

Inheritance

- Many people are of the opinion that **inheritance** is what object-oriented programming is all about.

35. “**public**” Inheritance Models “isa”

- *class Person {...};*
*class Student : **public** Person {...};*
- *void dance(const Person& p);* // anyone can dance
void study(const Student& s); // only students study
- *Person p;* // *p* is a *Person*
Student s; // *s* is a *Student*
dance(p); // fine, *p* is a *Person*
dance(s); // fine, *s* is a *Student*
 // and a *Student* **isa** *Person*
study(s); // fine
study(p); // error! *p* is not a *Student*

36. Interface & Implementation

- Declaring a **pure virtual function** is to have derived classes inherit a function **interface** only.
- Declaring a **simple virtual function** is to have derived classes inherit a function **interface** as well as a **default** implementation.
- Declaring a **nonvirtual function** is to have derived classes inherit a function **interface** as well as a **mandatory** implementation.

37. Inherited Non-Virtual Function

- Never redefine an inherited nonvirtual function.
- ***class B { public: void **mf()**; };***
class D: public B { void **mf()**; };
- ***D x;*** // ***x*** is an object of type ***D***
B *pB = &x; // get pointer to ***x***
pB->**mf()**; // call ***mf*** through pointer
D *pD = &x; // get pointer to ***x***
pD ->**mf()**; // call ***mf*** through pointer
- Two-faced behavior...

38. Inherited Default Parameters

- Never redefine an inherited default parameter value (of a virtual function, of course).
- ***class B {***
public:
virtual void f(int i=1) const = 0;
};
- ***class D: public B {***
void f(int i=2) const;
};

Inherited Default Parameters (2)

- ***B *pd = new D ;***
pd->f(); // calls ***$D::f(2)$*** or ***$D::f(1)$*** ?
- Virtual functions are dynamically bound, but default parameters are ***statically*** bound.

Inherited Default Parameters (3)

- A virtual function call uses the **default arguments** in the declaration of the virtual function determined by the **static** type of the pointer or reference denoting the object.
- An overriding function in a derived class does **not** acquire default arguments from the function it overrides.

39. Down Casting

- Avoid casts down the inheritance hierarchy.
- ***class B {};***
- ***class D: public B {
public:
 void f();
};***

Down Casting (2)

- ***template<class T>***
class A {
public:
T x;
};
- ***A AB;***
B *p0 = &(AB.x), *p1 = new D;
p0->f(); // error!
p1->f(); // okay
static_cast<D*>(p0)->f();
// works, but leads to a nightmare.

Down Casting (3)

- *D d;*
B &r1 = d, r2 = d;
static_cast<D>(r1).f(); // ??
static_cast<D>(r2).f(); // ??
- *class E: public B {*
public:
void f();
};

Down Casting (4)

- You need to write code like:

*if (*p points to a D)*

static_cast<D>(p)->f();*

else

static_cast<E>(p)->f();*

41. Inheritance & Template

- You need a collection of classes with many shared characteristics.
- Should you use "inheritance" and have all the classes derived from a common base class?
- Or should you use "templates" and have them all generated from a common code skeleton?

Inheritance & Template (cont)

- A template should be used to generate a collection of classes when the **type** of the objects does **not** affect the behavior of the class's functions.
- Inheritance (and virtual functions) should be used for a collection of classes when the **type** of the objects **does** affect the behavior of the class's functions.

42. Private Inheritance

- Members inherited from a **private** base class become private members of the derived class, even if they were protected or public in the base class.
- In contrast to public inheritance, compilers will generally "**not**" convert a derived class object into a base class object if the inheritance relationship between the classes is **private**.

Private Inheritance (2)

- *class Person {...};*
*class Student : **private** Person {...};*
- *void dance(const Person& p);* // anyone can dance
void study(const Student& s); // only students study
- *Person p;* // *p* is a *Person*
Student s; // *s* is a *Student*
dance(p); // fine, *p* is a *Person*
dance(s); // error! A *Student* is **not** a *Person*

Private Inheritance (3)

- Private inheritance means that **implementation** only should be inherited; **interface** should be ignored.
- If D privately inherits from B, D objects are implemented ***in terms of*** B objects; **no** conceptual relationship exists between objects of types B and D.

43. Multiple Inheritance

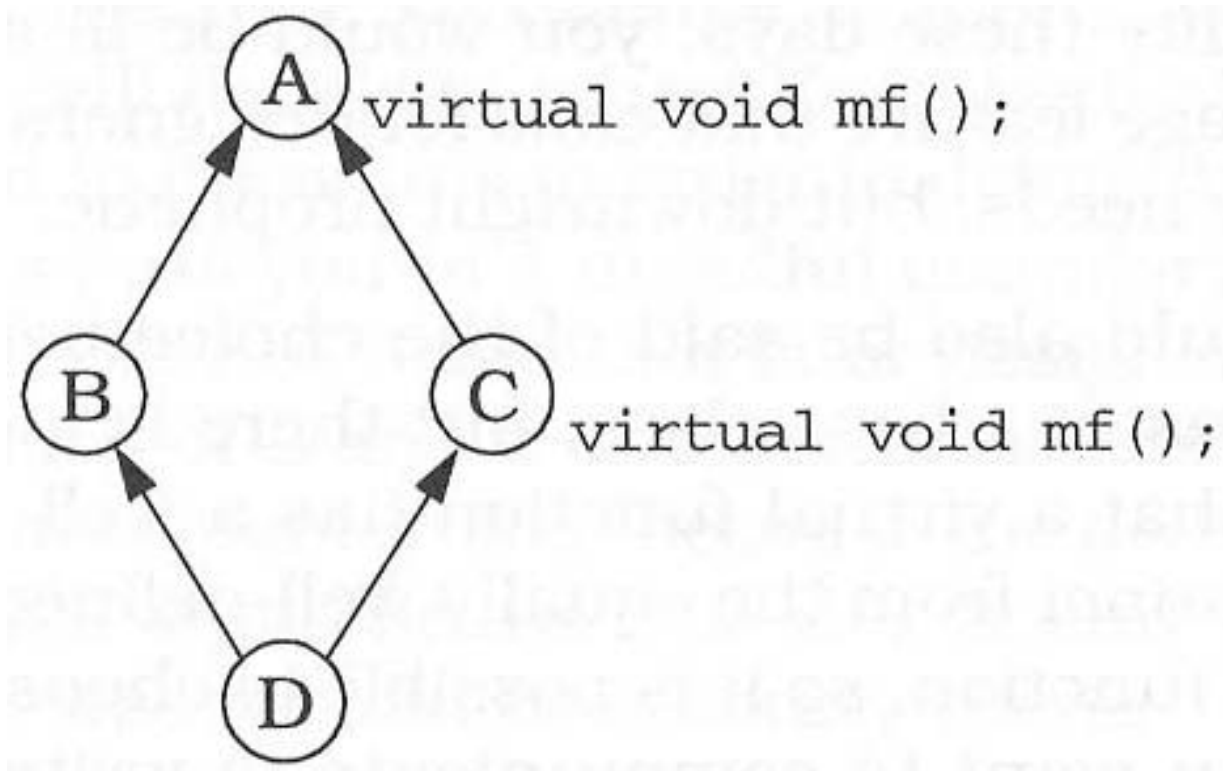
- *class A {
 virtual void mf();
};*
- *class B: public A {};*
 // or *class B: virtual public A {};*

Multiple Inheritance (2)

- *class C: public A {
 // or class C: **virtual** public A {
 virtual void mf(); // redefinition of *mf*
};*
- *class D: public B, public C {};*

Multiple Inheritance (3)

- *D *pd = new D;*
pd->mf(); // *A::mf* or *C::mf*?



Multiple Inheritance (4)

- If A is a nonvirtual base of B or C , the call is ambiguous.
- If A is **virtual** base of both B and C , the call is $C::mf$

45. When is an empty class not an empty class?

- Know what functions C++ silently writes and calls.
- If you write this:

class Empty{};

- It is the same as if you have written this (note these functions are generated only if **they are needed**):

When is an empty class not an empty class? (2)

```
class Empty {  
public:  
    Empty() {};                                // default constructor  
    Empty(const Empty& rhs);                    // copy constructor  
    ~Empty() {};                                // destructor  
  
    Empty& operator=(const Empty& rhs);  
        // assignment operator  
  
    Empty* operator&() { return this; };  
        // address-of operators  
    const Empty* operator&() const;  
};
```

When is an empty class not an empty class? (3)

- The following code will cause each function to be generated:

```
const Empty e1;      // default constructor  
                      // destructor  
Empty e2(e1);        // copy constructor  
e2 = e1;            // assignment operator  
Empty *pe2= &e2;    // address-of-operator (non-const)  
const Empty *pe1 = &e1;  
                    // address-of operator (const)
```

Last Example

```
class Base {  
    public: virtual void f(int x); };  
class Derived: public Base {  
    public: virtual void f(double *pd); };
```

```
Derived *pd = new Derived;  
pd->f(10);    // error!
```