Virtual Functions

Ho Dac Hung

Virtual Functions

- Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class.

Normal Member Functions Accessed with Pointers

```
class Base //base class
{
    public:
        void show() //normal function
        { cout << “Base\n”; }
};
```

Normal Member Functions Accessed with Pointers

```
class Derv1 : public Base //derived class 1
{
    public:
        void show()
        { cout << “Derv1\n”; }
};
```

Normal Member Functions Accessed with Pointers

```
class Derv2 : public Base //derived class 2
{
    public:
        void show()
        { cout << “Derv2\n”; }
};
```

Normal Member Functions Accessed with Pointers

```
int main()
{
    Derv1 dv1; //object of derived class 1
    Derv2 dv2; //object of derived class 2
    Base* ptr; //pointer to base class
    ptr = &dv1; //put address of dv1 in pointer
    ptr->show(); //execute show()
    ptr = &dv2; //put address of dv2 in pointer
    ptr->show(); //execute show()
    return 0;
}
```

Normal Member Functions Accessed with Pointers

```
class Base //base class
{
    public:
        virtual void show() //normal function
        { cout << “Base\n”; }
};
```

Abstract Classes and Pure Virtual Functions

- When we will never want to instantiate objects of a base class, we call it an abstract class. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects. It may also provide an interface for the class hierarchy.
- We create abstract class by placing at least one pure virtual function in the base class. A pure virtual function is one with the expression =0 added to the declaration.

Abstract Classes and Pure Virtual Functions

```
class Base //base class
{
    public:
        virtual void show() = 0; //pure virtual function
};
```

Abstract Classes and Pure Virtual Functions

```
class Derv1 : public Base //derived class 1
{
    public:
        void show()
        { cout << “Derv1\n”; }
};
```

Abstract Classes and Pure Virtual Functions

```
class Derv2 : public Base //derived class 2
{
    public:
        void show()
       { cout << “Derv2\n”; }
};
```

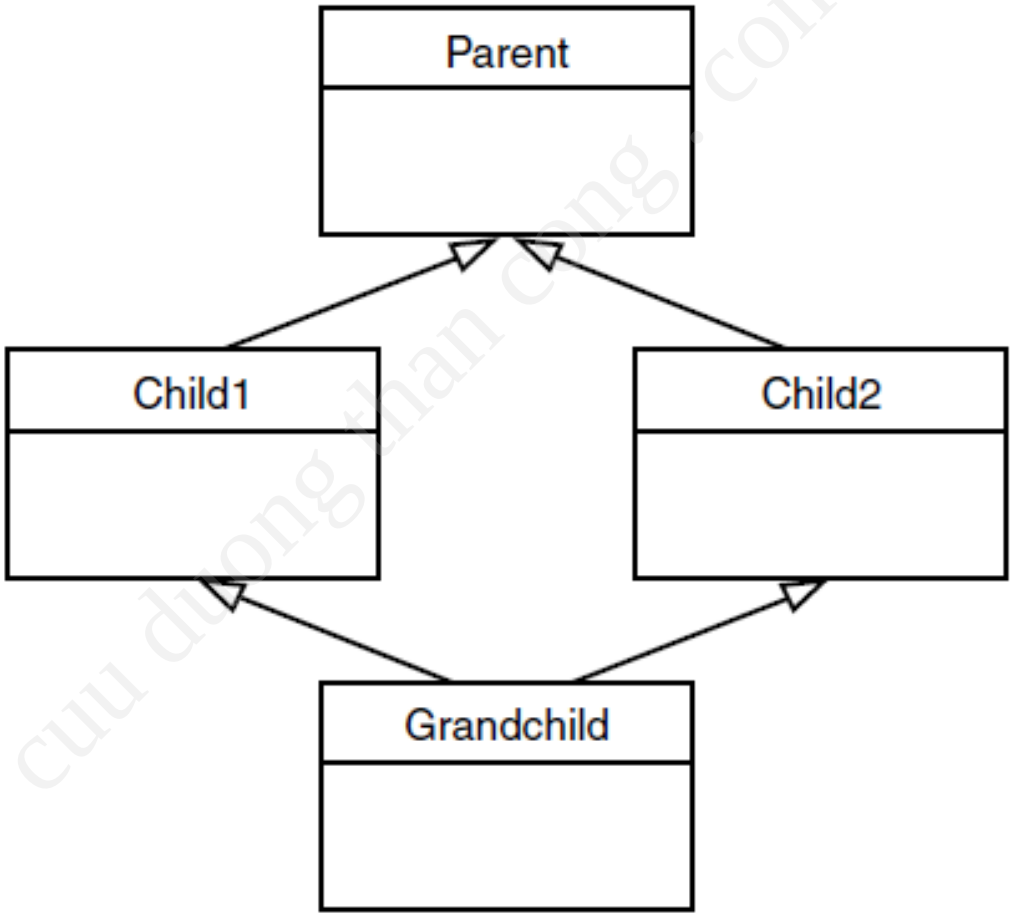
Abstract Classes and Pure Virtual Functions

```
int main()
{
    // Base bad; //can’t make object from abstract class
    Base* arr[2]; //array of pointers to base class
    Derv1 dv1; //object of derived class 1
    Derv2 dv2; //object of derived class 2
    arr[0] = &dv1; //put address of dv1 in array
    arr[1] = &dv2; //put address of dv2 in array
    arr[0]->show(); //execute show() in both objects
    arr[1]->show();
    return 0;
}
```

Virtual Base Classes

- In this arrangement a problem can arise if a member function in the Grandchild class wants to access data or functions in the Parent class.

Virtual Base Classes



```
class Parent
{
    protected:
        int basedata;
};
class Child1 : public Parent
{ };
class Child2 : public Parent
{ };
class Grandchild : public Child1, public Child2
{
    public:
        int getdata()
        { return basedata; } // ERROR: ambiguous
};
```

```
class Parent
{
    protected:
        int basedata;
};
class Child1 : virtual public Parent
{ };
class Child2 : virtual public Parent
{ };
class Grandchild : public Child1, public Child2
{
    public:
        int getdata()
        { return basedata; } // ERROR: ambiguous
};
```

Friend Functions

- The concepts of encapsulation and data hiding dictate that nonmember functions should not be able to access an object's private or protected data. The policy is, if you're not a member, you can't get in. However, there are situations where such rigid discrimination leads to considerable inconvenience.

Friends as Bridges

- Imagine that you want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. In this situation there's nothing like a friend function.

```
class beta; //needed for frifunc declaration
class alpha
{
    private:
        int data;
    public:
        alpha() : data(3) { } //no-arg constructor
        friend int frifunc(alpha, beta); //friend function
};
```

```
class beta
{
    private:
        int data;
    public:
beta() : data(7) { } //no-arg constructor
friend int frifunc(alpha, beta); //friend function
};
int frifunc(alpha a, beta b) //function definition
{
return( a.data + b.data );
}
```

```
int main()
{
alpha aa;
beta bb;
cout << frifunc(aa, bb) << endl; //call the function
return 0;
}
```

Breaching the Walls

- We should note that friend functions are controversial. During the development of C++, arguments raged over the desirability of including this feature. On the one hand, it adds flexibility to the language; on the other, it is not in keeping with data hiding, the philosophy that only member functions can access a class's private data.

friends for Functional Notation

- Sometimes a friend allows a more obvious syntax for calling a function than does a member function.

```
class Distance //English Distance class
{
    private:
        int feet;
        float inches;
    public:
        Distance() : feet(0), inches(0.0)
        { }
        Distance(int ft, float in) : feet(ft), inches(in)
        { }
        void showdist() //display distance
        { cout << feet << "\'-" << inches << '\"'; }
        friend float square(Distance); //friend function
};
```

```
float square(Distance d) //return square of
{ //this Distance
    float fltfeet = d.feet + d.inches/12; //convert to float
    float feetsqrd = fltfeet * fltfeet; //find the square
    return feetsqrd; //return square feet
}
int main()
{
    Distance dist(3, 6.0); //two-arg constructor (3'-6")
    float sqft;
    sqft = square(dist); //return square of dist
    cout << "\nDistance = "; dist.showdist();
    cout << "\nSquare = " << sqft << " square feet\n";
    return 0;
}
```

friend Classes

- The member functions of a class can all be made friends at the same time when you make the entire class a friend

```
class alpha
{
    private:
        int data1;
    public:
        alpha() : data1(99) { } //constructor
        friend class beta; //beta is a friend class
};
class beta
{ //all member functions can
    public: //access private alpha data
        void func1(alpha a) { cout << “\ndata1=” << a.data1; }
        void func2(alpha a) { cout << “\ndata1=” << a.data1; }
};
```

```
int main()
{
    alpha a;
    beta b;
    b.func1(a);
    b.func2(a);
    cout << endl;
    return 0;
}
```

Static Functions

- A static data member is not duplicated for each object; rather a single data item is shared by all objects of a class. Let's extend this concept by showing how functions as well as data may be static.

```
class gamma
{
    private:
        static int total; //total objects of this class
        // (declaration only)
        int id; //ID number of this object
    public:
        gamma() //no-argument constructor
        {
            total++; //add another object
            id = total; //id equals current total
        }
```

```
~gamma() //destructor
{
    total--;
    cout << “Destroying ID number “ << id << endl;
}
static void showtotal() //static function
{
    cout << “Total is “ << total << endl;
}
void showid() //non-static function
{
    cout << “ID number is “ << id << endl;
}
};
```

```
int gamma::total = 0;
int main()
{
    gamma g1;
    gamma::showtotal();
    gamma g2, g3;
    gamma::showtotal();
    g1.showid();
    g2.showid();
    g3.showid();
    cout << “----------end of program----------\n”;
    return 0;
}
```

Accessing static Functions

- We shouldn't need to refer to a specific object when we're doing something that relates to the entire class. It's more reasonable to use the name of the class itself with the scope-resolution operator.

```
gamma::showtotal(); // more reasonable
```

The this Pointer

- The member functions of every object have access to a sort of magic pointer named this, which points to the object itself.

```
class where
{
    private:
        char charray[10]; //occupies 10 bytes
    public:
        void reveal()
        { cout << “\nMy object’s address is “ << this; }
};
int main()
{
    where w1, w2;
    w1.reveal(); //see where they are
    w2.reveal();
    return 0;
}
```

Accessing Member Data with this

- When you call a member function, it comes into existence with the value of this set to the address of the object for which it was called. The this pointer can be treated like any other pointer to an object, and can thus be used to access the data in the object it points to.

```
class what
{
    private:
        int alpha;
    public:
        void tester()
        {
            this->alpha = 11; //same as alpha = 11;
            cout << this->alpha; //same as cout << alpha;
        }
};
```