9

Pointers, Virtual Functions and Polymorphism

Key Concepts

- Polymorphism
- Pointers
- Pointers to objects
- this pointer
- Pointers to derived classes
- Virtual functions
- Pure virtual function

9.1 Introduction

Polymorphism is one of the crucial features of OOP. It simply means 'one name, multiple forms'. We have already seen how the concept of *polymorphism* is implemented using the overloaded functions and operators. The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called *early binding* or *static binding* or *static linking*. Also known as *compile time polymorphism*, early binding simply means that an object is bound to its function call at compile time.

Now let us consider a situation where the function name and prototype is the same in both the base and derived classes. For example, consider the following class definitions:

```
class A
{
    int x;
   public:
```

```
    void show() {....}          // show() in base class
};
class B: public A
{
        int y;
    public:
        void show() {....}      // show() in derived class
};
```

How do we use the member function **show()** to print the values of objects of both the classes **A** and **B**?. Since the prototype of **show()** is the same in both the places, the function is not overloaded and therefore static binding does not apply. We have seen earlier that, in such situations, we may use the class resolution operator to specify the class while invoking the functions with the derived class objects.

It would be nice if the appropriate member function could be selected while the program is running. This is known as *run time polymorphism*. How could it happen? C++ supports a mechanism known as *virtual function* to achieve run time polymorphism. Please refer Fig. 9.1.

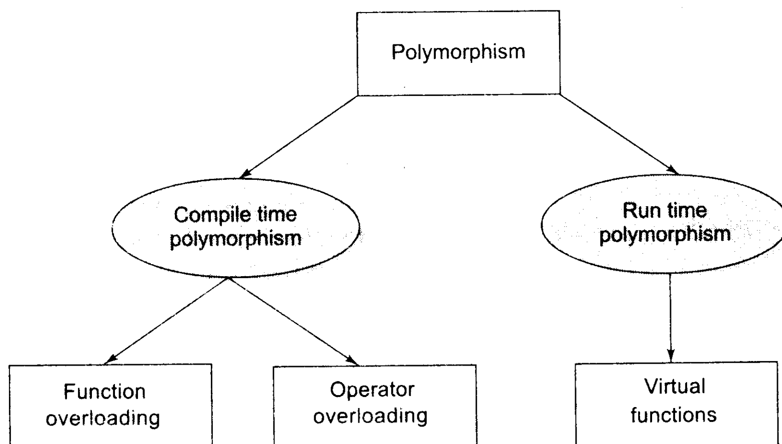

Fig. 9.1 ⇔ *Achieving polymorphism*

At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as *late binding*. It is also known as *dynamic binding* because the selection of the appropriate function is done dynamically at run time.

Dynamic binding is one of the powerful features of C++. This requires the use of pointers to objects. We shall discuss in detail how the object pointers and virtual functions are used to implement dynamic binding.

9.2 Pointers

Pointers is one of the key aspects of C++ language similar to that of C. As we know, pointers offer a unique approach to handle data in C and C++. We have seen some of the applications of pointers in Chapters 3 and 5. In this section, we shall discuss the rudiments of pointers and the special usage of them in C++.

We know that a pointer is a derived data type that refers to another data variable by storing the variable's memory address rather than data. A pointer variable defines where to get the value of a specific data variable instead of defining actual data.

Like C, a pointer variable can also refer to (or point to) another pointer in C++. However, it often points to a data variable. Pointers provide an alternative approach to access other data objects.

Declaring and Initializing Pointers

As discussed in Chapter 3, we can declare a pointer variable similar to other variables in C++. Like C, the declaration is based on the data type of the variable it points to. The declaration of a pointer variable takes the following form:

```
data-type *pointer-variable;
```

Here, *pointer-variable* is the name of the pointer, and the *data-type* refers to one of the valid C++ data types, such as int, char, float, and so on. The *data-type* is followed by an asterisk (*) symbol, which distinguishes a pointer variable from other variables to the compiler.

> *note*
>
> We can locate asterisk (*) immediately before the pointer variable, or between the data type and the pointer variable, or immediately after the data type. It does not cause any effect in the execution process.

As we know, a pointer variable can point to any type of data available in C++. However, it is necessary to understand that a pointer is able to point to only one data type at the specific time. Let us declare a pointer variable, which points to an integer variable, as follows:

```
int *ptr;
```

Here, **ptr** is a pointer variable and points to an integer data type. The pointer variable, ptr, should contain the memory location of any integer variable. In the same manner, we can declare pointer variables for other data types also.

Like other programming languages, a variable must be initialized before using it in a C++ program. We can initialize a pointer variable as follows:

```
int *ptr, a; // declaration
ptr=&a; // initialization
```

The pointer variable, **ptr**, contains the address of the variable **a**. Like C, we use the 'address of' operator or reference operator i.e. '&' to retrieve the address of a variable. The second statement assigns the address of the variable **a** to the pointer **ptr.**

We can also declare a pointer variable to point to another pointer, similar to that of C. That is, a pointer variable contains address of another pointer. Program 9.1 explains how to refer to a pointer's address by using a pointer in a C++ program.

EXAMPLE OF USING POINTERS

```
#include <iostream.h>
#include <conio.h>
void main()
{
int a, *ptrl, **ptr2;
clrscr();
ptrl = &a;
ptr2=&ptrl;
cout << "The address of a : " << ptrl << "\n";
cout << "The address of ptrl : " << ptr2;
cout << "\n\n";
cout << "After incrementing the address values:\n\n";
ptrl+=2;
cout << "The address of a : " << ptrl << "\n";
ptr2+=2;
cout << "The address of ptrl : " << ptr2 << "\n";
}
```

PROGRAM 9.1

The memory location is always addressed by the operating system. The output may vary depends on the system. Output of Program 9.1 would look like:

```
The address of a :      0x8fb6fff4
The address of ptrl:    0x8fb6fff2
After incrementing the address values:
The address of a :      0x8fb6fff8
The address of a :      0x8fb6fff6
```

We can also use *void pointers*, known as generic pointers, which refer to variables of any data type. Before using void pointers, we must type cast the variables to the specific data types that they point to.

note

The pointers, which are not initialized in a program, are called Null pointers. Pointers of any data type can be assigned with one value i.e., '0' called null address.

Manipulation of Pointers

As discussed earlier, we can manipulate a pointer with the indirection operator, i.e. '*', which is also known as dereference operator. With this operator, we can indirectly access the data variable content. It takes the following general form:

```
*pointer_variable
```

As we know, dereferencing a pointer allows us to get the content of the memory location that the pointer points to. After assigning address of the variable to a pointer, we may want to change the content of the variable. Using the dereference operator, we can change the contents of the memory location.

Let us consider an example that illustrates how to dereference a pointer variable. The value associated with the memory address is divided by 2 using the dereference operator. The division affects only the memory contents and not the memory address itself. Program 9.2 illustrates the use of dereference operator in C++.

```
#include <iostream.h>

#include <conio.h>

void main()
```

(Contd)

```
{
int a=10, *ptr;
ptr = &a;
clrscr();
cout << "The value of a is : " << a;
cout << "\n\n";
*ptr=(*ptr)/2;
cout << "The value of a is : " << (*ptr);
cout << "\n\n";
}
```

PROGRAM 9.2

Output of Program 9.2:

```
The value of a is : 10
The value of a is : 5
```

caution

Before dereferencing a pointer, it is essential to assign a value to the pointer. If we attempt to dereference an uninitialized pointer, it will cause runtime error by referring to any other location in memory.

Pointer Expressions and Pointer Arithmetic

As discussed in Chapter 3, there are a substantial number of arithmetic operations that can be performed with pointers. C++ allows pointers to perform the following arithmetic operations:

- A pointer can be incremented (++) (or) decremented (– –)
- Any integer can be added to or subtracted from a pointer
- One pointer can be subtracted from another

Example:

```
int a[6];
int *aptr;
aptr=&a[0];
```

Obviously, the pointer variable, **aptr**, refers to the base address of the variable **a**. We can increment the pointer variable, shown as follows:

```
aptr++ (or) ++aptr
```

This statement moves the pointer to the next memory address. Similarly, we can decrement the pointer variable, as follows:

```
aptr-- (or) --aptr
```

This statement moves the pointer to the previous memory address. Also, if two pointer variables point to the same array can be subtracted from each other.

We cannot perform pointer arithmetic on variables which are not stored in contiguous memory locations. Program 9.3 illustrates the arithmetic operations that we can perform with pointers.

ARITHMETIC OPERATIONS ON POINTERS

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int num[]={56,75,22,18,90};
    int *ptr;
    int i;
    clrscr();
    cout << "The array values are:\n";
    for(i=0;i<5;i++)
           cout<< num[i]<<"\n";
    /* Initializing the base address of str to ptr */
    ptr = num;
    /* Printing the value in the array */
    cout << "\nValue of ptr   : "<< *ptr;
    cout << "\n";
    ptr++;
    cout<<"\nValue of ptr++   : "<<*ptr;
    cout << "\n";
    ptr--;
    cout<<"\nValue of ptr--   : "<<*ptr;
    cout << "\n";
    ptr=ptr+2;
```

(Contd)

```
    cout<<"\nValue of ptr+2   : "<<*ptr;
    cout << "\n";
    ptr=ptr-1;
    cout <<"\nValue of ptr-1: "<< *ptr;
    cout << "\n";
    ptr+=3;
    cout<<"\nValue of ptr+=3: "<<*ptr;
    ptr-=2;
    cout << "\n";
    cout<<"\nValue of ptr-=2: "<<*ptr;
    cout << " \n";
    getch();
}
```

PROGRAM 9.3

Output of Program 9.3:

```
The array values are:
56
75
22
18
90
Value of ptr     :  56
Value of ptr++   :  75
Value of ptr--   :  56
Value of ptr+2   :  22
Value of ptr-1   :  75
Value of ptr+=3  :  90
Value of ptr-=2  :  22
```

Using Pointers with Arrays and Strings

Pointer is one of the efficient tools to access elements of an array. Pointers are useful to allocate arrays dynamically, i.e. we can decide the array size at run time. To achieve this, we use the functions, namely **malloc()** and **calloc()**, which we already discussed in Chapter 3. Accessing an array with pointers is simpler than accessing the array index.

In general, there are some differences between pointers and arrays; arrays refer to a block of memory space, whereas pointers do not refer to any section of memory. The memory addresses of arrays cannot be changed, whereas the content of the pointer variables, such as the memory addresses that it refer to, can be changed.

Even though there are subtle differences between pointers and arrays, they have a strong relationship between them.

note

There is no error checking of array bounds in C++. Suppose we declare an array of size 25. The compiler issues no warnings if we attempt to access 26th location. It is the programmer's task to check the array limits.

We can declare the pointers to arrays as follows:

```
int *nptr;
nptr=number[0];
```

Or

```
nptr=number;
```

Here, **nptr** points to the first element of the integer array, number[0]. Also, consider the following example:

```
float *fptr;
fptr=price[0];
```

Or

```
fptr=price;
```

Here, **fptr** points to the first element of the array of float, price[0]. Let us consider an example of using pointers to access an array of numbers and sum up the even numbers of the array. Initially, we accept the count as an input to know the number of inputs from the user. We use pointer variable, ptr to access each element of the array. The inputs are checked to identify the even numbers. Then the even numbers are added, and stored in the variable, sum. If there is no even number in the array, the output will be 0. Program 9.4 illustrates how to access the array contents using pointers.

POINTERS WITH ARRAYS

```
#include <iostream.h>
void main()
{
        int numbers[50], *ptr;
        int n,i;
        cout << "\nEnter the count\n";
        cin >> n;
```

(Contd)

```
        cout << "\nEnter the numbers one by one\n";
        for(i=0;i<n;i++)
        cin >> numbers[i];
        /* Assigning the base address of numbers to ptr and initializing
        the sum to 0*/
        ptr = numbers;
        int sum=0;
        /* Check out for even inputs and sum up them*/
        for(i=0;i<n;i++)
    {
        if (*ptr%2==0)
             sum=sum+*ptr;
        ptr++;
    }

    cout << "\n\nSum of even numbers: " << sum;
}
```

PROGRAM 9.4

Output of Program 9.4:

```
Enter the count
5
Enter the numbers one by one
10
16
23
45
34
Sum of even numbers:    60
```

Arrays of Pointers

Similar to other variables, we can create an array of pointers in C++. The array of pointers represents a collection of addresses. By declaring array of pointers, we can save a substantial amount of memory space.

An array of pointers point to an array of data items. Each element of the pointer array points to an item of the data array. Data items can be accessed either directly or by dereferencing the elements of pointer array. We can reorganize the pointer elements without affecting the data items.

We can declare an array of pointers as follows:

```
int *inarray[10];
```

This statement declares an array of 10 pointers, each of which points to an integer. The address of the first pointer is inarray[0], and the second pointer is inarray[1], and the final pointer points to inarray[9]. Before initializing, they point to some unknown values in the memory space. We can use the pointer variable to refer to some specific values. Program 9.5 explains the implementation of array of pointers.

ARRAYS OF POINTERS

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
void main()
{
	int i=0;
	char *ptr[10] = {
			"books",
			"television",
			"computer",
			"sports"
			};
	char str[25];
	clrscr();
	cout << "\n\n\n\nEnter your favorite leisure pursuit: " ;
	cin >> str;
	for(i=0; i<4; i++)
	{
		if(!strcmp(str, *ptr[i]))
		{
		cout << "\n\nYour favorite pursuit " << " is available here"
		<< endl;
		break;
		}
```

(Contd)

```
    }
    if(i==4)
        cout << "\n\nYour favorite " << " not available here" << endl;
   getch();
}
```

PROGRAM 9.5

Output of Program 9.5:

```
Enter your favorite leisure pursuit: books
Your favorite pursuit is available here
```

Pointers and Strings

We have seen the usage of pointers with one dimensional array elements. However, pointers are also efficient to access two dimensional and multi-dimensional arrays in C++. There is a definite relationship between arrays and pointers. C++ also allows us to handle the special kind of arrays, i.e. strings with pointers.

We know that a string is one dimensional array of characters, which start with the index 0 and ends with the null character '\0' in C++. A pointer variable can access a string by referring to its first character. As we know, there are two ways to assign a value to a string. We can use the character array or variable of type char *. Let us consider the following string declarations:

```
char num[]="one";
const char *numptr= "one";
```

The first declaration creates an array of four characters, which contains the characters, 'o','n','e','\0', whereas the second declaration generates a pointer variable, which points to the first character, i.e. 'o' of the string. There is numerous string handling functions available in C++. All of these functions are available in the header file <cstring>.

Program 9.6 shows how to reverse a string using pointers and arrays.

ACCESSING STRINGS USING POINTERS AND ARRAYS

```
#include <iostream.h>
#include <string.h>
void main()
```

(Contd)

```
{
    char str[] = "Test";
    int len = strlen(str);
    for(int i=0; i<len; i++)
    {
        cout << str[i] << i[str] << *(str+i) << *(i+str);
    }
    cout << endl;
    //String reverse
    int lenM = len / 2;
    len--;
    for(i=0; i<lenM; i++)
    {
        str[i]     = str[i] + str[len-i];
        str[len-i] = str[i] - str[len-i];
        str[i]     = str[i] - str[len-i];
    }
    cout << " The string reversed : "  << str;
    }
```

PROGRAM 9.6

Output of Program 9.6:

```
TTTTeeeessssstttt
The string reversed : tseT
```

Pointers to Functions

Even though pointers to functions (or function pointers) are introduced in C, they are widely used in C++ for dynamic binding, and event-based applications. The concept of pointer to function acts as a base for pointers to members, which we have discussed in Chapter 5.

The pointer to function is known as callback function. We can use these function pointers to refer to a function. Using function pointers, we can allow a C++ program to select a function dynamically at run time. We can also pass a function as an argument to another function. Here, the function is passed as a pointer. The function pointers cannot be dereferenced. C++ also allows us to compare two function pointers.

C++ provides two types of function pointers; function pointers that point to static member functions and function pointers that point to non-static member functions. These two function pointers are incompatible with each other. The function pointers that point to the non-static member function requires hidden argument.

Like other variables, we can declare a function pointer in C++. It takes the following form:

```
data_type(*function_name)();
```

As we know, the data_type is any valid data types used in C++. The function_name is the name of a function, which must be preceded by an asterisk (*). The function_name is any valid name of the function.

Example:

```
int (*num_function(int x));
```

Remember that declaring a pointer only creates a pointer. It does not create actual function. For this, we must define the task, which is to be performed by the function. The function must have the same return type and arguments. Program 9.7 explains how to declare and define function pointers in C++.

POINTERS TO FUNCTIONS

```
#include <iostream.h>
typedef void (*FunPtr)(int, int);
void Add(int i, int j)
{
    cout << i << " + " << j << " = " << i + j;
}
void Subtract(int i, int j)
{
    cout << i << " - " << j << " = " << i - j;
}
void main()
{
    FunPtr ptr;
    ptr = &Add;
    ptr(1,2);
    cout << endl;
    ptr = &Subtract;
    ptr(3,2);
}
```

PROGRAM 9.7

Output of Program 9.7:

```
1 + 2 = 3
3 - 2 = 1
```

9.3 Pointers to Objects

We have already seen how to use pointers to access the class members. As stated earlier, a pointer can point to an object created by a class. Consider the following statement:

```
item x;
```

where **item** is a class and x is an object defined to be of type item. Similarly we can define a pointer **it_ptr** of type **item** as follows:

```
item *it_ptr;
```

Object pointers are useful in creating objects at run time. We can also use an object pointer to access the public members of an object. Consider a class **item** defined as follows:

```
class item
{
      int code;
      float price;
   public:

      void getdata(int a, float b)
      {
         code = a;
         price = b;
      }

      void show(void)
      {
          cout << "Code : ". << code << "\n";
                << "Price: " << price << "\n\n";
      }
};
```

Let us declare an **item** variable **x** and a pointer **ptr** to **x** as follows:

```
item x;
item *ptr = &x;
```

The pointer **ptr** is initialized with the address of **x**.

We can refer to the member functions of **item** in two ways, one by using the *dot operator* and *the object*, and another by using the *arrow operator* and the *object pointer*. The statements

```
x.getdata(100,75.50);
x.show();
```

are equivalent to

```
ptr->getdata(100, 75.50);
ptr->show();
```

Since ***ptr** is an alias of **x**, we can also use the following method:

```
(*ptr).show();
```

The parentheses are necessary because the dot operator has higher precedence than the *indirection operator* *.

We can also create the objects using pointers and **new** operator as follows:

```
item *ptr = new item;
```

This statement allocates enough memory for the data members in the object structure and assigns the address of the memory space to **ptr**. Then **ptr** can be used to refer to the members as shown below:

```
ptr -> show();
```

If a class has a constructor with arguments and does not include an empty constructor, then we must supply the arguments when the object is created.

We can also create an array of objects using pointers. For example, the statement

```
item *ptr = new item[10];       // array of 10 objects
```

creates memory space for an array of 10 objects of **item**. Remember, in such cases, if the class contains constructors, it must also contain an empty constructor.

Program 9.8 illustrates the use of pointers to objects.

POINTERS TO OBJECTS

```
#include <iostream>

using namespace std;

class item
{
     int code;
     float price;
   public:
     void getdata(int a, float b)
     {
            code = a;
            price = b;
     }

     void show(void)
     {
            cout << "Code : " << code << "\n";
            cout << "Price: " << price << "\n";
     }
};

const int size = 2;

int main()
{
     item *p = new item [size];
     item *d = p;
     int x, i;
     float y;

     for(i=0; i<size; i++)
     {
            cout << "Input code and price for item" << i+1;
            cin >> x >> y;
            p->getdata(x,y);
            p++;
     }

     for(i=0; i<size; i++)
     {
            cout << "Item:" << i+1 << "\n";
```

(Contd)

```
                d->show();
                d++;
            }

            return 0;
        }
```

PROGRAM 9.8

The output of Program 9.8 will be:

```
Input code and price for item1 40 500
Input code and price for item2 50 600
Item:1
Code : 40
Price: 500
Item:2
Code : 50
Price: 600
```

In Program 9.8 we created space dynamically for two objects of equal size. But this may not be the case always. For example, the objects of a class that contain character strings would not be of the same size. In such cases, we can define an array of pointers to objects that can be used to access the individual objects. This is illustrated in Program 9.9.

ARRAY OF POINTERS TO OBJECTS

```
#include <iostream>
#include <cstring>

using namespace std;

class city
{
  protected:
       char *name;
       int len;
  public:
       city()

       {
              len = 0;
              name = new char[len+1];
```

(Contd)

```
        }
        void getname(void)
        {
                char *s;
                s = new char[30];

                cout << "Enter city name:";
                cin >> s;
                len = strlen(s);
                name = new char[len + 1];
                strcpy(name, s);
        }
        void printname(void)
        {
                cout << name << "\n";
        }
};

int main()
{
        city *cptr[10];            // array of 10 pointers to cities

        int n = 1;
        int option;

        do
        {
                cptr[n] = new city;  // create new city
                cptr[n]->getname();
                n++;
                cout << "Do you want to enter one more name?\n";
                cout << "(Enter 1 for yes 0 for no):";
                cin >> option;
        }
        while(option);

        cout << "\n\n";
        for(int i=1; i<=n; i++)
        {
                cptr[i]->printname();
        }

        return 0;
}
```

PROGRAM 9.9

The output of Program 9.9 would be:

```
Enter city name:Hyderabad
Do you want to enter one more name?
(Enter 1 for yes 0 for no);1
Enter city name:Secunderabad
Do you want to enter one more name?
(Enter 1 for yes 0 for no);1
Enter city name:Malkajgiri
Do you want to enter one more name?
(Enter 1 for yes 0 for no);0

Hyderabad
Secunderabad
Malkajgiri
```

9.4 this Pointer

C++ uses a unique keyword called **this** to represent an object that invokes a member function. **this** is a pointer that points to the object for which *this* function was called. For example, the function call **A.max()** will set the pointer **this** to the address of the object **A**. The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer **this** acts as an *implicit* argument to all the member functions. Consider the following simple example:

```
class ABC
{
      int a;
      .....
      .....
};
```

The private variable 'a' can be used directly inside a member function, like

```
a = 123;
```

We can also use the following statement to do the same job:

```
this->a = 123;
```

Since C++ permits the use of shorthand form **a = 123**, we have not been using the pointer **this** explicitly so far. However, we have been implicitly using the pointer **this** when overloading the operators using member function.

Recall that, when a binary operator is overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer **this**. One important application of the pointer **this** is to return the object it points to. For example, the statement

```
return *this;
```

inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare two or more objects inside a member function and return the *invoking object* as a result. Example:

```
person & person :: greater(person & x)
{
      if x.age > age
           return x;                    // argument object
      else
             return *this;             // invoking object
}
```

Suppose we invoke this function by the call

```
max = A.greater(B);
```

The function will return the object **B** (argument object) if the age of the person **B** is greater than that of **A**, otherwise, it will return the object **A** (invoking object) using the pointer **this**. Remember, the dereference operator * produces the contents at the address contained in the pointer. A complete program to illustrate the use of **this** is given in Program 9.10.

THIS POINTER

```
#include <iostream>
#include <cstring>

using namespace std;

class person
{
       char name[20];
       float age;
  public:
       person(char *s, float a)
       {
```

(Contd)

```
            strcpy(name, s);
            age = a;
        }
        person & person :: greater(person & x)

        {
            if(x.age >= age)
                return x;
            else
                return *this;
        }

        void display(void)
        {
            cout << "Name: " << name << "\n"
                 << "Age:  " << age << "\n";
        }
};

int main()
{
        person P1("John", 37.50),
               P2("Ahmed", 29.0),
               P3("Hebber", 40.25);

        person P = P1.greater(P3);        // P3.greater(P1)
        cout << "Elder person is: \n";
        P.display();

        P = P1.greater(P2);               // P2.greater(P1)
        cout << "Elder person is: \n";
        P.display();

        return 0;
}
```

PROGRAM 9.10

The output of Program 9.10 would be:

```
Elder person is:
Name: Hebber
Age:  40.25
Elder person is:
Name: John
Age:  37.5
```