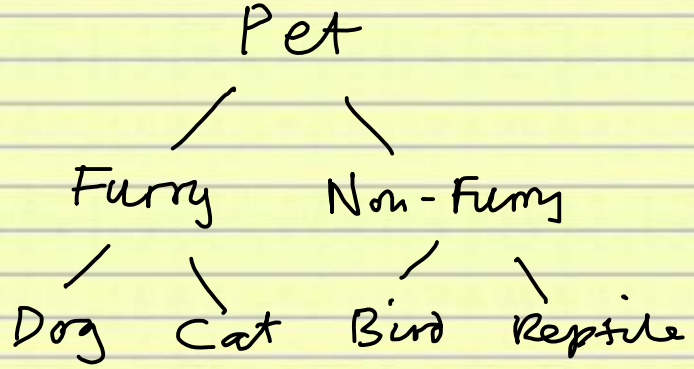

```
class B {
  ...
};
class D : public B {
  ...
};
```

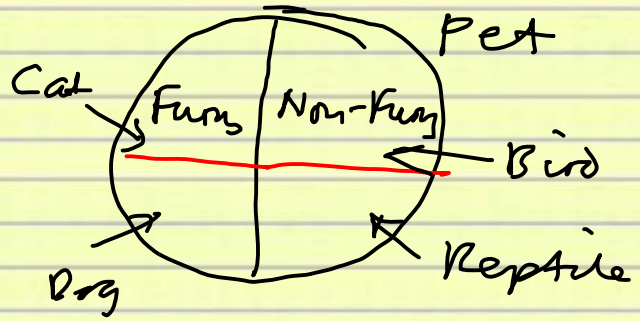
inheritance

```
class B {
  ...
}
class D extends B {
  ...
}
```

We call this "subclassing".
public subclassing also creates a subtype relationship.

We can see this in a hierarchy of natural types.





When we put classes in an inheritance hierarchy we are really determining subsets of objects. Eventually we get to level of individuals:

Dog
|
Terrier — "Fido"

When we inherit, we can inherit fields and methods.

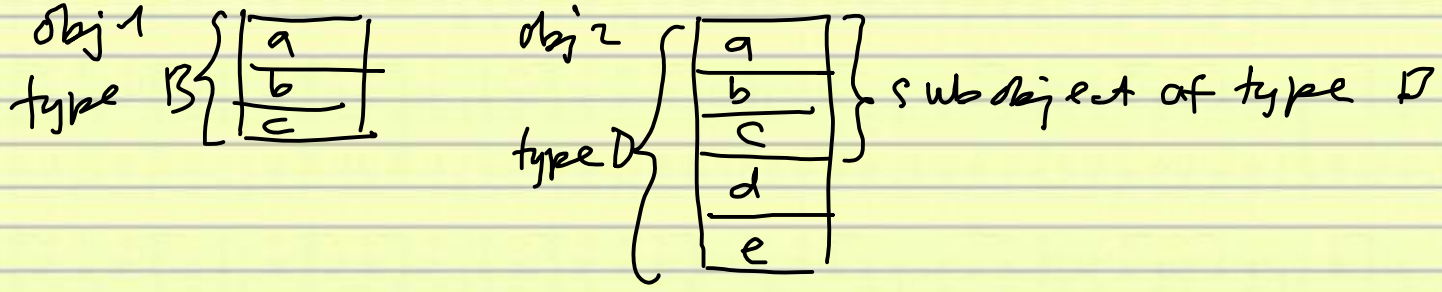
```
class B {
private:
  int a,b,c;
public:
  void f1() {...}
};

class D : public B {
private:
  int d,e;
public:
  void f2() {...}
};
```

Use;

```
B obj1;
obj1.f1();

D obj2;
obj2.f2();
obj2.f1(); <- f1 is inherited
```



When things are inherited, their access modifier remains the same: private becomes private, public becomes public:

We can open up access in a limited sense with access type "protected"

```
class B {                          <- base class
protected:
    int a;
    ...
};

                                   <- derived class
class D : public B {
    ...
public:
    void f1() { ... a ... }
};
```

direct access to a protected member of a base class

Use:

```
D obj1;
obj1.a;      X
obj1.f1();   ✓
```

There are two ways a define a member function (a method)

1. single file

```
class A {
  ...
public:
  void f1() {...}
};
```

2. two files:

```
class A {
  ...
public:
  void f1();
};
```

```
void A::f1() { ... }
```

scope resolution operator

Typically declarations are put in .h files and these are #included in the other (definitions) file.

Java has to define all methods "inline". Java avoids long files by only allowing one class per file.

* Up and down-casting

Public Inheritance creates a subtype relationship.

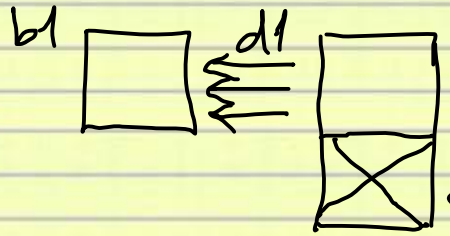
```
class B {
    ...
};
class D : public B {
    ...
};
```

```
B b1;
D d1;
b1 = d1;
```

This is allowed because of the subtype principle.

"An object of a derived type may substitute for an object of a base type"

In C++, the following happens:

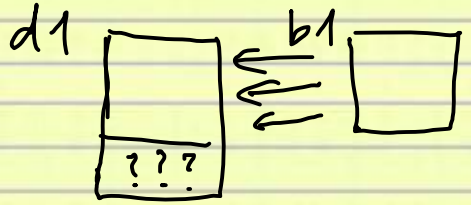


Object slicing

← part of object that belongs to class D

Let's try the inverse:

```
d1 = b1;
```



The first is called upcasting (derived → base). The second is downcasting and is not allowed.

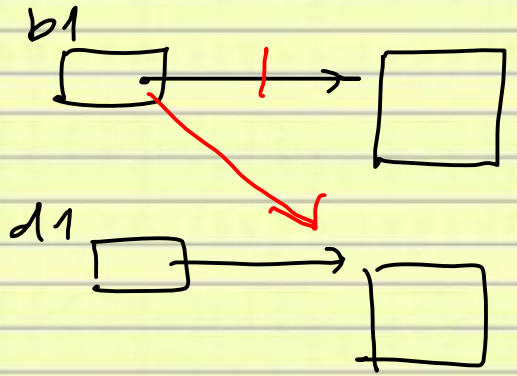
We could try explicit casting :

```
b1 = (B) d1;   ✓
d1 = (D) b1;   ✗
```

In Java, we only copy references with assignment

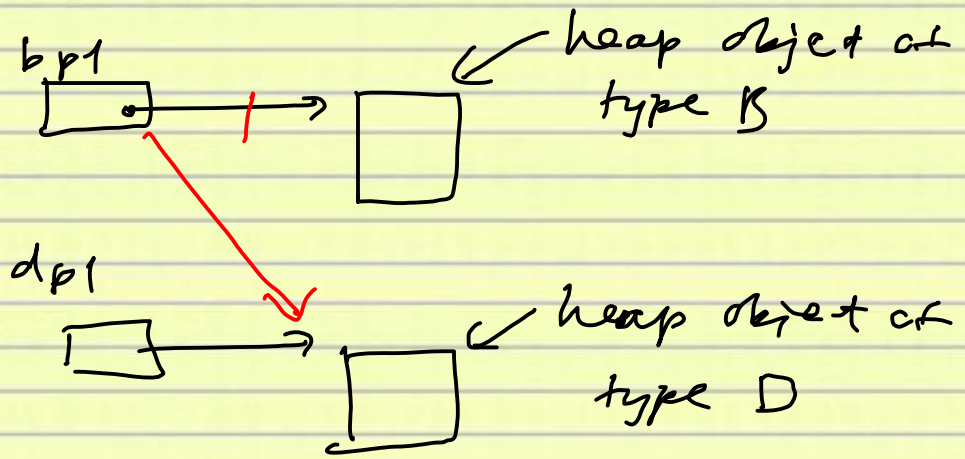
```
B b1 = new B();
D d1 = new D();

b1 = d1;
```



This is more complex with explicit ptrs. in C++.

```
B *bp1 = new B(); // new returns a ptr.
D *dp1 = new D();
bp1 = (B *) dp1;
```



```
B * bp1;
D * dp1 = new D();
bp1 = dp1;
D *dp2 = dynamic_cast<D *>(bp1);
if (dp2 == 0)
    ... cast failed
else
    dp2->f1();
```