Class Hierarchies

Inheritance has been used to add functionality to an existing class. Now let's look at an example where inheritance is used for a different purpose: as part of the original design of a program.

Our example models a database of employees of a widget company. We've simplified the situation so that only three kinds of employees are represented. Managers manage, scientists perform research to develop better widgets, and laborers operate the dangerous widget-stamping presses.

The database stores a name and an employee identification number for all employees. However, for managers, it also stores their titles and golf club dues. For scientists, it stores the number of scholarly articles they have published. Laborers need no additional data beyond their names and numbers.

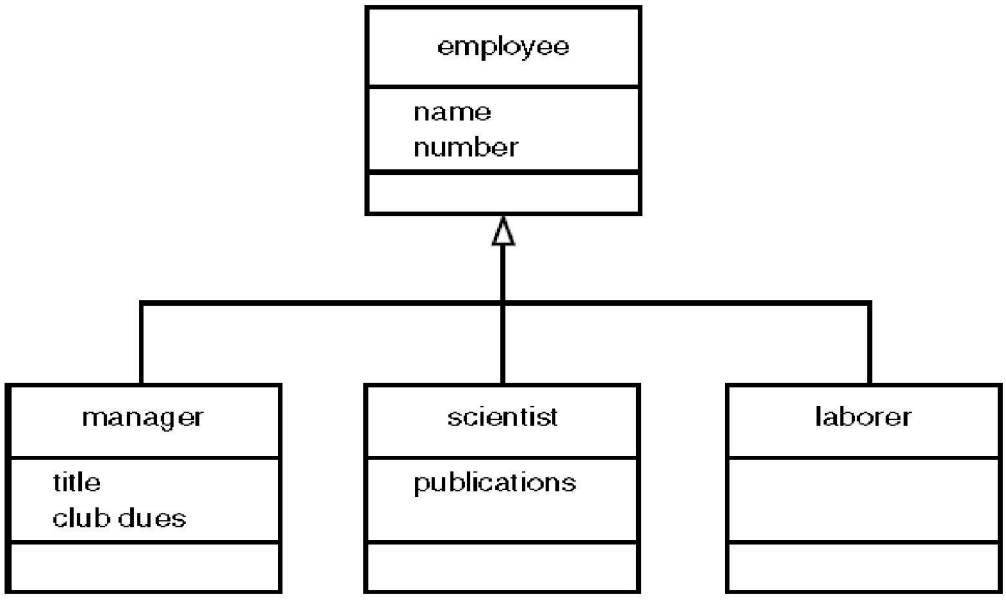

Figure 1: ***class diagram for EMPLOY.***

Example 1:-Write an oo program to model employ database using inheritance.

```
#include <iostream.h>
const int LEN = 80; //maximum length of names
////////////////////////////////////////////////////////////////
class employee //employee class
{
private:
char name[LEN]; //employee name
unsigned long number; //employee number
public:
void getdata()
{
cout << "\n Enter last name: "; cin >> name;
cout << " Enter number: "; cin >> number;
}
void putdata() const
{
cout << "\n Name: " << name;
cout << "\n Number: " << number;
}
};
////////////////////////////////////////////////////////////////
class manager : public employee  //management class
{
private:
char title[LEN];       //"vice-president" etc.
double dues;           //golf club dues
public:
void getdata()
{
employee::getdata();
cout << " Enter title: "; cin >> title;
cout << " Enter golf club dues: "; cin >> dues;
}
void putdata() const
{
employee::putdata();
cout << "\n Title: " << title;
cout << "\n Golf club dues: " << dues;
}
};
////////////////////////////////////////////////////////////////
class scientist : public employee     //scientist class
{
private:
int pubs;           //number of publications
public:
void getdata()
{
employee::getdata();
cout << " Enter number of pubs: "; cin >> pubs;
}
void putdata() const
{
employee::putdata();
cout << "\n Number of publications: " << pubs;
}
};
////////////////////////////////////////////////////////////////
class laborer : public employee     //laborer class
{
};
```

```
//////////////////////////////////////////////////////////////////
void main()
{
manager m1, m2;
scientist s1;
laborer l1;
cout << endl;      //get data for several employees
cout << "\nEnter data for manager 1";
m1.getdata();
cout << "\nEnter data for manager 2";
m2.getdata();
cout << "\nEnter data for scientist 1";
s1.getdata();
cout << "\nEnter data for laborer 1";
l1.getdata();
                  //display data for several employees
cout << "\nData on manager 1";
m1.putdata();
cout << "\nData on manager 2";
m2.putdata();
cout << "\nData on scientist 1";
s1.putdata();
cout << "\nData on laborer 1";
l1.putdata();
}
```

The main() part of the program declares four objects of different classes: two managers, a scientist, and a laborer. (Of course many more employees of each type could be defined, but the output would become rather large.) It then calls the getdata() member functions to obtain information about each employee, and the putdata() function to display this information.

Output of program

Enter data for manager 1

Enter last name: Wainsworth

Enter number: 10

Enter title: President

Enter golf club dues: 1000000

Enter data on manager 2

Enter last name: Bradley

Enter number: 124

Enter title: Vice-President

Enter golf club dues: 500000

Enter data for scientist 1

Enter last name: Hauptman-Frenglish

Enter number: 234234

Enter number of pubs: 999

Enter data for laborer 1

Enter last name: Jones

Enter number: 6546544

The program then plays it back.

Data on manager 1

Name: Wainsworth

Number: 10

Title: President

Golf club dues: 1000000

Data on manager 2

Name: Bradley

Number: 124

Title: Vice-President

Golf club dues: 500000

Data on scientist 1

Name: Hauptman-Frenglish

Number: 234234

Number of publications: 999

Data on laborer 1

Name: Jones

Number: 6546544

"Abstract" Base Class

. It may seem that the laborer class is unnecessary, but by making it a separate class we emphasize that all classes are descended from the same source, employee. Also, if in the future we decided to modify the laborer class, we would not need to change the declaration for employee.

Constructors and Member Functions

There are no constructors in either the base or derived classes, so the compiler creates objects of the various classes automatically when it encounters definitions like manager m1, m2; using the default constructor for manager calling the default constructor for employee.

The getdata() and putdata() functions in employee accept a name and number from the user and display a name and number. Functions also called getdata() and putdata() in the manager and scientist classes use the functions in employee, and also do their own work.

In manager, the getdata() function asks the user for a title and the amount of golf club dues, and putdata() displays these values. In scientist, these functions handle the number of publications.

Access Combinations

There are so many possibilities for access that it's instructive to look at an example programthat shows what works and what doesn't. Here's the listing for Example 2:

Example 2:-

```
#include <iostream.h>
//////////////////////////////////////////////////////////////////
class A //base class
{
private:
int privdataA;    //(functions have the same access
protected:        //rules as the data shown here)
int protdataA;
public:
int pubdataA;
};
//////////////////////////////////////////////////////////////////
class B : public A   //publicly-derived class
{
public:
void funct()
{
int a;
a = privdataA;    //error: not accessible
a = protdataA;    //OK
a = pubdataA;     //OK
}
};
//////////////////////////////////////////////////////////////////
class C : private A //privately-derived class
{
public:
void funct()
{
int a;
a = privdataA;    //error: not accessible
a = protdataA;    //OK
a = pubdataA;     //OK
}
};
```

```
//////////////////////////////////////////////////////////////////
void main()
{
int a;
B objB;
a = objB.privdataA;    //error: not accessible
a = objB.protdataA;    //error: not accessible
a = objB.pubdataA;     //OK (A public to B)
C objC;
a = objC.privdataA; //error: not accessible
a = objC.protdataA; //error: not accessible
a = objC.pubdataA; //error: not accessible (A private to C)
}
```

The program specifies a base class, A, with private, protected, and public data items. Two classes, B and C, are derived from A. B is publicly derived and C is privately derived.

As we've seen before, functions in the derived classes can access protected and public data in the base class. Objects of the derived classes cannot access private or protected members of the base class.

What's new is the difference between publicly derived and privately derived classes. Objects of the publicly derived class B can access public members of the base class A, while objects of the privately derived class C cannot; they can only access the public members of their own derived class. This is shown in Figure 2.

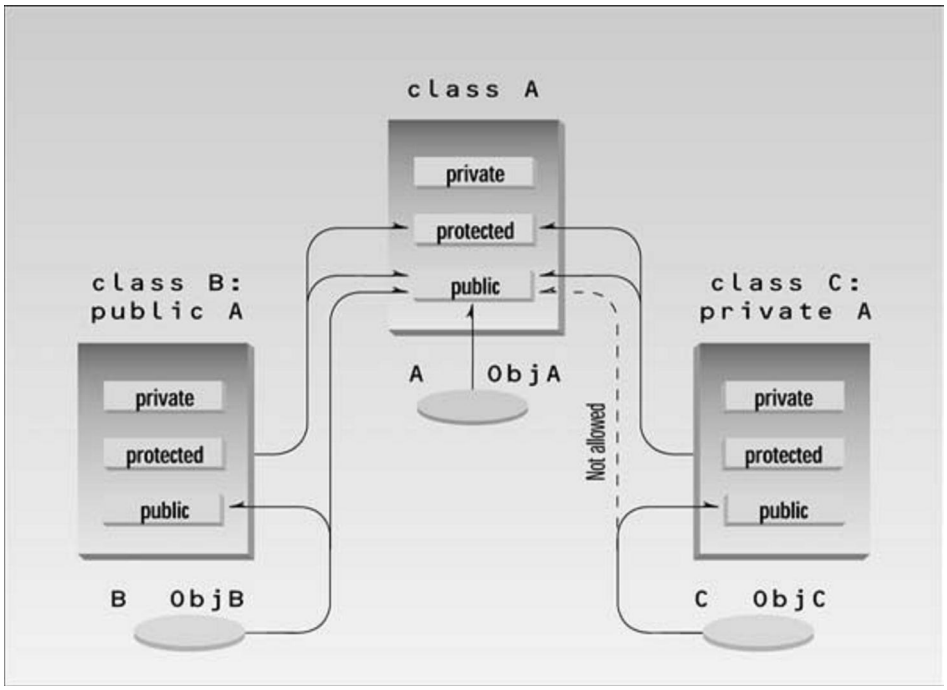

Figure2: ***Public and private derivation.***

Levels of Inheritance

Classes can be derived from classes that are themselves derived. Here's a miniprogram that shows the idea:

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

Here B is derived from A, and C is derived from B. The process can be extended to an arbitrary number of levels D could be derived from C, and so on. Suppose that we decided to add a special kind of laborer called a *foreman* to the EMPLOY program. Since a foreman is a kind of laborer, the foreman class is derived from the laborer class, as shown in Figure 3.

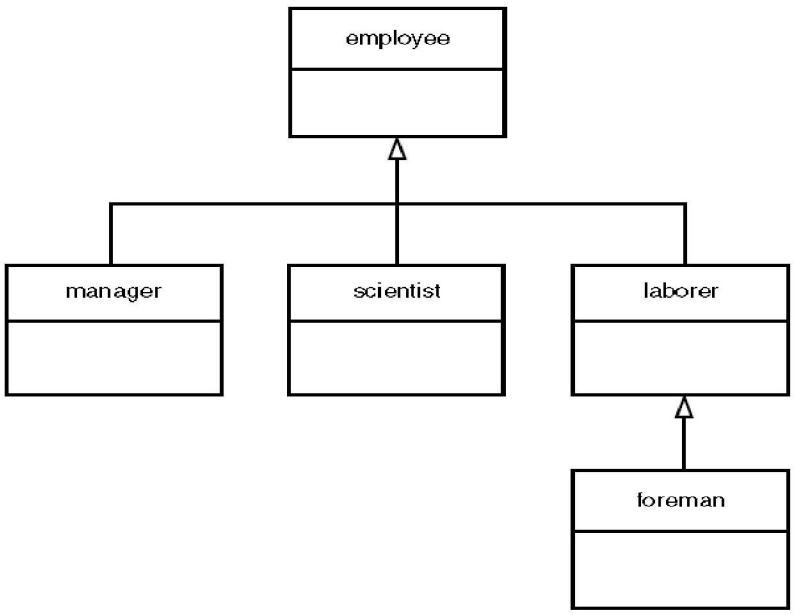

Figure 3: ***class diagram for EMPLOY2.***

Example 3:-Write an oo program to model employ database using multi levels inheritance.

```
#include <iostream.h>
const int LEN = 80; //maximum length of names
////////////////////////////////////////////////////////////////
class employee
{
private:
char name[LEN]; //employee name
unsigned long number; //employee number
public:
void getdata()
{
cout << "\n Enter last name: "; cin >> name;
cout << " Enter number: "; cin >> number;
}
void putdata() const
{
cout << "\n Name: " << name;
cout << "\n Number: " << number;
}
};
////////////////////////////////////////////////////////////////
class manager : public employee //manager class
{
private:
char title[LEN];    //"vice-president" etc.
double dues;        //golf club dues
public:
void getdata()
{
employee::getdata();
cout << " Enter title: "; cin >> title;
cout << " Enter golf club dues: "; cin >> dues;
}
void putdata() const
{
employee::putdata();
cout << "\n Title: " << title;
cout << "\n Golf club dues: " << dues;
}
};
////////////////////////////////////////////////////////////////
class scientist : public employee    //scientist class
{
private:
int pubs;        //number of publications
public:
void getdata()
{
employee::getdata();
cout << " Enter number of pubs: "; cin >> pubs;
}
void putdata() const
{
employee::putdata();
cout << "\n Number of publications: " << pubs;
}
};
```

```
////////////////////////////////////////////////////////////////
class laborer : public employee    //laborer class
{
};
////////////////////////////////////////////////////////////////
class foreman : public laborer     //foreman class
{
private:
float quotas;      //percent of quotas met successfully
public:
void getdata()
{
laborer::getdata();
cout << " Enter quotas: "; cin >> quotas;
}
void putdata() const
{
laborer::putdata();
cout << "\n Quotas: " << quotas;
}
};
////////////////////////////////////////////////////////////////
int main()
{
laborer l1;
foreman f1;
cout << endl;
cout << "\nEnter data for laborer 1";
l1.getdata();
cout << "\nEnter data for foreman 1";
f1.getdata();
cout << endl;
cout << "\nData on laborer 1";
l1.putdata();
cout << "\nData on foreman 1";
f1.putdata();
cout << endl;
return 0;
}
```

A class hierarchy results from generalizing common characteristics. The more general the class, the higher it is on the chart. Thus a laborer is more general than a foreman, who is a specialized kind of laborer, so laborer is

shown above foreman in the class hierarchy, although a foreman is probably paid more than a laborer.

Multiple Inheritances

A class can be derived from more than one base class. This is called *multiple inheritances*.

Figure 4 shows how this looks when a class C is derived from base classes A and B.

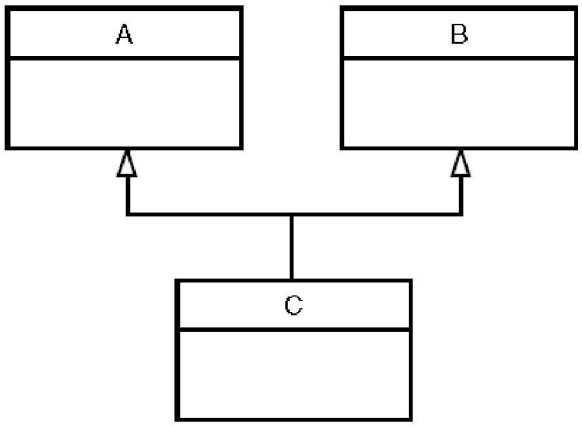

Figure 4: ***class diagram for multiple inheritances.***

The syntax for multiple inheritances is similar to that for single inheritance. In the situation shown in Figure 4, the relationship is expressed like this:

```
class A // base class A
{
};
class B // base class B
{
};
class C : public A, public B // C is derived from A and B
{
};
```

The base classes from which C is derived are listed following the colon in C's specification; they are separated by commas.

Member Functions in Multiple Inheritance

Suppose that we need to record the educational experience of some of the employees in the EMPLOY program. We've already developed a class called student that models students with different educational backgrounds. We decide that instead of modifying the employee class to incorporate educational data, we will add this data by multiple inheritances from the student class. The student class stores the name of the school or university last attended and the highest degree received. Both these data items are stored as strings. Two member functions, getedu() and putedu(), ask the user for this information and display it. Educational information is not relevant to every class of employee. We don't need to record the educational experience of laborers; it's only relevant for managers and scientists. We therefore modify manager and scientist so that they inherit from both the employee and student classes, as shown in Figure 5.

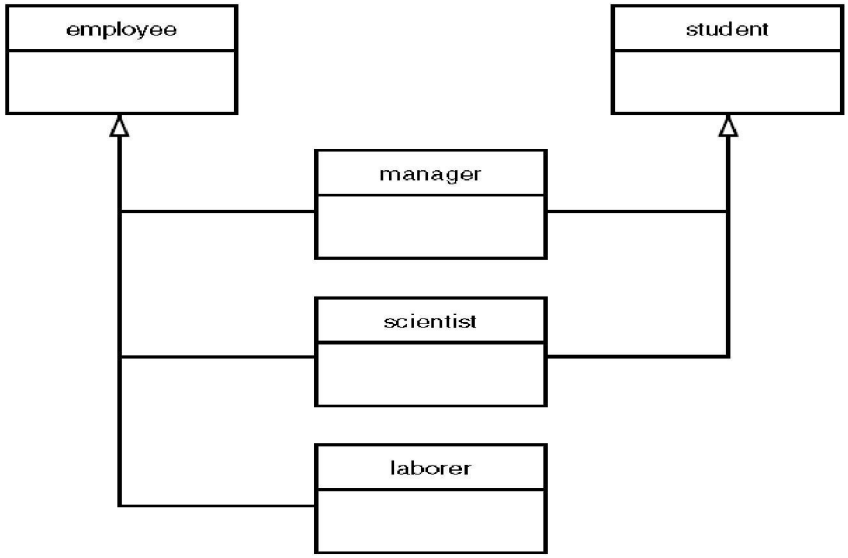

Figure5 :*UML class diagram for EMPMULT.*

Example 4:-Write an oo program to model employ database using multiple inheritances without using constructors.

```
#include <iostream.h>
const int LEN = 80; //maximum length of names
////////////////////////////////////////////////////////////////
class student //educational background
{
private:
char school[LEN]; //name of school or university
char degree[LEN]; //highest degree earned
public:
void getedu()
{
cout << " Enter name of school or university: ";
cin >> school;
cout << " Enter highest degree earned \n";
cout << " (Highschool, Bachelor's, Master's, PhD): ";
cin >> degree;
}
void putedu() const
{
cout << "\n School or university: " << school;
cout << "\n Highest degree earned: " << degree;
}
};
////////////////////////////////////////////////////////////////
class employee
{
private:
char name[LEN];        //employee name
unsigned long number;  //employee number
public:
void getdata()
{
cout << "\n Enter last name: "; cin >> name;
cout << " Enter number: "; cin >> number;
}
void putdata() const
{
cout << "\n Name: " << name;
cout << "\n Number: " << number;
}
};
////////////////////////////////////////////////////////////////
```

```
};
////////////////////////////////////////////////////////////////
class manager : private employee, private student //management
{
private:
char title[LEN];     //"vice-president" etc.
double dues;         //golf club dues
public:
void getdata()
{
employee::getdata();
cout << " Enter title: "; cin >> title;
cout << " Enter golf club dues: "; cin >> dues;
student::getedu();
}
void putdata() const
{
employee::putdata();
cout << "\n Title: " << title;
cout << "\n Golf club dues: " << dues;
student::putedu();
}
};
////////////////////////////////////////////////////////////////
class scientist : private employee, private student //scientist
{
private:
int pubs; //number of publications
public:
void getdata()
{
employee::getdata();
cout << " Enter number of pubs: "; cin >> pubs;
student::getedu();
}
void putdata() const
{
employee::putdata();
cout << "\n Number of publications: " << pubs;
student::putedu();
}
};
```

```
////////////////////////////////////////////////////////////////
class laborer : public employee //laborer
{
};
////////////////////////////////////////////////////////////////
int main()
{
manager m1;
scientist s1, s2;
laborer l1;
cout << endl;
cout << "\nEnter data for manager 1"; //get data for
m1.getdata();              //several employees
cout << "\nEnter data for scientist 1";
s1.getdata();
cout << "\nEnter data for scientist 2";
s2.getdata();
cout << "\nEnter data for laborer 1";
l1.getdata();
cout << "\nData on manager 1";   //display data for
m1.putdata(); //several employees
cout << "\nData on scientist 1";
s1.putdata();
cout << "\nData on scientist 2";
s2.putdata();
cout << "\nData on laborer 1";
l1.putdata();
cout << endl;
return 0;
}
```

The getdata() and putdata() functions in the manager and scientist classes incorporate calls to functions in the student class, such as

student::getedu();

and

student::putedu();

These routines are accessible in manager and scientist because these classes are descended from student.

Here's some sample interaction with Eexample4:

Enter data for manager 1

Enter last name: Bradley

Enter number: 12

Enter title: Vice-President

Enter golf club dues: 100000

Enter name of school or university: Yale

Enter highest degree earned

(Highschool, Bachelor's, Master's, PhD): Bachelor's

Enter data for scientist 1

Enter last name: Twilling

Enter number: 764

Enter number of pubs: 99

Enter name of school or university: MIT

Enter highest degree earned

(Highschool, Bachelor's, Master's, PhD): PhD

Enter data for scientist 2

Enter last name: Yang

Enter number: 845

Enter number of pubs: 101

Enter name of school or university: Stanford

Enter highest degree earned

(Highschool, Bachelor's, Master's, PhD): Master's

Enter data for laborer 1

Enter last name: Jones

Enter number: 48323

Private Derivation in EMPMULT

The manager and scientist classes in EMPMULT are privately derived from the employee and student classes. There is no need to use public derivation because objects of manager and scientist never call routines in the employee and student base classes. However, the laborer class must be publicly

derived from employer, since it has no member functions of its own and relies on those in employee.

Example 5:-Write an oo program to compute distance using constructors in multiple inheritances.

```
#include <iostream.h>
typedef   char *String;
#include <string.h>
////////////////////////////////////////////////////////////////
class Type    //type of lumber
{
private:
String dimensions;
String grade;
public:     //no-arg constructor
Type() : dimensions("N/A"), grade("N/A")
{ }
            //2-arg constructor
Type(String di,String gr) : dimensions(di), grade(gr)
{ }
void gettype()  //get type from user
{
dimensions="2x2";

grade="ABC";
}
void showtype() const //display type
{
cout << "\n Dimensions: " << dimensions;
cout << "\n Grade: " << grade;
}
};
```

```
////////////////////////////////////////////////////////////////
class Distance //English Distance class
{
private:
int feet;
float inches;
public: //no-arg constructor
Distance() : feet(0), inches(0.0)
{ } //constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist() //get length from user
{
cout << " Enter feet: "; cin >> feet;
cout << " Enter inches: "; cin >> inches;
}
void showdist() const //display distance
{ cout << feet << "-" << inches ; }
};
//////////////////////////////////////////////////////////////
```

```
//////////////////////////////////////////////////////////////
class Lumber : public Type, public Distance
{
private:
int quantity; //number of pieces
double price; //price of each piece
public: //constructor (no args)
Lumber() : Type(), Distance(), quantity(0), price(0.0)
{ }
Lumber(String di, String gr,int ft, float in, int qu, float prc ) : Type(di, gr), Distance(ft, in), quantity(qu), price(prc)
{ }
void getlumber()
{
Type::gettype();
Distance::getdist();
cout << " Enter quantity: "; cin >> quantity;
cout << " Enter price per piece: "; cin >> price;
}
void showlumber() const
{
Type::showtype();
cout << "\n Length: ";
Distance::showdist();
cout << "\n Price for " << quantity
<< " pieces: $" << price * quantity;
}
};
//////////////////////////////////////////////////////////////
void main()
{
Lumber siding;   //constructor (no args)
cout << "\nSiding data:\n";
siding.getlumber(); //get siding from user
Lumber studs( "2x4", "const", 8, 0.0, 200, 4.45F ); //constructor (6 args)
cout << "\nSiding"; siding.showlumber();   //display lumber data
cout << "\nStuds"; studs.showlumber();
cout << endl;
}
```

No-Argument Constructor

The no-argument constructor in Type looks like this:

Type()

{ strcpy(dimensions, “N/A”); strcpy(grade, “N/A”); }

This constructor fills in "N/A" (not available) for the dimensions and grade variables so the user will be made aware if an attempt is made to display data for an uninitialized lumber object. You're already familiar with the no-argument constructor in the Distance class:

```
Distance() : feet(0), inches(0.0)
{ }
```

The no-argument constructor in Lumber calls both of these constructors.

```
Lumber() : Type(), Distance(), quantity(0), price(0.0)
{ }
```

The names of the base-class constructors follow the colon and are separated by commas. When the Lumber() constructor is invoked, these base-class constructors—Type() and Distance()— will be executed. The quantity and price attributes are also initialized.

Multi-Argument Constructors

Here is the two-argument constructor for Type:

```
Type(string di, string gr) : dimensions(di), grade(gr)
{ }
```

This constructor copies string arguments to the dimensions and grade member data items. Here's the constructor for Distance, which is again familiar from previous programs:

```
Distance(int ft, float in) : feet(ft), inches(in)
{ }
```

The constructor for Lumber calls both of these constructors, so it must supply values for their arguments. In addition it has two arguments of its own: the quantity of lumber and the unit price.

Thus this constructor has six arguments. It makes two calls to the two constructors, each of which takes two arguments, and then initializes its own two data items. Here's what it looks like:

```
Lumber( string di, string gr, //args for Type
int ft, float in, //args for Distance
int qu, float prc ) :   //args for our data
Type(di, gr),     //call Type ctor
Distance(ft, in), //call Distance ctor
quantity(qu), price(prc) //initialize our data
{ }
```

Ambiguity in Multiple Inheritances

There are two types ambiguity in Multiple Inheritances

1. Two base classes have functions with the same name, while a class derived from both base classes has no function with this name. How do objects of the derived class access the correct base class function? The name of the function alone is insufficient, since the compiler can't figure out which of the two functions is meant.

Example: demonstrates ambiguity in multiple inheritance

```
#include <iostream.h>
class A
{
public:
void show() { cout << "Class A\n"; }
};
class B
```

```
{
public:
void show() { cout << "Class B\n"; }
};
class C : public A, public B
{
};
//////////////////////////////////////////////////////////////
int main()
{
C objC; //object of class C
// objC.show(); //ambiguous--will not compile
objC.A::show(); //OK
objC.B::show(); //OK
return 0;
}
```

The problem is resolved using the scope-resolution operator to specify the class in which the function lies. Thus

objC.A::show();

refers to the version of show() that's in the A class, while

objC.B::show();

refers to the function in the B class.

2. Another kind of ambiguity arises if you derive a class from two classes that are each derived from the same class. This creates a diamond-shaped inheritance tree.

Example: investigates diamond-shaped multiple inheritance

```
#include <iostream.h>
class A
{
public:
void func();
};
class B : public A
{ };
class C : public A
{ };
class D : public B, public C
{ };
//////////////////////////////////////////////////////////////
int main()
{
D objD;
objD.func();      //ambiguous: won't compile
return 0;
}
```

Classes B and C are both derived from class A, and class D is derived by multiple inheritance from both B and C. Trouble starts if you try to access a member function in class A from an object of class D. In this example objD tries to access func(). However, both B and C contain a copy of func(), inherited from A. The compiler can't decide which copy to use, and signals an error.

Function Template

Suppose you want to write a function that returns the absolute value of two numbers. The absolute value of a number is its value without regard to its sign: The absolute value of 3 is 3, and the absolute value of –3 is also 3. Ordinarily this function would be written for a particular data type:

```
int abs(int n) //absolute value of ints
{
return (n<0) ? -n : n; //if n is negative, return -n
}
```

Here the function is defined to take an argument of type int and to return a value of this same type. But now suppose you want to find the absolute value of a type long. You will need to write a completely new function:

```
long abs(long n) //absolute value of longs
{
return (n<0) ? -n : n;
}
```

And again, for type float:

```
float abs(float n) //absolute value of floats
{
return (n<0) ? -n : n;
}
```

The body of the function is written the same way in each case, but they are completely different functions because they handle arguments and return values of different types. It's true that in C++ these functions can all be overloaded to have the same name, but you must nevertheless write a

separate definition for each one. (In the C language, which does not support overloading, functions for different types can't even have the same name. In the C function library this leads to families of similarly named functions, such as abs(), fabs(), labs(), and cabs().

Rewriting the same function body over and over for different types is time-consuming and wastes space in the listing. Also, if you find you've made an error in one such function, you'll need to remember to correct it in each function body. Failing to do this correctly is a good way to introduce inconsistencies into your program. It would be nice if there were a way to write such a function just once, and have it work for many different data types. This is exactly what function templates do for you. The idea is shown schematically in Figure 1.

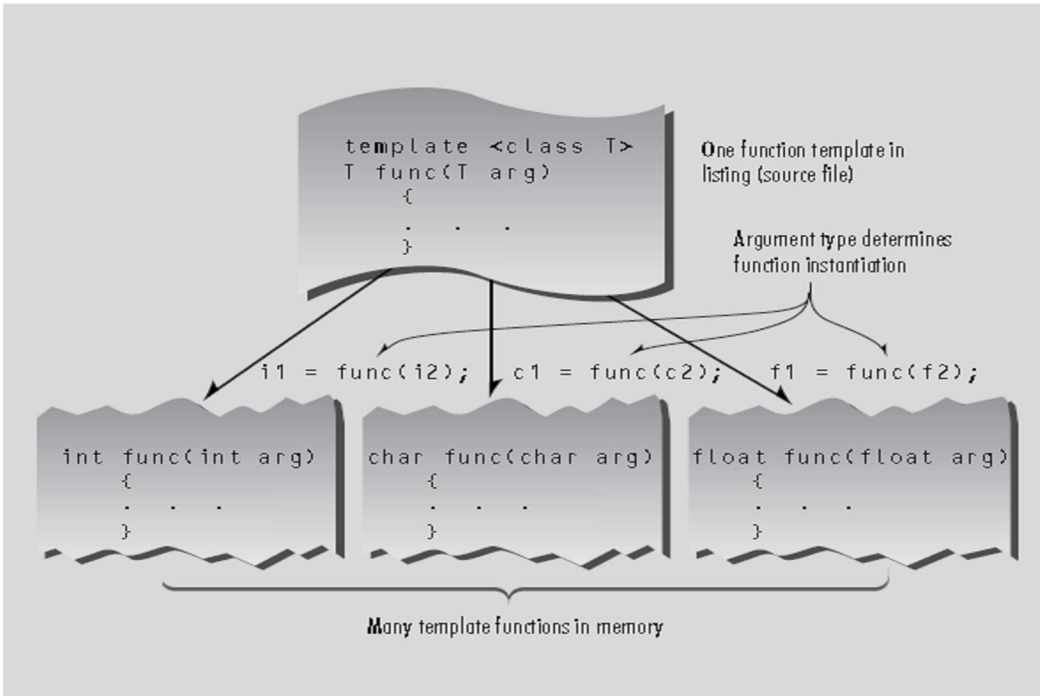

Figure1: ***A function template.***

A Simple Function Template

The first example shows how to write our absolute-value function as a template, so that it will work with any basic numerical type. This program defines a template version of abs() and then, in main(), invokes this function with different data types to prove that it works.

Example 1:Write an OO Program to find the absolute value using template function

```
#include <iostream.h>
//------------------------------------------------------------
template <class T> //function template
T abs(T n)
{
return (n < 0) ? -n : n;
}
//------------------------------------------------------------
int main()
{
int int1 = 5;
int int2 = -6;
long lon1 = 70000L;
long lon2 = -80000L;
double dub1 = 9.95;
double dub2 = -10.15;
//calls instantiate functions
cout << "\nabs(" << int1 << ")=" << abs(int1); //abs(int)
cout << "\nabs(" << int2 << ")=" << abs(int2); //abs(int)
cout << "\nabs(" << lon1 << ")=" << abs(lon1); //abs(long)
cout << "\nabs(" << lon2 << ")=" << abs(lon2); //abs(long)
cout << "\nabs(" << dub1 << ")=" << abs(dub1); //abs(double)
cout << "\nabs(" << dub2 << ")=" << abs(dub2); //abs(double)
cout << endl;
return 0;
}
```

the output of the program:

abs(5)=5
abs(-6)=6
abs(70000)=70000
abs(-80000)=80000
abs(9.95)=9.95
abs(-10.15)=10.15

The **abs()** function now works with all three of the data types (int, long, and double) that we use as arguments. It will work on other basic numerical types as well, and it will even work on user-defined data types, provided that the less-than operator (<) and the unary minus operator (-) are appropriately overloaded.

Here's how we specify the **abs()** function to work with multiple data types:

```
template <class T> //function template
T abs(T n)
{
return (n<0) ? -n : n;
}
```

This entire syntax, with a first line starting with the keyword template and the function definition following, is called a ***function template***. How does this new way of writing abs() give it such amazing flexibility?

Function Template Syntax

The key innovation in function templates is to represent the data type used by the function not as a specific type such as int, but by a name that can stand for *any* type. In the preceding function template, this name is T. The template keyword signals the compiler that we're about to define a function template. The keyword class, within the angle brackets, might just as well be called type. You can define your own data types using classes, so there's really no distinction between types and classes. The variable following the keyword class (T in this example) is called the ***template argument***.

Throughout the definition of the template, whenever a specific data type such as **int** would ordinarily be written, we substitute the template argument, T. In the abs() template this name appears only twice, both in the first line (the function declarator), as the argument type and return type. In more complex functions it may appear numerous times throughout the function body as well.

What the Compiler Does

What does the compiler do when it sees the template keyword and the function definition that follows it? The function template itself doesn't cause the compiler to generate any code. It can't generate code because it doesn't know yet what data type the function will be working with. It simply remembers the template for possible future use.

Code generation doesn't take place until the function is actually called (invoked) by a statement within the program. In example 1 this happens in expressions like abs(int1) in the statement

cout << "\nabs(" << int << ")=" << abs(int1);

When the compiler sees such a function call, it knows that the type to use is int, because that's the type of the argument int1. So it generates a specific version of the abs() function for type int, substituting int wherever it sees the name T in the function template. This is called ***instantiating*** the function template, and each instantiated version of the function is called a ***template function***. (That is, a *template function* is a specific instance of a *function template*.)The compiler also generates a call to the newly instantiated function, and inserts it into the code where abs(int1) is. Similarly, the

expression abs(lon1) causes the compiler to generate a version of abs() that operates on type long and a call to this function, while the abs(dub1) call generates a function that works on type double.

Function Templates with Multiple Arguments

The example bellow takes three arguments: two that are template arguments and one of a basic type. The purpose of this function is to search an array for a specific value. The function returns the array index for that value if it finds it, or –1 if it can't find it. The arguments are a pointer to the array, the value to search for, and the size of the array.

Example 2:Write an OO Program to find number in the array value using template function

```
#include <iostream.h>
//-----------------------------------------------------------------
//function returns index number of item, or -1 if not found
template <class atype>
int find(atype* array, atype value, int size)
{
for(int j=0; j<size; j++)
if(array[j]==value)
return j;
return -1;
}
//-----------------------------------------------------------------
char chrArr[] = {1, 3, 5, 9, 11, 13}; //array
char ch = 5; //value to find
int intArr[] = {1, 3, 5, 9, 11, 13};
int in = 6;
long lonArr[] = {1L, 3L, 5L, 9L, 11L, 13L};
long lo = 11L;
double dubArr[] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;
int main()
{
cout << "\n 5 in chrArray: index=" << find(chrArr, ch, 6);
cout << "\n 6 in intArray: index=" << find(intArr, in, 6);
cout << "\n11 in lonArray: index=" << find(lonArr, lo, 6);
cout << "\n 4 in dubArray: index=" << find(dubArr, db, 6);
cout << endl;
return 0;
}
```

The output of program

5 in chrArray: index=2
6 in intArray: index=-1
11 in lonArray: index=4
4 in dubArray: index=-1

Class Templates

The template concept can be extended to classes. Class templates are generally used for data storage (container) classes. The Stack class," for example, could store data only of type int. Here's a condensed version of that class.

```
class Stack
{
private:
int st[MAX]; //array of ints
int top; //index number of top of stack
public:
Stack(); //constructor
void push(int var); //takes int as argument
int pop(); //returns int value
};
```

If we wanted to store data of type long in a stack, we would need to define a completely new class:

```
class LongStack
{
private:
long st[MAX]; //array of longs
int top; //index number of top of stack
public:
LongStack(); //constructor
void push(long var); //takes long as argument
long pop(); //returns long value
};
```

Similarly, we would need to create a new stack class for every data type we wanted to store.

Example 3:Write an OO Program to implement stack class using template class

```
#include <iostream.h>
const int MAX = 100; //size of array
//////////////////////////////////////////////////////////////
template <class Type>
class Stack
{
private:
Type st[MAX]; //stack: array of any type
int top; //number of top of stack
public:
Stack() //constructor
{ top = -1; }
void push(Type var) //put number on stack
{ st[++top] = var; }
Type pop() //take number off stack
{ return st[top--]; }
};
//////////////////////////////////////////////////////////////
int main()
{
Stack<float> s1; //s1 is object of class Stack<float>
s1.push(1111.1F); //push 3 floats, pop 3 floats
s1.push(2222.2F);
s1.push(3333.3F);
cout << "1: " << s1.pop() << endl;
cout << "2: " << s1.pop() << endl;
cout << "3: " << s1.pop() << endl;
Stack<long> s2; //s2 is object of class Stack<long>
s2.push(123123123L); //push 3 longs, pop 3 longs
s2.push(234234234L);
s2.push(345345345L);
cout << "1: " << s2.pop() << endl;
cout << "2: " << s2.pop() << endl;
cout << "3: " << s2.pop() << endl;
return 0;
}
```

Here's the output:

1: 3333.3 //float stack
2: 2222.2
3: 1111.1
1: 345345345 //long stack
2: 234234234
3: 123123123

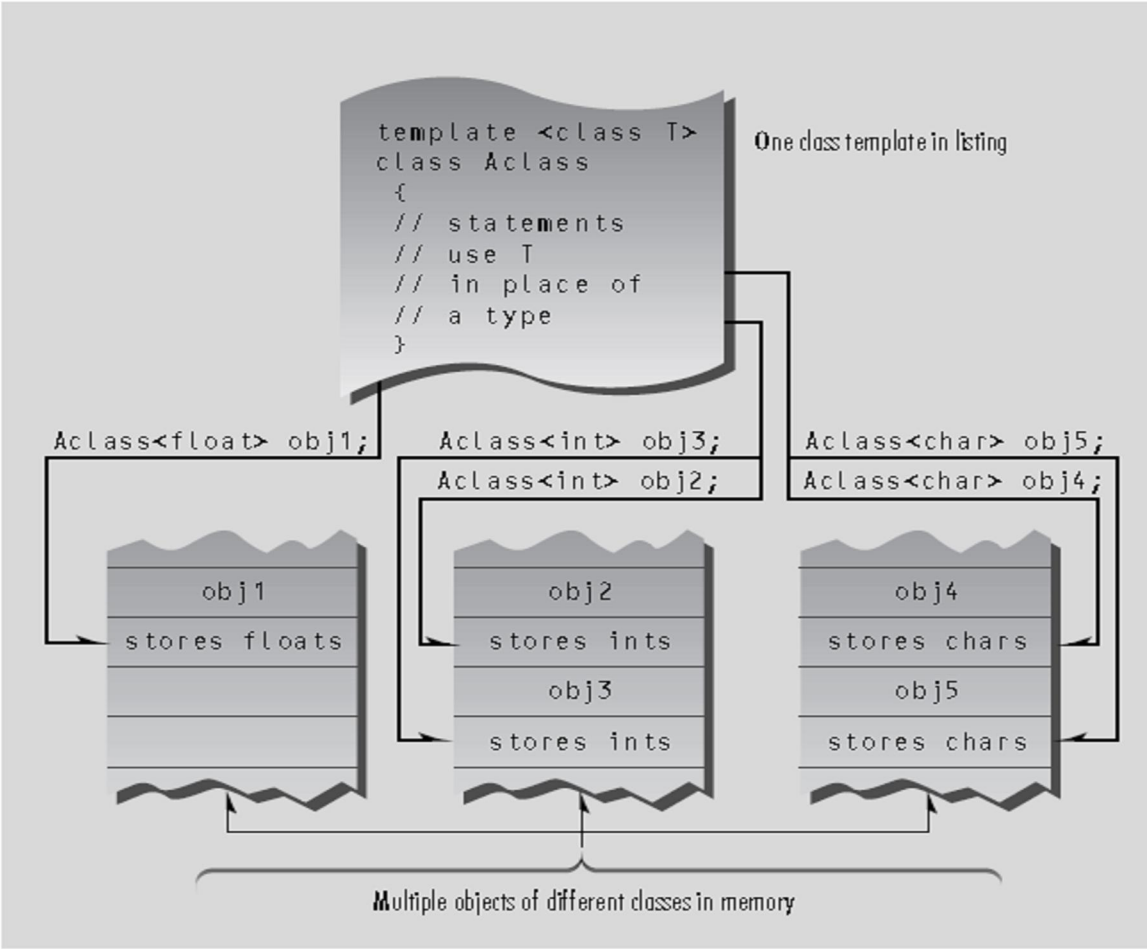

Figure2: ***A class template.***

Class Name Depends on Context

In the example 3, the member functions of the class template were all defined within the class. If the member functions are defined externally (outside of the class specification), we need a new syntax. The next program shows how this works.

Example 4: Write an OO Program to implement stack class using template class with scope operator.

```
#include <iostream.h>
const int MAX = 100;
//////////////////////////////////////////////////////////////
template <class Type>
class Stack
{
private:
Type st[MAX]; //stack: array of any type
int top; //number of top of stack
public:
Stack(); //constructor
void push(Type var); //put number on stack
Type pop(); //take number off stack
};
//////////////////////////////////////////////////////////////
template<class Type>
Stack<Type>::Stack() //constructor
{
top = -1;
}
//-------------------------------------------------------------
template<class Type>
void Stack<Type>::push(Type var) //put number on stack
{
st[++top] = var;
}
//-------------------------------------------------------------
template<class Type>
Type Stack<Type>::pop() //take number off stack
{
return st[top--];
}
//-------------------------------------------------------------
int main()
{
Stack<float> s1; //s1 is object of class Stack<float>
s1.push(1111.1F); //push 3 floats, pop 3 floats
s1.push(2222.2F);
s1.push(3333.3F);
cout << "1: " << s1.pop() << endl;
cout << "2: " << s1.pop() << endl;
cout << "3: " << s1.pop() << endl;

Stack<long> s2; //s2 is object of class Stack<long>
s2.push(123123123L); //push 3 longs, pop 3 longs
s2.push(234234234L);
s2.push(345345345L);
cout << "1: " << s2.pop() << endl;
cout << "2: " << s2.pop() << endl;
cout << "3: " << s2.pop() << endl;
return 0;
}
```

The expression template***<class Type>*** must precede not only the class definition, but each externally defined member function as well. Here's how the push() function looks:

```
template<class Type>
void Stack<Type>::push(Type var)
{
st[++top] = var;
}
```

The name Stack<Type> is used to identify the class of which push() is a member. In a normal non-template member function the name Stack alone would suffice:

```
void Stack::push(int var) //Stack() as a non-template function
{
st[++top] = var;
}
```

but for a function template we need the template argument as well: Stack<Type>.

Thus we see that the name of the template class is expressed differently in different contexts.

Within the class specification, it's simply the name itself: Stack. For externally defined member functions, it's the class name plus the template argument name: Stack<Type>. When you define actual objects for storing a specific data type, it's the class name plus this specific type:

Stack<float>, for example.

```
class Stack //Stack class specifier
{ };
void Stack<Type>::push(Type var) //push() definition
{ }
Stack<float> s1; //object of type Stack<float>
```

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. Why are virtual functions needed? Suppose you have a number of objects of different classes but you want to put them all in an array and perform a particular operation on them using the same function call. For example, suppose a graphics program includes several different shapes: a triangle, a ball, a square. Each of these classes has a member function draw () that causes the object to be drawn on the screen. Now suppose you plan to make a picture by grouping a number of these elements together and you want to draw the picture in a convenient way. One approach is to create an array that holds pointers to all the different objects in the picture. The array might be defined like this:

shape* ptrarr[100]; // array of 100 pointers to shapes .If you insert pointers to all the shapes into this array, you can then draw an entire picture using a simple loop:

For (int j=0; j<N; j++) ptrarr[j]->draw(); This is an amazing capability: Completely different functions are executed by the same function call. If the pointer in ptrarr points to a ball, the function that draws a ball is called; if it points to a triangle, the triangle-drawing function is called. This is called ***polymorphism***, which means ***different forms***. The functions have the same appearance, the draw() expression, but different actual functions are called, depending on the contents of ptrarr[j]. Polymorphism is one of the key features of object-oriented programming, after classes and inheritance.

For the polymorphic approach to work, several conditions must be met.

1) First, all the different classes of shapes, such as balls and triangles, must be descended from a single base class.
2) Second, the draw() function must be declared to be virtual in the base class.

Normal Member Functions Accessed with Pointers

Example 1 shows what happens when a base class and derived classes all have functions with the same name, and you access these functions using pointers but without using virtual functions.

Example 1:

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base //base class
{
public:
void show() //normal function
{ cout << "Base\n"; }
};
////////////////////////////////////////////////////////////////
class Derv1 : public Base //derived class 1
{
public:
void show()
{ cout << "Derv1\n"; }
};
////////////////////////////////////////////////////////////////
class Derv2 : public Base //derived class 2
{
public:
void show()
{ cout << "Derv2\n"; }
};
////////////////////////////////////////////////////////////////
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;
}
```

The Derv1 and Derv2 classes are derived from class Base. Each of these three classes has a member function show (). In main () we create objects of class Derv1 and Derv2, and a pointer to class Base. Then we put the address of a derived class object in the base class pointer in the line

ptr = &dv1; // derived class address in base class pointer . The rule is that pointers to objects of a derived class are type compatible with pointers to objects of the base class. Now the question is, when you execute the line **ptr->show();** what function is called? Is it **Base::show() or Derv1::show()**? Again, in the last two lines of not virtual we put the address of an object of class Derv2 in the pointer, and again execute **ptr->show();** Which of the show() functions is called here? **The output from the program :**

Base

Base

As you can see, the function in the base class is always executed. The compiler ignores the *contents* of the pointer **ptr** and chooses the member function that matches the *type* of the pointer, as shown in Figure 1

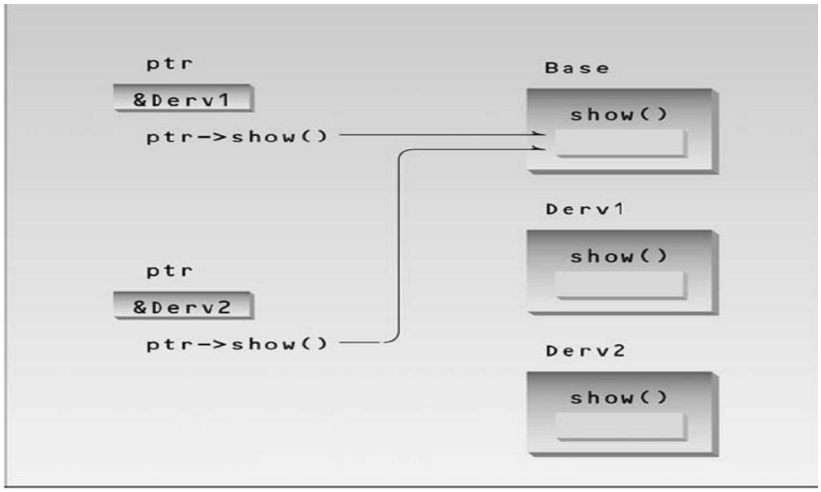

Figure 1 ***Nonvirtual pointer access.***

Virtual Member Functions Accessed with Pointers

Let's make a single change in our program: We'll place the keyword *virtual* in front of the declarator for the show() function in the base class. Here's the listing for the resulting program.

Example 2:

```
#include <iostream.h>
////////////////////////////////////////////////////////////////
class Base //base class
{
public:
virtual void show() //virtual function
{ cout << "Base\n"; }
};
////////////////////////////////////////////////////////////////
class Derv1 : public Base //derived class 1
{
public:
void show()
{ cout << "Derv1\n"; }
};
////////////////////////////////////////////////////////////////
class Derv2 : public Base //derived class 2
{
public:
void show()
{ cout << "Derv2\n"; }
};
////////////////////////////////////////////////////////////////
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;
}
```

The output of this program is

Derv1

Derv2

The member functions of the derived classes, not the base class, are executed. We change the contents of ptr from the address of Derv1 to that of Derv2, and the particular instance of show() that is executed also changes. So the same function call **ptr->show();** executes different functions, depending on the contents of ptr. The rule is that the compiler selects the function based on the *contents* of the pointer ptr, not on the *type* of the pointer, as in not virtual. This is shown in Figure 2

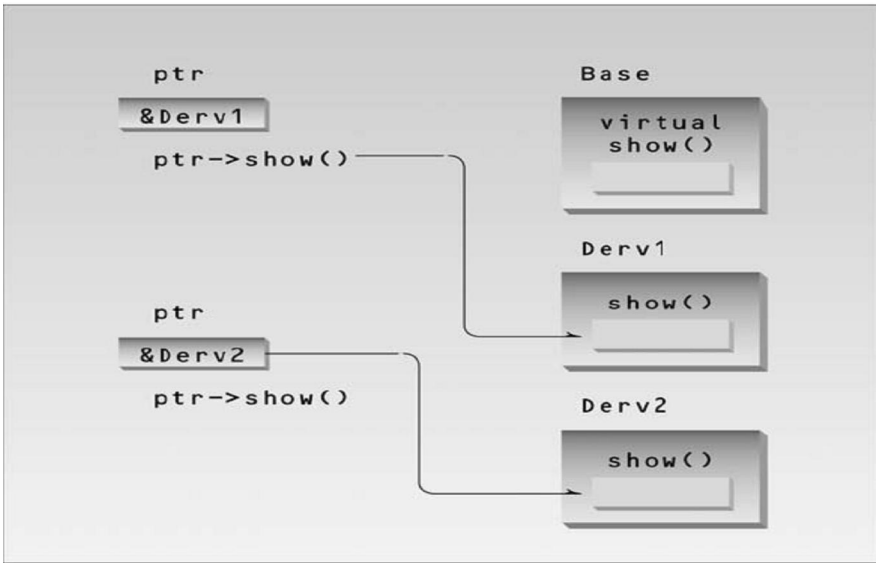

Figure 2 ***Virtual pointer access.***

Late Binding

The astute reader may wonder how the compiler knows what function to compile. In not virtual the compiler has no problem with the expression **ptr->show();** It always compiles a call to the show() function in the base class. But in virtual the compiler doesn't know what class the contents of ptr may contain. It could be the address of an object of the Derv1 class or of the Derv2 class. Which version of draw() does the compiler call? In fact the

compiler doesn't know what to do, so it arranges for the decision to be deferred until the program is running.

At runtime, when it is known what class is pointed to by ptr, the appropriate version of draw will be called. This is called ***late binding*** or ***dynamic binding***.

(Choosing functions in the normal way, during compilation, is called ***early binding*** or ***static binding***.) Late binding requires some overhead but provides increased power and flexibility.

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. 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. This is shown in the example3.

Example 3:

```
#include <iostream.h>
//////////////////////////////////////////////////////////////
class Base //base class
{
public:
virtual void show() = 0; //pure virtual function
};
//////////////////////////////////////////////////////////////
class Derv1 : public Base //derived class 1
{
public:
void show()
{ cout << "Derv1\n"; }
};
//////////////////////////////////////////////////////////////
class Derv2 : public Base //derived class 2
{
public:
void show()
{ cout << "Derv2\n"; }
};
//////////////////////////////////////////////////////////////
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;
}
```

Here the virtual function show() is declared as **virtual void show() = 0; // pure virtual function** .The equal sign here has nothing to do with assignment; the value 0 is not assigned to anything. The =0 syntax is simply how we tell the compiler that a virtual function will be pure. Now if in main() you attempt to create objects of class Base, the compiler will complain that you're trying to instantiate an object of an abstract class. It will also tell you the name of the pure virtual function that makes it an abstract class. Notice that, although this is only a declaration, you never need to write a definition of the base class show().

Virtual Functions and the person Class

Now that we understand some of the mechanics of virtual functions, let's look at a situation where it makes sense to use them. It uses the person class, with two derived classes, student and professor. These derived classes each contain a function called **isOutstanding()**. The person class is an abstract class because it contains the pure virtual functions getData() and isOutstanding().

Example 4:

```
#include <iostream.h>
////////////////////////////////////////////////////////////////
class person //person class
{
protected:
char name[40];
public:
void getName()
{ cout << " Enter name: "; cin >> name; }
void putName()
{ cout << "Name is: " << name << endl; }
virtual void getData() = 0; //pure virtual func
virtual bool isOutstanding() = 0; //pure virtual func
};
////////////////////////////////////////////////////////////////
class student : public person //student class
{
private:
float gpa; //grade point average
public:
void getData() //get student data from user
{
person::getName();
cout << " Enter student's GPA: "; cin >> gpa;
}
bool isOutstanding()
{ return (gpa > 3.5) ? true : false; }
};
////////////////////////////////////////////////////////////////
class professor : public person //professor class
{
private:
int numPubs; //number of papers published
public:
void getData() //get professor data from user
{
person::getName();
cout << " Enter number of professor's publications: ";
cin >> numPubs;
}
```

```
bool isOutstanding()
{ return (numPubs > 100) ? true : false; }
};
int main()
{
person* persPtr[100]; //array of pointers to persons
int n = 0; //number of persons on list
char choice;
do {
cout << "Enter student or professor (s/p): ";
cin >> choice;
if(choice=='s') //put new student
persPtr[n] = new student; // in array
else //put new professor
persPtr[n] = new professor; // in array
persPtr[n++]->getData(); //get data for person
cout << " Enter another (y/n)? "; //do another person?
cin >> choice;
} while( choice=='y' ); //cycle until not 'y'
for(int j=0; j<n; j++) //print names of all
{ //persons, and
persPtr[j]->putName(); //say if outstanding
if( persPtr[j]->isOutstanding() )
cout << " This person is outstanding\n";
}
return 0;
} //end main()
```

Here's some sample interaction:

Enter student or professor (s/p): s

Enter name: Timmy

Enter student's GPA: 1.2

Enter another (y/n)? y

Enter student or professor (s/p): s

Enter name: Brenda

Enter student's GPA: 3.9

Enter another (y/n)? y

Enter student or professor (s/p): s

Enter name: Sandy

Enter student's GPA: 2.4

Enter another (y/n)? y

Enter student or professor (s/p): p

Enter name: Shipley

Enter number of professor's publications: 714

Enter another (y/n)? y

Enter student or professor (s/p): p

Enter name: Wainright

Enter number of professor's publications: 13

Enter another (y/n)? n

Name is: Timmy

Name is: Brenda

This person is outstanding

Name is: Sandy

Name is: Shipley

This person is outstanding

Name is: Wainright

Virtual Destructors

Base class destructors should always be virtual. Suppose you use delete with a base class pointer to a derived class object to destroy the derived-class object. If the base-class destructor is not virtual then delete, like a normal member function, calls the destructor for the base class, not the destructor for the derived class. This will cause only the base part of the object to be destroyed. The program shows how this looks.

```
#include <iostream.h>
class Base
{
```

```
public:
~Base() //non-virtual destructor
// virtual ~Base() //virtual destructor
{ cout << "Base destroyed\n"; }
};
class Derv : public Base
{
public:
~Derv()
{ cout << "Derv destroyed\n"; }
};
//////////////////////////////////////////////////////////////
int main()
{
Base* pBase = new Derv;
delete pBase;
return 0;
}
```

The output for this program as written is **Base destroyed**

This shows that the destructor for the Derv part of the object isn't called. In the listing the base class destructor is not virtual, but you can make it so by commenting out the first definition for the destructor and substituting the second.

Now the output is

Derv destroyed

Base destroyed

Now both parts of the derived class object are destroyed properly. Of course, if none of the destructors has anything important to do (like deleting memory obtained with new) then virtual destructors aren't important. But in general, to ensure that derived-class objects are destroyed properly, you should make destructors in all base classes virtual.

Most class libraries have a base class that includes a virtual destructor, which ensures that all derived classes have virtual destructors.

Virtual Base Classes

Consider the situation shown in Figure 3, with a base class, Parent; two derived classes, Child1 and Child2; and a fourth class, Grandchild, derived from both Child1 and Child2.

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.

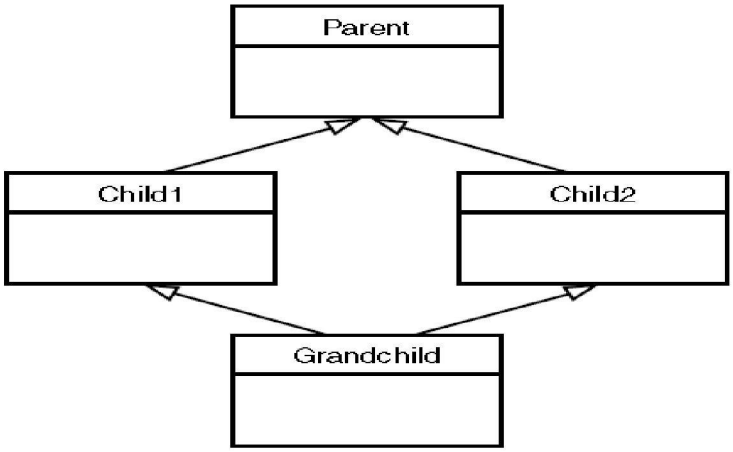

Figure 3 ***Virtual base classes.***

// ambiguous reference to base class

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
};
```

A compiler error occurs when the getdata() member function in Grandchild attempts to access basedata in Parent. Why? When the Child1 and Child2 classes are derived from Parent, each inherits a copy of Parent; this copy is called a *subobject*. Each of the two subobjects contains its own copy of Parent's data, including basedata. Now, when Grandchild refers to basedata, which of the two copies will it access? The situation is ambiguous, and that's what the compiler reports.

To eliminate the ambiguity, we make Child1 and Child2 into virtual base classes, as shown by the example bellow.

```
 // virtual base classes
class Parent
{
```

```
protected:
int basedata;
};
class Child1 : virtual public Parent // shares copy of Parent
{ };
class Child2 : virtual public Parent // shares copy of Parent
{ };
class Grandchild : public Child1, public Child2
{
public:
int getdata()
{ return basedata; } // OK: only one copy of Parent
};
```

The use of the keyword virtual in these two classes causes them to share a single common subobject of their base class Parent. Since there is only one copy of base data, there is no ambiguity when it is referred to in Grandchild. The need for virtual base classes may indicate a conceptual problem with your use of multiple inheritances, so they should be used with caution.

Polymorphism

Polymorphism is one of the crucial features of object oriented programming. It simply means "*one name, multiple forms*". However, polymorphism allows an entity (variable, function or object) to take a variety of representations (take a multiple forms).

Polymorphism refers to situation in which objects belonging to different classes can respond to the same message, usually in different ways. For example suppose we have classes box, triangle and circle, whose objects represent the corresponding geometrical figures. The objects of these classes might all understand a message draw (), which causes an object to draw the corresponding figure on the screen.

In C++ Polymorphism is implemented via ***virtual functions.***

Therefore we have to distinguish different three types of polymorphism:

- Polymorphism of Variables (or Members).
- Polymorphism of Functions (or Methods).
- Polymorphism of Objects.

A. Polymorphism of Variables:

The first type of polymorphism is similar to the concept of dynamic binding. Here, the type of a variable depends on its content. Thus, its type depends on the content at a specific time:

```
int     a=5;    //use a as integer
…..
char   a='g';  //use a as character
……
```

B. Polymorphism of Functions:

Another type of polymorphism can be defined for functions. For example, suppose you want to define a function *isNull( )* which returns TRUE if its argument is zero and FALSE otherwise. For integer numbers this is easy:

```
Bool  isNull(int  r)
{
If (r==0)
Return(true)
Else
Return(false)
}
```

However, if we want to check this for float numbers, we should use another comparison due to the precision problem:

```
Bool  isNull(float  k)
{
If (k<=0.01)&&(k>-0.99)
Return(true)
Else
Return(false)
}
```

In both cases we want the function to have the name *isNull*. In programming languages without polymorphism for functions we cannot declare these two functions because the name *isNull* would be doubly defined. Without polymorphism for functions, doubly defined names would be ambiguous. However, if the language would take the *parameters* of the function into account it would work. Thus, functions (or methods) are uniquely identified by:

- The *name* of the function (or method) and
- The *types* of its *parameter list*.

Since the parameter list of both *isNull* functions differs, the compiler is able to figure-out the correct function call by using the actual types of the arguments.

```
Int  r;
Float   k;
r=0;
k=0.0;
…..
If ( isNull(r))   //use isNull integer
…..
If ( isNull(k))   //use isNull float
```

This type of polymorphism allows us to reuse the same name for functions (or methods) as long as the parameter list differs. Sometimes this type of polymorphism is called ***overloading***.

C. Polymorphism of Objects:

The last type of polymorphism allows an object to choose correct methods. In this type, polymorphism refers to situation in which objects belong to different classes can be respond to the same message, usually in different ways. For example, suppose we have classes box, triangle, and circle, whose objects represent the corresponding geometrical figures, as shown in figure (4). The objects of these classes might all understand a message draw( ), which causes an object to draw the corresponding figure on the screen.

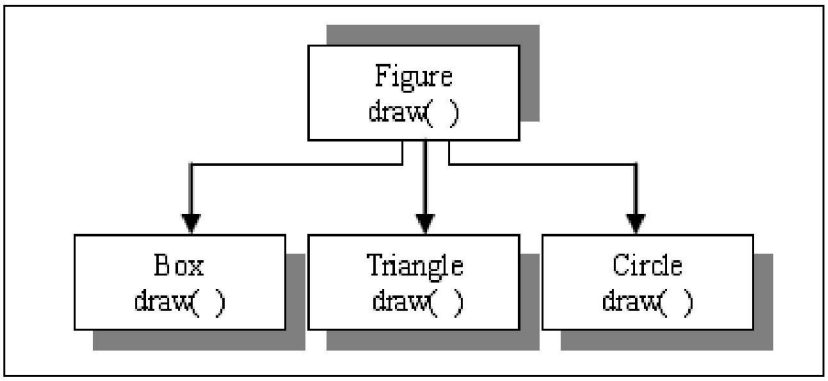

Figure 4: The class hierarchy for the figures example.

An essential feature of polymorphism is that we are able to send messages without knowing the class of the recipient object. For example, we might have a list of objects representing the figures that are to appear on the screen. To display the figures, we can send a draw( ) message to every object on the list, without having to worry about which objects represent boxes, which represent circles, and so on.

A list containing objects from different classes is called a heterogeneous list. Polymorphism greatly simplifies manipulating the objects in a heterogeneous list.