CHAPTER 8

Virtual Functions and Polymorphism

Virtual Functions and Polymorphism help in designing and implementing systems that are more extensible. Polymorphism helps in treating objects of all classes of a hierarchy generically as base class objects. Virtual Functions, the syntactic construct to implement polymorphism provides a means or way of dealing with objects of different types in a program. An alternative solution to virtual functions is switch statement to take the appropriate action based on the object type. Each object type correlates with the case labels of the switch statement. The bottlenecks associated with the switch logic, explicit testing of object types is required. An accidental omission of one of the cases would lead to one object type being not recognized.

Another disadvantage is that if a new class or object is added to the hierarchy, then correspondingly an equivalent case must be inserted in the switch logic to handle the new class's objects. Thus, addition or deletion of classes to handle new types requires the switch statements to be modified, which would be time-consuming or error-prone. Virtual functions, which are the syntactic construct to implement polymorphism, eliminate the need for switch logic. Virtual functions in comparison with switch logic contain less branching and thereby facilitating testing, debugging, program maintenance, and bug avoidance.

8.1 Polymorphism

Polymorphism is thus the ability of objects of different classes related by inheritance to respond differently to the same message or member function call. The same message (in terms of signatures) sent to many different types of objects thus takes many forms. Polymorphism, the dictionary meaning is an entity that can exist in multiple forms. In polymorphism, when a request is made through a base class pointer, the correct overridden function in the appropriate derived class is chosen.

To differentiate a polymorphic from non-polymorphic behaviour, let us assume that a non-virtual member function is defined in a base class and the same function is overridden in the derived class also. If such a member function is called through a base class pointer to the base class object, always the base class version is called. If the member function is referenced through a derived class pointer, the derived class version is used. This in other words, is non-polymorphic behaviour. This is also referred to as static or compile time binding wherein all member function calls are associated with the objects at compile time itself. Such member function calls are resolved statically.

Thus, with non-polymorphic behaviour it is the pointer type and not the actual object type to which it points to that decides the member function call. This is the reasoning behind a member function reference through a base class pointer to derived class object always invoking the base class version, when in fact the object to which it points to (derived class) member function should be invoked. Let us consider the following programming example:

```
Student S, * sptr = &S;
Graduate G * gptr = &G;
sptr→display ( ); // invokes Base class version
gptr→display ( ); // invokes derived class version
sptr = &G;
sptr→display ( ); // invokes base class version
```

In the above example, we consider the same student hierarchy explored in the earlier chapter. Student is the base class and Graduate is a class derived from Student. Both the base and derived classes have their own versions of function `display()`. Assuming `display()` is not a virtual function, referencing display through a base class pointer `[sptr]` always calls the base class version. The calls `sptr→display()` and `gptr→display()` call the intended base class and derived class versions [member function call is decided by the pointer type and not by the actual object type to which it points to].

The call `sptr→display()` after having assigned the address of a derived class object to `sptr` in the previous statement invokes the base class version of display. Logically speaking `sptr` is pointing to an object of derived class type and the member function reference `sptr→display()` should actually invoke the derived class version of `display()`. For such a polymorphic behaviour to be supported it requires dynamic binding or binding function calls at run-time.

Well, virtual functions are the syntactic provisions in C++ to implement polymorphic behaviour or run-time binding of function calls. That is functions that need to support polymorphic behaviour should be explicitly declared by the programmer as virtual. However, to invoke a derived class version through a base class pointer pointing to a derived class object [function being non virtual]. This is just an alternative or a way out in cases where virtual functions are not supported. For example:

```
sptr→Graduate::  display();
```

or

```
gptr→Student:d:  display ();
```

Through the use of virtual functions member function calls can be made to cause different actions depending on the object type receiving the respective function call.

8.2 Virtual Functions

To support polymorphism or to treat al objects related by inheritance as objects of the base class and there by let the program determine dynamically at run-time which version of derived class to be used, we declare functions as virtual. To enable this kind of polymorphic behaviour, the member function that needs to support polymorphism is declared virtual in the base class.

For example, with respect to the student inheritance hierarchy the `display()` functions should be declared as virtual in the base class. The function prototype or header (only) is preceded by the keyword virtual to make the respective function virtual and thereby support polymorphic behaviour. The exact syntax to declare a virtual function is as follows:

```
virtual void display();
```

There is no need for the virtual keyword to be repeated again in the function header of the function definition. The above function `display()` can very well be made a constant function as well, since it requires only read and not write permissions.

A function that has been declared as virtual in the base class remains virtual throughout the inheritance hierarchy, even though it is not declared virtual in a class that overrides it. Thus, a function that has been declared virtual in the base class (explicitly) is implicitly treated as a virtual function in all its derived classes as well. Though functions are implicitly declared as virtual in the successive derived classes that override, to promote program clarity it is a good programming practice that functions be explicitly declared as virtual even in classes that overrides it. And also when a derived class decides not to override or redefine a virtual function, the derived class inherits its immediate base class's virtual function version.

When a virtual function is invoked by referencing an object by name and the . operator, the call is resolved at compile time and the virtual function that is called is the one inherited or overridden by the class of that particular object. They dynamic or polymorphic behaviour is supported only when a base class pointer referring to a derived class object references a virtual function overridden or inherited in one of the derived classes.

8.3 Abstract Classes

All along classes that were defined were instantiated at some stage or the other of an OO program. Such classes that are instantiated or objects or instances of which are created is referred to as concrete classes. But in certain cases, classes may exist solely as a part of an inheritance hierarchy and no objects or instances of such a class intend to be created. This class is purely existing for the purpose of inheritance to distribute its members to classes that shall be derived from it. Such classes are called abstract classes and no instances or objects of an abstract class can be created. In fact, they are also referred to as Abstract Base Class, since they are existing only as a base class [for future derivations]. In fact, abstract classes are generic while concrete (normal) classes are specific in their existence.

Abstract classes are normally at the top of an inheritance hierarchy. As and when we get deeper and deeper into the hierarchy, it is in fact derived classes that are more specific instances of the base class. A class can be declared to be an abstract class by declaring one or more of its member functions to be virtual and not just virtual, but pure virtual! A pure virtual function is one such function that is initialized to 0 in its declaration. An example of pure virtual function declaration is virtual void `display () =0`. For a pure virtual function, there should be no definition in the base class. There should be at least one pure virtual function in a class to be treated as an abstract class.

A class that is derived from an abstract class [has at least one pure virtual function] and the derived class does not provide a definition for that pure virtual function (s) in it, then the function becomes pure virtual even in the derived class and thus consequently the derived

class also becomes an abstract class and cannot be instantiated. It can only be used to derive further classes from it. Note that attempts to create objects or instantiate an abstract class causes syntax error.

Thus, virtual functions and polymorphism help the programmer in implementing generic behaviour and let the specifics to be decided at run-time. New types of objects that will respond to existing message can be added without modifying the base system. Abstract class defines interface for the various members in a class hierarchy. The pure virtual functions in an abstract class are defined in its derived class. All objects in the inheritance hierarchy can make use of the same interface through polymorphism.

Though abstract classes cannot be instantiated, pointers and references to abstract classes can be created and thereby support polymorphic behaviour/manipulations of derived class objects. Polymorphism is most often used for layered software systems. In operating systems development, physical devices operate differently from others. Regardless of the type of the physical device, read or write commands from and to devices have certain uniformity. However, each message needs to be interpreted specifically in the context of the device deriver.

An OO OS might use an abstract base class to provide an interface appropriate to all device drivers. Through inheritance, derived classes are created and all these classes operate uniformly. These capabilities (interfaces) offered by the device drivers are provided as pure virtual functions in the abstract base class. Implementations of virtual functions are provided in the derived classes that correspond to the specific type of device drivers. Having had sufficient theoretical discussion behind virtual functions, let us now explore a programming example to understand the syntax and functioning of virtual functions. We consider the employee set-up in this example, code for which is as shown in Figure 8.1.

```
"employee.h"
class Employee
{
public:
Employee (const char*,const char*);
~ Employee();
virtual double salary()const =0;
virtual void display() const;
private: char *firstname;
char * lastname;
};
"functions.cpp"
#include"employee.h"
Employee::Employee(const char *fn,const char*ln)
{firstname=new char[strlen(fn)+1];
strcpy(firstname,fn);
name=new char[strlen(ln)+1];
strcpy(firstname,fn);
}
Employee::~Employee
```

Figure 8.1: Continued

```
{delete [] firstname;
delete [] lastname;
}void Employee::display()const
{
cout<<firstname<<lastname;
}
"foreman.h"
#include"employee.h"
class Employee::public Foreman
{
public:
Foreman(const char*,const char*,double);
private: double pay;
};
"Foreman.cpp"
#include"foreman.h"
Foreman::Foreman(const char*f,const char *l,double p)
::Employee(f,l)
{
pay=p;
}
double Foreman::salary()const
{
cout<<"Employee's Pay is "<<pay;
}
"Manager.h"
#include"employee.h"
class Employee::public Manager
{
public:
Manager(const char*,const char*,double,double);
private:
double basic;
double perks;
};
"Manager.cpp"
#include"Manager.h"
Manager::Manager(const char*fn,const char *ln,double bp,double pk)
::Employee(fn,ln)
{
basic=bp;
perks=pk;
}double Manager::salary()const
{
cout<<"Employee's Pay is "<<basic+perks;
}
"main.cpp"
#include<iostream.h>
#include"employee.h"
```

Figure 8.1: Continued

```
#include"foreman.h"
#include"Manager.h"
int main()
{
Foreman f("Shankar","Kumar",4587.00);
Manager m("Ashok","Singhal",6500.85,3000);
Employee *ep;
f.display();
m.display();
ep=&f;
ep→display();
ep=&m;
ep→display();
return 0;
}
```

Figure 8.1: Virtual functions, C++ code.

8.4 Code Interpretation

In the above example in Figure 8.1, the base class employee (abstract class) contains data members' first and last name. Member function display that is expected to display the first name and last name details of an employee object is made virtual while salary is a pure virtual function. That is to say that future derivations from Employee class may or may not override the behaviour of display function (depending on their requirement) but every future derived class must necessarily redefine or override the salary function. Thus the salary computation logic for the varied employees is assumed to be unique and different from one another. Note the driver routine creates two instances of class Foreman and Manager and displays the salary details. A pointer to the base class (ep) is assigned the address of the either derived class objects (f and m), and because the member functions have been declared to be virtual, the appropriate functions (in the order of Foreman's followed by that of Manager's) are called.

Polymorphism and virtual functions are helpful to the programmer when all possible classes are not known in advance. New classes are accommodated by dynamic binding (late binding) or binding at run-time. An object's type need not be known at compile time for a virtual function call to be compiled. At run-time the virtual function call is matched with the appropriate member function of the called object. Dynamic binding also helps in maintaining code secrecy when software is distributed to end users. Such distributions contain only header and objects files and not the implementation. Source code (of member functions) are not revealed. End users use inheritance to derive classes from those that are distributed.

Another important issue to be resolved with polymorphism is related to destructors. If an object is destroyed by explicitly applying the delete operator to a base class pointer to the derived class object, the base class destructor is called, as a result of matching by pointer type. Always the base class destructor is called irrespective of the type of the object to which the base class pointer actually points. This happens even though the derived class's destructor differs in its name or function identifier. Well, this issue is overcome by declaring the destructor of the base class to be virtual.

With such a declaration in place, automatically the derived class's destructor also becomes virtual, even though it differs in its name. With such virtual base class destructors, on deleting a base class pointer to a derived class object, the destructor for the appropriate class derived is called. Well, after destroying the derived class explicit members, automatically the base class constructor is called to destroy the base class portion of the derived class object.

The reason why this issue has been resolved is that otherwise the base class destructor would destruct the base class portion of the derived class object, leaving the derived class object's explicit members undestroyed and also the principle that "destructions are in the reverse order of construction" gets violated since the base class destructor gets called before the derived class destructor, (even if we were to assume that the derived class destructor gets called somehow). Hence, the need for virtual destructors to ensure proper and as per principle destruction.

Well, the next issue that should strike or disturb any oo Programmer is whether constructors should be made virtual and of course the next issue can constructors be made virtual. The answer is that constructors cannot be made virtual; doing so leads to syntax errors. Again there is reasoning behind this syntax. Constructors are required at the point of object creation for initialization of data members to valid state. Well, dynamic binding or polymorphism is in terms of behaviours associated with objects that have been already created. Hence constructors cannot be made virtual and there is no need for them to be virtual as well.

8.5 Internals Involved with Virtual Functions and Polymorphism

When the C++ compiler encounters a class that has one or more virtual functions, the compiler creates or builds a `vtable` or virtual function table for that class. This `vtable` is used to select appropriate functions every time a virtual function of that class is to be called. Let us consider the employee hierarchy again here to illustrate the contents of `vtable` and how virtual function calls actually get resolved, code snippet of which is shown in Figure 8.2. Let us consider the following class structure. Again here we will not be very specific with the member function

```
Class Employee
{
char * fname;
char * lname;
public :
virtual void f1() const
{
cout <<"Function F1 \n";
}
virtual void f2() const
{
cout <<"Function F1 \n";
}
virtual void f3() const =0;
virtual void f4() const =0;
};
```

Figure 8.2: Sample class specification to illustrate virtual functions.

definitions/their names. We will be more interested to understand the intricacies behind the virtual function calls resolution.

The `vtable` is actually a collection of function pointers, the number of function pointers being equal to the total number of virtual and pure virtual functions in the base class. When the base class is compiled, the `vtable` consisting of four function pointers is created (we exclude the virtual destructor in this discussion). The first function pointer points to the implementation of the function `f1()` that displays the string function `F1`.

The second function pointer points to the definition of the function `f2()` that displays the string function `F2`. Note that functions `f1` and `f2` are virtual while `f3` and `f4` are pure virtual. That is `f1` and `f2` may or may not be overridden in the derive class. But `f3` and `f4` should be necessarily overridden in derived classes, else the derived class would become abstract class. We have discussed this already.

Getting back to `vtable`, the third and fourth function pointers that are for pure virtual functions are each set to `0` since these pure virtual functions do have implementation or function definition in the base class. Any class that has one or more zero pointers in the `vtable` is an abstract class. Classes without any `0` pointers in their `vtable` are concrete classes. Since virtual functions are propagated throughout the inheritance hierarchy, the compiler builds the `vtable` for all the derived classes as well.

Derived classes in which functions `f1` and `f2` are not redefined or overridden, the `vtable` function pointers are set to the copies of the `f1` and `f2` implementation in the base class or base class `vtable`. Derived classes in which `f1` and `f2` are overridden, the function pointers point to the respective function definitions. The remaining function pointers [which are for pure virtual functions] of each derived class `vtable` points to their respective function definitions. Polymorphism is achieved via a data structure that consists of three levels of pointers.

The function pointers discussed above form the first level. These pointers point to the actual functions that are to be executed when a virtual function is invoked. Whenever an object of a class with virtual functions is instantiated, the compiler automatically attaches a pointer to the `vtable` for the respective class in front of the instantiated object. These form the second level pointers. The third pointer is the handle on the object that receives the virtual function.

Let us assume the following function call of `bptr→f3()` after having assigned to `bptr` (base class pointer) the address of a derived class object. When the compiler encounters the above statement, it determines that the call is being made off a base class pointer and `f3` is a virtual function. The `vtable` pointer associated with the call is dereferenced to reach the respective derived class's `vtable`. Then the required number of bytes [in this case 8] is skipped to reach the function `f3()'s` function pointer in the `vtable`. This is with the assumption that a function pointer consumes 4 bytes of memory. Then this function pointer is dereferenced to form the actual function call to be invoked.

All these data structures are manipulated internally by the compiler and completely hidden from the programmer. The `vtable` and `vtable` pointers consume some memory. The additional execution time required for virtual function calls is because of the pointer dereferencing operations and memory accesses that occur on every virtual function call. A pictorial description of the virtual function resolutions is shown in Figure 8.3.

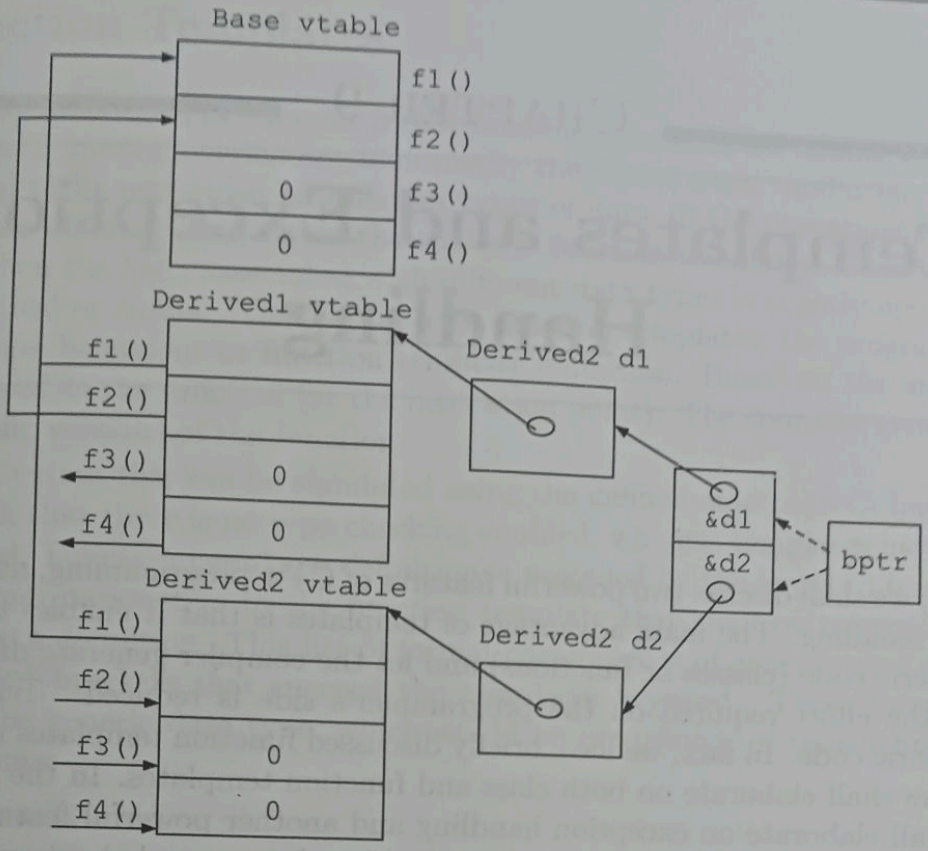

Figure 8.3: Virtual functions—Internals' illustration.

Review Questions

1. Define the term polymorphism and explain the support of polymorphism in C++.
2. Define the term abstract base class and discuss the syntax to create abstract base class in C++.
3. Can abstract base classes be instantiated? If not, what is the purpose of having such classes?
4. Appreciate the notion of polymorphism and virtual functions from a day to day computers related device usage point of view. Example with respect to the various media types used such as floppy, pen drive, cdrom, support polymorphic operations of open, read, write and close. Emphasis must be on the broad outlay of classes and member functions without getting into the details of physical read and write effects.
5. Explain in detail the data structure used to support virtual functions in C++.
6. Differentiate static from dynamic binding of objects.
7. For the university package created in Chapter 7 (exercise) support polymorphism with respect to operations that are common to different entities of the university such as printing the details of student/employee, insurance details, etc.
8. Differentiate a virtual from a pure virtual function.
9. Justify the need for virtual destructors in C++.
10. Give a few examples of programming situations in real life(day to day usage of computers) where polymorphic behaviour is required.