W 9.2

Virtual Functions and Polymorphism
VIRTUAL FUNCTIONS and POLYMORPHISM

• Polymorphism, the ability for objects of different classes related by inheritance to use a function of the same name but with different behaviour is facilitated by the use of *virtual functions*.

• When an invocation is made through a base class pointer to use a virtual function, C++ uses the correct redefined function in the appropriate derived class associated with the object.
Using Polymorphism

• Suppose we want to draw a picture which is composed of several objects.

• One way of doing it might be to create an array of pointers to the various elements and call the draw() function for each in turn.

```cpp
Shape* ptrarr[100];
for (int j=0; j<N; j++)
    ptrarr[j]->draw();
```
Using Polymorphism (cont.)

• This means that when pointer ptrarr points at a square a square is drawn, triangles and circles likewise.

• Must meet some conditions to do this.
  – All different classes must be derived from common base class.
  – Draw function must be declared to be virtual in base class.
Using Polymorphism (cont.)

• Lets look at some examples to see how this may be achieved.

• Lets look at an inheritance hierarchy with a common function \textit{show}().
//notvirt.cpp
//normal functions accessed from pointers
#include <iostream.h>

class Base{
public:
    void show(){cout<<"In Base\n";}
};

class Derive1 : public Base{
public:
    void show(){cout<<"\n In Derive1\n";}
};

class Derive2 : public Base{
public:
    void show(){cout<<"\n In Derive2\n";}
};

void main(){
    Derive1 dv1;
    Derive2 dv2;
    Base* ptr;
    ptr= &dv1;
    ptr->show();
    ptr= &dv2;
    ptr->show();
}
Accessing Member Functions

• In the above example, we tried to access a derived class function, so what happened.

• Problem 1
  – \( \text{ptr} = \&\text{dv1} \) is attempting to assign the address of one type (\text{Derive1}) to a pointer of another (\text{Base})..
  – Actually this Ok as type checking has been relaxed.
  – Pointers to objects of derived class are type compatible with pointers to objects of base
Accessing Member Functions (cont.)

- Which function then was called?
- Actually it was always the base class function, not the derived class functions as we may have intended.
- The compiler ignores the \textit{contents} of the pointer and chooses the member function that matches the \textit{type} of the pointer.
Now use a Virtual Function

• Make one change only to the above program
  – place the keyword virtual in front of the declaration for show() in the base class.
  – virtual void show(){cout<<"In Base\n";}

• The output will now be

  Now the derived class function is called, as would be intended

  In Derive1
  In Derive2
//notvirt.cpp
//normal functions accessed from pointers
#include <iostream.h>
class Base{
public:
  virtual void show(){cout<<"In Base\n";}
};
class Derive1 : public Base{
public:
  void show(){cout<<"\n In Derive1\n";}
};
class Derive2 : public Base{
public:
  void show(){cout<<"\n In Derive2\n";}
};
void main(){
  Derive1 dv1;
  Derive2 dv2;
  Base* ptr;
  ptr= &dv1;
  ptr->show();
  ptr= &dv2;
  ptr->show();
}
Virtual Members Accessed with Pointers

- The members of the derived classes, not base classes executed.
- Rule is that the compiler selects the function based on the contents of the pointer, not just the type as before.
- Rules changed because we declared the function as virtual.
Pure Virtual Functions

• In the next example, there is a pure virtual function.
  – virtual void show()=0;

• There is no body to the function, the =0 syntax indicates to the compiler that we never intend to run this function here. We run only the versions in the derived classes.
Pure Virtual Functions (cont.)

- The compiler will not know until execution time which function to run. This is called dynamic binding or late binding.
//virt.cpp
#include <iostream.h>

class Base{
public:
    virtual void show()=0;
};

class Derive1 : public Base{
public:
    void show(){cout<<"\n In Derive1\n";}
};

class Derive2 : public Base{
public:
    void show(){cout<<"\n In Derive2\n";}
};

void main(){
    Derive1 dv1;
    Derive2 dv2;
    Base* list[2];
    list[0]= &dv1;
    list[1]=&dv2;
    list[0]->show();
    list[1]->show();
}
Abstract and Concrete Classes

• Some classes are better never instantiated.
• Abstract base classes are used as base classes for use in inheritance hierarchies.
• Concrete classes are classes which may be instantiated.
Graded Exercises

• Check out summary and other material of Ch. 10 (pp 654..657)
• Answer Exercises 10.5, 10.6
• Run the code for Fig. 10.1 in the book & satisfy yourself that you understand it. Get help from tutor as necessary.