8

Inheritance: Extending Classes

8.1 Introduction

Reusability is yet another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save time and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of **reusability**. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called **inheritance (or derivation)**. The old class is referred to as the **base class** and the new one is called the **derived class or subclass**.
The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class, is called single inheritance and one with several base classes is called multiple inheritance. On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another ‘derived class’ is known as multilevel inheritance. Figure 8.1 shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance. (Some authors show the arrow in opposite direction meaning “inherited from”.)

![Diagram showing forms of inheritance](image)

**Fig. 8.1 Forms of inheritance**

### 8.2 Defining Derived Classes

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:
class derived-class-name : visibility-mode base-class-name
{
    ....//
    ....// members of derived class
    ....//
};

The colon indicates that the derived-class-name is derived from the base-class-name. The visibility-mode is optional and, if present, may be either private or public. The default visibility-mode is private. Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

Examples:

class ABC: private XYZ // private derivation
{
    members of ABC
};

class ABC: public XYZ // public derivation
{
    members of ABC
};

class ABC: XYZ // private derivation by default
{
    members of ABC
};

When a base class is privately inherited by a derived class, 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. Remember, a public member of a class can be accessed by its own objects using the dot operator. The result is that no member of the base class is accessible to the objects of the derived class.

On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of the derived class and therefore they are accessible to the objects of the derived class. In both the cases, the private members are not inherited and therefore, the private members of a base class will never become the members of its derived class.

In inheritance, some of the base class data elements and member functions are 'inherited' into the derived class. We can add our own data and member functions and thus extend the
functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

### 8.3 Single Inheritance

Let us consider a simple example to illustrate inheritance. Program 8.1 shows a base class B and a derived class D. The class B contains one private data member, one public data member, and three public member functions. The class D contains one private data member and two public member functions.

```cpp
#include <iostream>

using namespace std;

class B
{
    int a; // private; not inheritable
    public:
        int b; // public; ready for inheritance
        void get_ab();
        int get_a(void);
        void show_a(void);
};

class D : public B // public derivation
{
    int c;
    public:
        void mul(void);
        void display(void);
};

void B :: get_ab(void)
{
    a = 5; b = 10;
}

int B :: get_a()
{
    return a;
}

void B :: show_a()
{

(Contd)
```
cout << "a = " << a << "\n";
}
void D :: mul()
{
    c = b * get_a();
}
void D :: display()
{
    cout << "a = " << get_a() << "\n";
    cout << "b = " << b << "\n";
    cout << "c = " << c << "\n\n";
}

int main()
{
    D d;
    d.get_ab();
    d.mul();
    d.show_a();
    d.display();
    d.b = 20;
    d.mul();
    d.display();
    return 0;
}

Given below is the output of Program 8.1:

a = 5
a = 5
b = 10
c = 50

a = 5
b = 20
c = 100

The class D is a public derivation of the base class B. Therefore, D inherits all the public members of B and retains their visibility. Thus a public member of the base class B is also a public member of the derived class D. The private members of B cannot be inherited.
by D. The class D, in effect, will have more members than what it contains at the time of declaration as shown in Fig. 8.2.

![Diagram of Class D]

**Fig. 8.2** Adding more members to a class (by public derivation)

The program illustrates that the objects of class D have access to all the public members of B. Let us have a look at the functions `show_a()` and `mul()`:

```cpp
void show_a()
{
    cout << "a = " << a << "\n";
}

void mul()
{
    c = b * get_a(); // c = b * a
}
```

Although the data member a is private in B and cannot be inherited, objects of D are able to access it through an inherited member function of B.

Let us now consider the case of private derivation.
class B
{
    int a;
    public:
    int b;
    void get_ab();
    void get_a();
    void show_a();
};

class D : private B // private derivation
{
    int c;
    public:
    void mul();
    void display();
};

The membership of the derived class D is shown in Fig. 8.3. In private derivation, the public members of the base class become private members of the derived class. Therefore, the objects of D cannot have direct access to the public member functions of B.
The statements such as

```cpp
d.get_ab();       // get_ab() is private
d.get_a();       // so also get_a()
d.show_a();      // and show_a()
```

will not work. However, these functions can be used inside `mul()` and `display()` like the normal functions as shown below:

```cpp
void mul()
{
    get_ab();
    c = b * get_a();
}

void display()
{
    show_a();          // outputs value of 'a'
    cout << "b = " << b << "\n"
         << "c = " << c << "\n\n";
}
```

Program 8.2 incorporates these modifications for private derivation. Please compare this with Program 8.1.

```
SINGLE INHERITANCE: PRIVATE

#include <iostream>

using namespace std;

class B
{
    int a;          // private; not inheritable
    public:
        int b;      // public; ready for inheritance
        void get_ab();
        int get_a(void);
        void show_a(void);
};

class D : private B // private derivation
{
    int c;
}(Contd)
```
public:
    void mul(void);
    void display(void);
};

void B :: get_ab(void)
{
    cout << "Enter values for a and b: ";
    cin >> a >> b;
}

int B :: get_a()
{
    return a;
}

void B :: show_a()
{
    cout << "a = " << a << "\n";
}

void D :: mul()
{
    get_ab();
    c = b * get_a(); // 'a' cannot be used directly
}

void D :: display()
{
    show_a(); // outputs value of 'a'
    cout << "b = " << b << "\n" << "c = " << c << "\n\n";
}

// ------------------------------------------------------------------

int main()
{
    D d;
    // d.get_ab(); WON'T WORK
    d.mul();
    // d.show a(); WON'T WORK
    d.display();

    (Cont'd)
// d.b = 20;            
WON'T WORK; b has become private 
d.mul(); 
d.display(); 

return 0;

The output of Program 8.2 would be:

Enter values for a and b: 5 10
a = 5 
b = 10 
c = 50 
Enter values for a and b: 12 20
a = 12 
b = 20 
c = 240

Suppose a base class and a derived class define a function of the same name. What will happen when a derived class object invokes the function? In such cases, the derived class function supersedes the base class definition. The base class function will be called only if the derived class does not redefine the function.

8.4 Making a Private Member Inheritable

We have just seen how to increase the capabilities of an existing class without modifying it. We have also seen that a private member of a base class cannot be inherited and therefore it is not available for the derived class directly. What do we do if the private data needs to be inherited by a derived class? This can be accomplished by modifying the visibility limit of the private member by making it public. This would make it accessible to all the other functions of the program, thus taking away the advantage of data hiding.

C++ provides a third visibility modifier, protected, which serve a limited purpose in inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes. A class can now use all the three visibility modes as illustrated below:

```c++
class alpha 
{
    private:                      // optional
        .......                      // visible to member functions
```
When a `protected` member is inherited in `public` mode, it becomes `protected` in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A `protected` member, inherited in the `private` mode derivation, becomes `private` in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since `private` members cannot be inherited). Figure 8.4 is the pictorial representation for the two levels of derivation.
The keywords **private**, **protected**, and **public** may appear in any order and any number of times in the declaration of a class. For example,

```cpp
class beta
{
    protected:
    ......
    public:
    ......
    private:
    ......
    public:
    ......
};
```

is a valid class definition.

However, the normal practice is to use them as follows:

```cpp
class beta
{
    ...... // private by default
    ......
    protected:
    ......
    public:
    ......
};
```

It is also possible to inherit a base class in **protected** mode (known as **protected derivation**). In protected derivation, both the **public** and **protected** members of the base class become **protected** members of the derived class. Table 8.1 summarizes how the visibility of base class members undergoes modifications in all the three types of derivation.

Now let us review the access control to the **private** and **protected** members of a class. What are the various functions that can have access to these members? They could be:

1. A function that is a friend of the class.
2. A member function of a class that is a friend of the class.
3. A member function of a derived class.

While the friend functions and the member functions of a friend class can have direct access to both the **private** and **protected** data, the member functions of a derived class can directly access only the **protected** data. However, they can access the **private** data through the member functions of the base class. Figure 8.5 illustrates how the access control...
mechanism works in various situations. A simplified view of access control to the members of a class is shown in Fig. 8.6.

<table>
<thead>
<tr>
<th>Base class visibility</th>
<th>Derived class visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>Public derivation</td>
</tr>
<tr>
<td></td>
<td>Not inherited</td>
</tr>
<tr>
<td></td>
<td>Protected derivation</td>
</tr>
<tr>
<td></td>
<td>Not inherited</td>
</tr>
<tr>
<td></td>
<td>Private</td>
</tr>
<tr>
<td></td>
<td>Protected</td>
</tr>
<tr>
<td></td>
<td>Public</td>
</tr>
</tbody>
</table>

**Table 8.1** Visibility of inherited members

![Diagram](image)

**Fig. 8.5** ⇔ Access mechanism in classes

### 8.5 Multilevel Inheritance

It is not uncommon that a class is derived from another derived class as shown in Fig. 8.7. The class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. The class B is known as intermediate base class since it provides a link for the inheritance between A and C. The chain ABC is known as inheritance path.
A derived class with multilevel inheritance is declared as follows:

```cpp
class A{.....}; // Base class
class B: public A {.....}; // B derived from A
class C: public B {.....}; // C derived from B
```

This process can be extended to any number of levels.

Let us consider a simple example. Assume that the test results of a batch of students are stored in three different classes. Class `student` stores the roll-number, class `test` stores the marks obtained in two subjects and class `result` contains the total marks obtained in the test. The class `result` can inherit the details of the marks obtained in the test and the roll-number of students through multilevel inheritance. Example:
class student
{
    protected:
    int roll_number;
    public:
    void get_number(int);
    void put_number(void);
};
void student :: get_number(int a)
{
    roll_number = a;
}
void student :: put_number()
{
    cout << "Roll Number: " << roll_number << "\n";
}

class test : public student // First level derivation
{
    protected:
    float sub1;
    float sub2;
    public:
    void get_marks(float, float);
    void put_marks(void);
};
void test :: get_marks(float x, float y)
{
    sub1 = x;
    sub2 = y;
}
void test :: put_marks()
{
    cout << "Marks in SUB1 = " << sub1 << "\n";
    cout << "Marks in SUB2 = " << sub2 << "\n";
}
class result : public test // Second level derivation
{
    float total; // private by default
    public:
    void display(void);
};

The class **result**, after inheritance from ‘grandfather’ through ‘father’, would contain the following members:
private:
  float total;  // own member
protected:
  int roll_number;  // inherited from student via test
  float sub1;  // inherited from test
  float sub2;  // inherited from test
public:
  void get_number(int);  // from student via test
  void put_number(void);  // from student via test
  void get_marks(float, float);  // from test
  void put_marks(void);  // from test
  void display(void);  // own member

The inherited functions put_number() and put_marks() can be used in the definition of display() function:

```cpp
void result :: display(void)
{
  total = sub1 + sub2;
  put_number();
  put_marks();
  cout << "Total = " << total << "\n";
}
```

Here is a simple main() program:

```cpp
int main()
{
  result student1;  // student1 created
  student1.get_number(111);
  student1.get_marks(75.0, 59.5);
  student1.display();

  return 0;
}
```

This will display the result of student1. The complete program is shown in Program 8.3.

```cpp
MULTILEVEL INHERITANCE
#include <iostream>
using namespace std;

class student

(Contd)
```
Inheritance: Extending Classes

```cpp
{
    protected:
        int roll_number;
    public:
        void get_number(int);
        void put_number(void);
};

void student :: get_number(int a)
{
    roll_number = a;
}

void student :: put_number()
{
    cout << "Roll Number: " << roll_number << "\n";
}

class test : public student // First level derivation
{
    protected:
        float sub1;
        float sub2;
    public:
        void get_marks(float, float);
        void put_marks(void);
};

void test :: get_marks(float x, float y)
{
    sub1 = x;
    sub2 = y;
}

void test :: put_marks()
{
    cout << "Marks in SUB1 = " << sub1 << "\n";
    cout << "Marks in SUB2 = " << sub2 << "\n";
}

class result : public test // Second level derivation // private by default
{
    float total;
    public:
        void display(void);
};

void result :: display(void)
{
(Contd)
```
total = sub1 + sub2;
put_number();
put_marks();
cout << "Total = " << total << "\n";
}

int main()
{
    result student1;    // student1 created
    student1.get_number(111);
    student1.get_marks(75.0, 59.5);
    student1.display();
    return 0;
}

Program 8.3 displays the following output:

Roll Number: 111
Marks in SUB1 = 75
Marks in SUB2 = 59.5
Total = 134.5

8.6 Multiple Inheritance

A class can inherit the attributes of two or more classes as shown in Fig. 8.8. This is known as multiple inheritance. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another.

![Diagram of multiple inheritance]

Fig. 8.8 ⇔ Multiple inheritance
The syntax of a derived class with multiple base classes is as follows:

```c++
class D: visibility B-1, visibility B-2 ...
{
    .....
    ...(Body of D)
    .....
};
```

where, *visibility* may be either `public` or `private`. The base classes are separated by commas.

Example:

```c++
class P : public M, public N
{
    public:
    void display(void);
};
```

Classes M and N have been specified as follows:

```c++
class M
{
    protected:
    int m;
    public:
    void get_m(int);
};
void M :: get_m(int x)
{
    m = x;
}
class N
{
    protected:
    int n;
    public:
    void get_n(int);
};
void N :: get_n(int y)
{
```

---

Copyrighted material
n = y;

}

The derived class \texttt{P}, as declared above, would, in effect, contain all the members of \texttt{M} and \texttt{N} in addition to its own members as shown below:

```cpp
class P
{
    protected:
    int m; // from M
    int n; // from N

d public:
    void get_m(int); // from M
    void get_n(int); // from N
    void display(void); // own member
};
```

The member function \texttt{display()} can be defined as follows:

```cpp
void P :: display(void)
{
    cout << "m = " << m << "\n";
    cout << "n = " << n << "\n";
    cout << "m*n =" << m*n << "\n";
}
```

The main() function which provides the user-interface may be written as follows:

```cpp
main()
{
    P p;
    p.get_m(10);
    p.get_n(20);
    p.display();
}
```

Program 8.4 shows the entire code illustrating how all the three classes are implemented in multiple inheritance mode.
#include <iostream>

using namespace std;

class M
{
  protected:
    int m;
  public:
    void get_m(int);
};

class N
{
  protected:
    int n;
  public:
    void get_n(int);
};

class P : public M, public N
{
  public:
    void display(void);
};

void M :: get_m(int x)
{
  m = x;
}

void N :: get_n(int y)
{
  n = y;
}

void P :: display(void)
{
  cout << "m = " << m << "\n";
  cout << "n = " << n << "\n";
  cout << "m*n = " << m*n << "\n";
}

int main()
{

  (Contd)
The output of Program 8.4 would be:

\[
\begin{align*}
m &= 10 \\
n &= 20 \\
m \times n &= 200
\end{align*}
\]

**Ambiguity Resolution in Inheritance**

Occasionally, we may face a problem in using the multiple inheritance, when a function with the same name appears in more than one base class. Consider the following two classes.

```cpp
class M
{
public:
    void display(void)
    {
        cout << "Class M\n";
    }
};

class N
{
public:
    void display(void)
    {
        cout << "Class N\n";
    }
};
```

Which `display()` function is used by the derived class when we inherit these two classes? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function as shown below:

```cpp
class P : public M, public N
```
{  
    public:
    void display(void)  // overrides display() of M and N
    {
        M :: display();
    }
};

We can now use the derived class as follows:

    int main()
    {
        P p;
        p.display();
    }

Ambiguity may also arise in single inheritance applications. For instance, consider the following situation:

class A
{
    public:
    void display()
    {
        cout << "A\n";
    }
};
class B : public A
{
    public:
    void display()
    {
        cout << "B\n";
    }
};

In this case, the function in the derived class overrides the inherited function and, therefore, a simple call to display() by B type object will invoke function defined in B only. However, we may invoke the function defined in A by using the scope resolution operator to specify the class.

Example:

    int main()
    {
B b;  // derived class object
b.display();  // invokes display() in B
b.A::display();  // invokes display() in A
b.B::display();  // invokes display() in B

return 0;

This will produce the following output:

B
A
B

8.7 Hierarchical Inheritance

We have discussed so far how inheritance can be used to modify a class when it did not satisfy the requirements of a particular problem on hand. Additional members are added through inheritance to extend the capabilities of a class. Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.

As an example, Fig. 8.9 shows a hierarchical classification of students in a university. Another example could be the classification of accounts in a commercial bank as shown in Fig. 8.10. All the students have certain things in common and, similarly, all the accounts possess certain common features.

![Hierarchical classification of students](image)
In C++, such problems can be easily converted into class hierarchies. The base class will include all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.

8.8 Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. For instance, consider the case of processing the student results discussed in Sec. 8.5. Assume that we have to give weightage for sports before finalising the results. The weightage for sports is stored in a separate class called sports. The new inheritance relationship between the various classes would be as shown in Fig. 8.11.
The **sports** class might look like:

```cpp
class sports {
    protected:
        float score;
    public:
        void get_score(float);
        void put_score(void);
};
```

The result will have both the multilevel and multiple inheritances and its declaration would be as follows:

```cpp
class result : public test, public sports {
    ..... 
    ..... 
};
```

Where **test** itself is a derived class from **student**. That is

```cpp
class test : public student {
    ..... 
    ..... 
};
```

Program 8.5 illustrates the implementation of both multilevel and multiple inheritance.

```cpp
#include <iostream>

using namespace std;

class student {
    protected:
        int roll_number;
    public:
        void get_number(int a) {
            roll_number = a;
        }
};

// Contd
```

Copyrighted material
```cpp
void put_number(void)
{
    cout << "Roll No: " << roll_number << "\n";
}
}

class test : public student
{
    protected:
    float part1, part2;
    public:
    void get_marks(float x, float y)
    {
        part1 = x; part2 = y;
    }
    void put_marks(void)
    {
        cout << "Marks obtained: " << "\n"
            << "Part1 = " << part1 << "\n"
            << "Part2 = " << part2 << "\n";
    }
};

class sports
{
    protected:
    float score;
    public:
    void get_score(float s)
    {
        score = s;
    }
    void put_score(void)
    {
        cout << "Sports wt: " << score << "\n\n";
    }
};

class result : public test, public sports
{
    float total;
    public:
    void display(void);

(Contd)
```
Here is the output of Program 8.5:

Roll No: 1234  
Marks obtained:  
Part1 = 27.5  
Part2 = 33  
Sports wt: 6  

Total Score: 66.5

**8.9 Virtual Base Classes**

We have just discussed a situation which would require the use of both the multiple and multilevel inheritance. Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in Fig. 8.12. The 'child' has two *direct base classes* 'parent1' and 'parent2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as *indirect base class*. 

Copyrighted material
Inheritance by the 'child' as shown in Fig. 8.12 might pose some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. This means, 'child' would have duplicate sets of the members inherited from 'grandparent'. This introduces ambiguity and should be avoided.

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as virtual base class while declaring the direct or intermediate base classes as shown below:

```cpp
class A  // grandparent
{
    ....
    ....
};
class B1 : virtual public A  // parent1
{
    ....
    ....
};
class B2 : public virtual A  // parent2
{
    ....
    ....
};
class C : public B1, public B2  // child
{
    ....  // only one copy of A
    ....  // will be inherited
};
```

When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.
For example, consider again the student results processing system discussed in Sec. 8.8. Assume that the class `sports` derives the `roll_number` from the class `student`. Then, the inheritance relationship will be as shown in Fig. 8.13.

A program to implement the concept of virtual base class is illustrated in Program 8.6.

```cpp
VIRTUAL BASE CLASS
#include <iostream>

using namespace std;

class student
{
    protected:
        int roll_number;
    public:
        void get_number(int a)
        {
        }
}

(Contd)
```
roll_number = a;
}
void put_number(void)
{
    cout << "Roll No: " << roll_number << "\n";
}

class test : virtual public student
{
    protected:
        float part1, part2;
    public:
        void get_marks(float x, float y)
        {
            part1 = x; part2 = y;
        }
        void put_marks(void)
        {
            cout << "Marks obtained: " << "\n"
               << "Part1 = " << part1 << "\n"
               << "Part2 = " << part2 << "\n";
        }
};

class sports : public virtual student
{
    protected:
        float score;
    public:
        void get_score(float s)
        {
            score = s;
        }
        void put_score(void)
        {
            cout << "Sports wt: " << score << "\n\n";
        }
};
class result : public test, public sports
{
    float total;
    public:
        void display(void);
};

(Contd)
void result :: display(void)
{
    total = part1 + part2 + score;
    put_number();
    put_marks();
    put_score();
    cout << "Total Score: " << total << "\n";
}

int main()
{
    result student_1;
    student_1.get_number(678);
    student_1.get_marks(30.5, 25.5);
    student_1.get_score(7.0);
    student_1.display();

    return 0;
}

The output of Program 8.6 would be

Roll No: 678
Marks obtained:
Part1 = 30.5
Part2 = 25.5
Sport wt: 7

Total Score: 63

8.10 Abstract Classes

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes). It is a design concept in program development and provides a base upon which other classes may be built. In the previous example, the student class is an abstract class since it was not used to create any objects.

8.11 Constructors in Derived Classes

As we know, the constructors play an important role in initializing objects. We did not use them earlier in the derived classes for the sake of simplicity. One important thing to note
here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. Remember, while applying inheritance we usually create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance.

Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. How are they passed to the base class constructors so that they can do their job? C++ supports a special argument passing mechanism for such situations.

The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor.

The general form of defining a derived constructor is:

```
Derived-constructor (Arglist1, Arglist2, ... ArglistN, Arglist(D)
    base1(arglist1),
    base2(arglist2),
    ....
    ....
    baseN(arglistN),
    { arguments for base(N)
      Body of derived constructor
    }
```

The header line of `derived-constructor` function contains two parts separated by a colon (:). The first part provides the declaration of the arguments that are passed to the `derived-constructor` and the second part lists the function calls to the base constructors.

`base1(arglist1), base2(arglist2) ...` are function calls to base constructors `base1(), base2(), ...` and therefore `arglist1, arglist2, ...` etc. represent the actual parameters that are passed to the base constructors. `Arglist1 through ArglistN` are the argument declarations for base constructors `base1 through baseN`. `ArglistD` provides the parameters that are necessary to initialize the members of the derived class.
Example:

```cpp
D(int a1, int a2, float b1, float b2, int d1):
A(a1, a2),    /* call to constructor A */
B(b1, b2)     /* call to constructor B */
{
    d = d1;    // executes its own body
}
```

A(a1, a2) invokes the base constructor A() and B(b1, b2) invokes another base constructor B(). The constructor D() supplies the values for these four arguments. In addition, it has one argument of its own. The constructor D() has a total of five arguments. D() may be invoked as follows:

```cpp
......
D objD(5, 12, 2.5, 7.54, 30);
......
```

These values are assigned to various parameters by the constructor D() as follows:

<table>
<thead>
<tr>
<th></th>
<th>→</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>→  a1</td>
</tr>
<tr>
<td>12</td>
<td>→  a2</td>
</tr>
<tr>
<td>2.5</td>
<td>→  b1</td>
</tr>
<tr>
<td>7.54</td>
<td>→  b2</td>
</tr>
<tr>
<td>30</td>
<td>→  d1</td>
</tr>
</tbody>
</table>

The constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared. Any non-virtual bases are then constructed before the derived class constructor is executed. See Table 8.2.

**Table 8.2** Execution of base class constructors

<table>
<thead>
<tr>
<th>Method of inheritance</th>
<th>Order of execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class B: public A</td>
<td>A( ); base constructor</td>
</tr>
<tr>
<td></td>
<td>B( ); derived constructor</td>
</tr>
<tr>
<td>class A : public B, public C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B( ); base(first)</td>
</tr>
<tr>
<td></td>
<td>C( ); base(second)</td>
</tr>
<tr>
<td></td>
<td>A( ); derived</td>
</tr>
<tr>
<td>class A : public B, virtual public C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C( ); virtual base</td>
</tr>
<tr>
<td></td>
<td>B( ); ordinary base</td>
</tr>
<tr>
<td></td>
<td>A( ); derived</td>
</tr>
</tbody>
</table>

Copyrighted material
Program 8.7 illustrates how constructors are implemented when the classes are inherited.

```cpp
#include <iostream>

using namespace std;

class alpha
{
    int x;
    public:
        alpha(int i)
        {
            x = i;
            cout << "alpha initialized \n";
        }
        void show_x(void)
        { cout << "x = " << x << "\n"; }
};

class beta
{
    float y;
    public:
        beta(float j)
        {
            y = j;
            cout << "beta initialized \n";
        }
        void show_y(void)
        { cout << "y = " << y << "\n"; }
};

class gamma: public beta, public alpha
{
    int m, n;
    public:
        gamma(int a, float b, int c, int d):
            alpha(a), beta(b)
        {
            m = c;
            n = d;
            cout << "gamma initialized \n";
        }

(Contd)
void show_mn (void)
{
    cout << "m = " << m << "\n"
    << "n = " << n << "\n";
}

int main()
{
    gamma g(5, 10.75, 20, 30);
    cout << "\n";
    g.show_x();
    g.show_y();
    g.show_mn();
    return 0;
}

The output of Program 8.7 would be:

beta initialized
alpha initialized
gamma initialized

x = 5
y = 10.75
m = 20
n = 30

**note**

*beta* is initialized first, although it appears second in the derived constructor. This is because it has been declared first in the derived class header line. Also, note that *alpha(a)* and *beta(b)* are function calls. Therefore, the parameters should not include types.

C++ supports another method of initializing the class objects. This method uses what is known as initialization list in the constructor function. This takes the following form:

```
constructor (arglist) : initialization-section
{
    assignment-section
}
```

The *assignment-section* is nothing but the body of the constructor function and is used to assign initial values to its data members. The part immediately following the colon is known
as the *initialization section*. We can use this section to provide initial values to the base constructors and also to initialize its own class members. This means that we can use either of the sections to initialize the data members of the constructors class. The initialization section basically contains a list of initializations separated by commas. This list is known as *initialization list*. Consider a simple example:

```cpp
class XYZ
{
    int a;
    int b;
    public:
        XYZ(int i, int j) : a(i), b(2 * j) { }
};

main()
{
    XYZ x(2, 3);
}
```

This program will initialize `a` to 2 and `b` to 6. Note how the data members are initialized, just by using the variable name followed by the initialization value enclosed in the parenthesis (like a function call). Any of the parameters of the argument list may be used as the initialization value and the items in the list may be in any order. For example, the constructor `XYZ` may also be written as:

```cpp
XYZ(int i, int j) : b(i), a(i + j) { }
```

In this case, `a` will be initialized to 5 and `b` to 2. Remember, the data members are initialized in the order of declaration, independent of the order in the initialization list. This enables us to have statements such as

```cpp
XYZ(int i, int j) : a(i), b(a * j) { }
```

Here `a` is initialized to 2 and `b` to 6. Remember, `a` which has been declared first is initialized first and then its value is used to initialize `b`. However, the following will not work:

```cpp
XYZ(int i, int j) : b(i), a(b * j) { }
```

because the value of `b` is not available to `a` which is to be initialized first.

The following statements are also valid:

```cpp
XYZ(int i, int j) : a(i) { b = j; }
XYZ(int i, int j) { a = i; b = j; }
```
We can omit either section, if it is not needed. Program 8.8 illustrates the use of initialization lists in the base and derived constructors.

```cpp
#include <iostream>

using namespace std;

class alpha
{
    int x;
public:
    alpha(int i)
    {
        x = i;
        cout << "\n alpha constructed";
    }

    void show_alpha(void)
    {
        cout << " x = " << x << "\n";
    }
};

class beta
{
    float p, q;
public:
    beta(float a, float b): p(a), q(b+p)
    {
        cout << "\n beta constructed";
    }

    void show_beta(void)
    {
        cout << " p = " << p << "\n";
        cout << " q = " << q << "\n";
    }
};
class gamma : public beta, public alpha
{
    int u,v;
public:

(Contd)
gamma(int a, int b, float c):
  alpha(a*2), beta(c, c), u(a)
  \{ v = b; cout << "\n gamma constructed\n"; \}

void show_gamma(void)
  \{
  cout << " u = " << u << "\n"
  cout << " v = " << v << "\n"
  \}

int main()
  \{
  gamma g(2, 4, 2.5);
  cout << "\n\n Display member values " << "\n\n"
  g.show_alpha();
  g.show_beta();
  g.show_gamma();
  return 0;
  \};

The output of Program 8.8 would be:

beta constructed
alpha constructed
gamma constructed

Display member values

x = 4
p = 2.5
q = 5
u = 2
v = 4

note

The argument list of the derived constructor gamma contains only three parameters a, b and c which are used to initialize the five data members contained in all the three classes.
8.12 Member Classes: Nesting of Classes

Inheritance is the mechanism of deriving certain properties of one class into another. We have seen in detail how this is implemented using the concept of derived classes. C++ supports yet another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its members as shown below:

```cpp
class alpha {....};
class beta {....};
class gamma
{
    alpha a;               // a is an object of alpha class
    beta b;                // b is an object of beta class
    ....
};
```

All objects of `gamma` class will contain the objects `a` and `b`. This kind of relationship is called `containership` or `nesting`. Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then the other 'ordinary' members are created. This means, constructors of all the member objects should be called before its own constructor body is executed. This is accomplished using an initialization list in the constructor of the nested class.

Example:

```cpp
class gamma
{
    ....
    alpha a;               // a is object of alpha
    beta b;                // b is object of beta
    public:
    gamma(arglist): a(arglist1), b(arglist2)
    {
        // constructor body
    }
};
```

`arglist` is the list of arguments that is to be supplied when a `gamma` object is defined. These parameters are used for initializing the members of `gamma`. `arglist1` is the argument list
for the constructor of a and arglist2 is the argument list for the constructor of b. arglist1 and arglist2 may or may not use the arguments from arglist. Remember, a(arglist1) and b(arglist2) are function calls and therefore the arguments do not contain the data types. They are simply variables or constants.

Example:

```c
gamma(int x, int y, float z) : a(x), b(x, z)
{
    Assignment section(for ordinary other members)
}
```

We can use as many member objects as are required in a class. For each member object we add a constructor call in the initializer list. The constructors of the member objects are called in the order in which they are declared in the nested class.

SUMMARY

- The mechanism of deriving a new class from an old class is called inheritance. Inheritance provides the concept of reusability. The C++ classes can be reused using inheritance.
- The derived class inherits some or all of the properties of the base class.
- A derived class with only one base class is called single inheritance.
- A class can inherit properties from more than one class which is known as multiple inheritance.
- A class can be derived from another derived class which is known as multilevel inheritance.
- When the properties of one class are inherited by more than one class, it is called hierarchical inheritance.
- A private member of a class cannot be inherited either in public mode or in private mode.
- A protected member inherited in public mode becomes protected, whereas inherited in private mode becomes private in the derived class.
- A public member inherited in public mode becomes public, whereas inherited in private mode becomes private in the derived class.
- The friend functions and the member functions of a friend class can directly access the private and protected data.
The member functions of a derived class can directly access only the protected and public data. However, they can access the private data through the member functions of the base class.

Multipath inheritance may lead to duplication of inherited members from a 'grandparent' base class. This may be avoided by making the common base class a virtual base class.

In multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.

In multilevel inheritance, the constructors are executed in the order of inheritance.

A class can contain objects of other classes. This is known as containership or nesting.

Key Terms

- abstract class
- access control
- access mechanism
- ancestor class
- assignment section
- base class
- base constructor
- child class
- common base class
- containership
- derivation
- derived class
- derived constructor
- direct base class
- dot operator
- duplicate members
- father class
- friend
- grandfather class
- grandparent class
- hierarchical inheritance
- hybrid inheritance
- indirect base class
- inheritance
- inheritance path
- initialization list
- initialization section
- intermediate base
- member classes
- multilevel inheritance
- multiple inheritance
- nesting
- private
- private derivation
- private members
- privately derived
- protected
- protected members
- public
- public derivation
- public members
- publicly derived
- reusability
- single inheritance
- subclass
- virtual base class
- visibility mode
- visibility modifier
Review Questions

8.1 What does inheritance mean in C++?
8.2 What are the different forms of inheritance? Give an example for each.
8.3 Describe the syntax of the single inheritance in C++.
8.4 We know that a private member of a base class is not inheritable. Is it anyway possible for the objects of a derived class to access the private members of the base class? If yes, how? Remember, the base class cannot be modified.
8.5 How do the properties of the following two derived classes differ?
   (a) class D1: private B(/ / ...);
   (b) class D2: public B(/ / ...);
8.6 When do we use the protected visibility specifier to a class member?
8.7 Describe the syntax of multiple inheritance. When do we use such an inheritance?
8.8 What are the implications of the following two definitions?
   (a) class A: public B, public C(/ / ....);
   (b) class A: public C, public B(/ / ....);
8.9 What is a virtual base class?
8.10 When do we make a class virtual?
8.11 What is an abstract class?
8.12 In what order are the class constructors called when a derived class object is created?
8.13 Class D is derived from class B. The class D does not contain any data members of its own. Does the class D require constructors? If yes, why?
8.14 What is containership? How does it differ from inheritance?
8.15 Describe how an object of a class that contains objects of other classes created?
8.16 State whether the following statements are TRUE or FALSE:
   (a) Inheritance helps in making a general class into a more specific class.
   (b) Inheritance aids data hiding.
   (c) One of the advantages of inheritance is that it provides a conceptual framework.
   (d) Inheritance facilitates the creation of class libraries.
   (e) Defining a derived class requires some changes in the base class.
   (f) A base class is never used to create objects.
   (g) It is legal to have an object of one class as a member of another class.
   (h) We can prevent the inheritance of all members of the base class by making base class virtual in the definition of the derived class.

Debugging Exercises

8.1 Identify the error in the following program.

    #include <iostream.h>
class Student {
    char* name;
    int rollNumber;
private:
    Student() {
        name = "AlanKay";
        rollNumber = 1025;
    }
    void setNumber(int no) {
        rollNumber = no;
    }
    int getRollNumber() {
        return rollNumber;
    }
};

class AnualTest: Student {
    int mark1, mark2;
public:
    AnualTest(int m1, int m2) :
        mark1(m1), mark2(m2) {
    }
    int getRollNumber() {
        return Student::getRollNumber();
    }
};

void main()
{
    AnualTest test1(92, 85);
    cout << test1.getRollNumber();
}

8.2 Identify the error in the following program.

#include <iostream.h>
class A
{
public:
    A()
    {

}
cout << "A";
}
}
class B: public A
{
public:
    B()
    {
        cout << "B";
    }
}
class C: public B
{
public:
    C()
    {
        cout << "C";
    }
}
class D
{
public:
    D()
    {
        cout << "D";
    }
}
class E: public C, public D
{
public:
    E()
    {
        cout << "D";
    }
}
class F: B, virtual E
{
public:
    F()
```cpp
{
    cout << "F";
}
void main()
{
    F f;
}
```

8.3 Identify the error in the following program.

```cpp
#include <iostream.h>
class A
{
    int i;
};
class AB: virtual A
{
    int j;
};
class AC: A, ABAC
{
    int k;
};
class ABAC: AB, AC
{
    int l;
};
void main()
{
    ABAC abac;
    cout << "sizeof ABAC: " << sizeof(abac);
}
```

8.4 Find errors in the following program. State reasons.

```cpp
// Program test
#include <iostream.h>
class X
```
```cpp
{  
    private:  
        int x1;  
    protected:  
        int x2;  
    public:  
        int x3;
};

class Y: public X
{  
    public:  
        void f()
        {
            int y1,y2,y3;  
            y1 = x1;  
            y2 = x2;  
            y3 = x3;
        }
};

class Z: X
{  
    public:  
        void f()
        {
            int z1,z2,z3;  
            z1 = x1;  
            z2 = x2;  
            z3 = x3;
        }
};

main()
{
    int m,n,p;  
    Y y;  
    m = y.x1;  
    n = y.x2;  
    p = y.x3;  
    Z z;  
    m = z.x1;  
    n = z.x2;  
    p = z.x3;
}
8.5 Debug the following program.

```cpp
// Test program
#include <iostream.h>

class B1
{
    int b1;
    public:
    void display()
    {
        cout << b1 << "\n";
    }
};

class B2
{
    int b2;
    public:
    void display()
    {
        cout << b2 << "\n";
    }
};
class D: public B1, public B2
{
    // nothing here
};
main()
{
    D d;
    d.display()
    d.B1::display();
    d.B2::display();
}
```

---

**Programming Exercises**

8.1 Assume that a bank maintains two kinds of accounts for customers, one called as savings account and the other as current account. The savings account provides compound interest and withdrawal facilities but no cheque book facility. The current account provides cheque book facility but no interest. Current account holders should also maintain a minimum balance and if the balance falls below this level, a service charge is imposed.
Create a class **account** that stores customer name, account number and type of account. From this derive the classes **cur_acct** and **sav_acct** to make them more specific to their requirements. Include necessary member functions in order to achieve the following tasks:

(a) Accept deposit from a customer and update the balance.
(b) Display the balance.
(c) Compute and deposit interest.
(d) Permit withdrawal and update the balance.
(e) Check for the minimum balance, impose penalty, necessary, and update the balance.

Do not use any constructors. Use *member functions* to initialize the class members.

8.2 Modify the program of Exercise 8.1 to include constructors for all the three classes.

8.3 An educational institution wishes to maintain a database of its employees. The database is divided into a number of classes whose hierarchical relationships are shown in Fig. 8.14. The figure also shows the minimum information required for each class. Specify all the classes and define functions to create the database and retrieve individual information as and when required.

![Class relationships](image)

8.4 The database created in Exercise 8.3 does not include educational information of the staff. It has been decided to add this information to teachers and officers (and not for typists) which will help the management in decision making with regard to training, promotion, etc. Add another data class called **education** that holds
two pieces of educational information, namely, highest qualification in general education and highest professional qualification. This class should be inherited by the classes teacher and officer. Modify the program of Exercise 8.19 to incorporate these additions.

8.5 Consider a class network of Fig. 8.15. The class master derives information from both account and admin classes which in turn derive information from the class person. Define all the four classes and write a program to create, update and display the information contained in master objects.

![Class Network Diagram]

Fig. 8.15  Multipath inheritance (for Exercise 8.21)

8.6 In Exercise 8.3, the classes teacher, officer, and typist are derived from the class staff. As we know, we can use container classes in place of inheritance in some situations. Redesign the program of Exercise 8.3 such that the classes teacher, officer, and typist contain the objects of staff.

8.7 We have learned that OOP is well suited for designing simulation programs. Using the techniques and tricks learned so far, design a program that would simulate a simple real-world system familiar to you.
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:

```cpp
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.

---

![Polymorphism Diagram](image)

**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:

```cpp
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.

```
#include <iostream.h>
#include <conio.h>
void main()
{
    int a, *ptr1, **ptr2;
    clrscr();
    ptr1 = &a;
    ptr2=&ptr1;
    cout << "The address of a : " << ptr1 << "\n";
    cout << "The address of ptr1 : " << ptr2;
    cout << "\n\n";
    cout << "After incrementing the address values:\n\n";
    ptr1+=2;
    cout << "The address of a : " << ptr1 << "\n";
    ptr2+=2;
    cout << "The address of ptr1 : " << 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 ptr1: 0x8fb6fff2
- After incrementing the address values:
  - The address of a: 0x8fb6fff8
  - The address of a: 0x8fb6ffff

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++.

```
MANIPULATION OF POINTERS

#include <iostream.h>
#include <conio.h>
void main()

(Contd)
```
```cpp
{ 
  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";
}```

### 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:**

```cpp
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

```c++
#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:
";
    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();

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:

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

Or

```c
nptr=number;
```

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

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

Or

```c
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**

```c
#include <iostream.h>

void main()
{
    int numbers[50], *ptr;
    int n,i;
    cout << "\nEnter the count\n";
    cin >> n;
    (Contd)
```
cout << "Enter 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 << "Sum of even numbers: " << sum;

**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:

```c
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.

```c
#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 << "Enter your favorite leisure pursuit: " ;
    cin >> str;
    for(i=0; i<4; i++)
    {
        if(!strcmp(str, *ptr[i]))
        {
            cout << "Your favorite pursuit " "is available here"
            << endl;
            break;
        }
    }

(Cont'd)
```

Copyrighted material
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:

```c
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.


```c++
{
    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;
}
```

Output of Program 9.6:

`TTTTeeeesssstttttt 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:

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

As we know, the `data_type` is any valid data type 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:

```c
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++.

```c
#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:

\[\begin{align*}
1 + 2 &= 3 \\
3 - 2 &= 1
\end{align*}\]

### 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:

```cpp
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:

```cpp
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:

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

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

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

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

```cpp
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

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

are equivalent to

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

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

```cpp
(*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 the *new operator* as follows:

```cpp
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:

```cpp
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

```cpp
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.
#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)
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.

```c++
#include <iostream>
#include <cstring>

using namespace std;

class city
{
    protected:
        char *name;
        int len;
    public:
        city()
        {
            len = 0;
            name = new char[len+1];
        }
```
void getline(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 printline(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]->getline();
        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;
}
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:

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

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

```cpp
a = 123;
```

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

```cpp
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.

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

using namespace std;

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

(Cont'd)
```cpp
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;
}
```

The output of Program 9.10 would be:

Elder person is:
Name: Hebber
Age: 40.25
Elder person is:
Name: John
Age: 37.5
9.5 Pointers to Derived Classes

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type-compatible with pointers to objects of a derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes. For example, if $B$ is a base class and $D$ is a derived class from $B$, then a pointer declared as a pointer to $B$ can also be a pointer to $D$. Consider the following declarations:

```c
B *cptr;  // pointer to class B type variable
B b;      // base object
D d;      // derived object
cptr = &b; // cptr points to object b
```

We can make `cptr` to point to the object $d$ as follows:

```c
cptr = &d;  // cptr points to object d
```

This is perfectly valid with C++ because $d$ is an object derived from the class $B$.

However, there is a problem in using `cptr` to access the public members of the derived class $D$. Using `cptr`, we can access only those members which are inherited from $B$ and not the members that originally belong to $D$. In case a member of $D$ has the same name as one of the members of $B$, then any reference to that member by `cptr` will always access the base class member.

Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer declared as pointer to the derived type.

Program 9.11 illustrates how pointers to a derived object are used.

```c
POINTERS TO DERIVED OBJECTS
#include <iostream>
using namespace std;

class BC
{
    public:
        int b;
        void show()
        { cout << "b = " << b << "\n"; }
};
```

(Contd)
class DC : public BC
{
    public:
        int d;
        void show()
        { cout << "b = " << b << "\n"
            << "d = " << d << "\n";
        }
};

int main()
{
    BC *bp;
    BC base;
    bp = &base;
    // base address
    bp->b = 100;
    // access BC via base pointer
    cout << "bp points to base object \n";
    bp->show();
    // derived class
    DC derived;
    bp = &derived;
    // address of derived object
    bp->b = 200;
    // access DC via base pointer
    /* bp->d = 300;*/
    // won't work
    cout << "bp now points to derived object \n";
    bp->show();
    // bp now points to derived object

    /* accessing d using a pointer of type derived class DC */
    DC *dp;
    dp = &derived;
    dp->d = 300;
    cout << "dp is derived type pointer\n";
    dp->show();

    cout << "using ((DC *)bp)\n";
    ((DC *)bp) -> d = 400;
    ((DC *)bp) -> show();

    return 0;
}

PROGRAM 9.11
Program 9.11 produces the following output:

```
bptr points base object
bptr now points to derived object
bptr is derived type pointer
using ((DC *)bptr)
b = 200
d = 400
```

**Note**

We have used the statement

```
bptr -> show();
```

two times. First, when `bptr` points to the base object, and second when `bptr` is made to point to the derived object. But, both the times, it executed `BC::show()` function and displayed the content of the base object. However, the statements

```
dptr -> show();
((DC *) bptr) -> show(); // cast bptr to DC type
```

display the contents of the derived object. This shows that, although a base pointer can be made to point to any number of derived objects, it cannot directly access the members defined by a derived class.

### 9.6 Virtual Functions

As mentioned earlier, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes. This necessitates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the derived objects. But, we just discovered that a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. How do we then achieve polymorphism? It is achieved using what is known as ‘virtual’ functions.
When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration. When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function. Program 9.12 illustrates this point.

```cpp
#include <iostream>

using namespace std;

class Base {
    public:
        void display() {cout << "\n Display base ";}
        virtual void show() {cout << "\n show base";}
};
class Derived : public Base {
    public:
        void display() {cout << "\n Display derived";}
        void show() {cout << "\n show derived";}
};

int main() {
    Base B;
    Derived D;
    Base *bptr;

    cout << "\n bptr points to Base \n";
    bptr = &B;
    bptr -> display();    // calls Base version
    bptr -> show();        // calls Base version

    cout << "\n\n bptr points to Derived\n";
    bptr = &D;
    bptr -> display();    // calls Base version
    bptr -> show();        // calls Derived version

    return 0;
}
```

Program 9.12
The output of Program 9.12 would be:

bptr points to Base
Display base
Show base

bptr points to Derived
Display base
Show derived

**note**

When `bptr` is made to point to the object `D`, the statement

```
bptr -> display();
```

calls only the function associated with the `Base` (i.e. `Base :: display( )`), whereas the statement

```
bptr -> show();
```

calls the `Derived` version of `show()`. This is because the function `display()` has not been made `virtual` in the `Base` class.

One important point to remember is that, we must access `virtual` functions through the use of a pointer declared as a pointer to the base class. Why can't we use the object name (with the dot operator) the same way as any other member function to call the virtual functions?. We can, but remember, run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

Let us take an example where `virtual` functions are implemented in practice. Consider a book shop which sells both books and video-tapes. We can create a class known as `media` that stores the title and price of a publication. We can then create two derived classes, one for storing the number of pages in a book and another for storing the playing time of a tape. Figure 9.2 shows the class hierarchy for the book shop.
The classes are implemented in Program 9.13. A function `display()` is used in all the classes to display the class contents. Notice that the function `display()` has been declared virtual in `media`, the base class.

In the `main` program we create a heterogeneous list of pointers of type `media` as shown below:

```cpp
gmedia *list[2] = { &book1, &tape1};
```

The base pointers `list[0]` and `list[1]` are initialized with the addresses of objects `book1` and `tape1` respectively.

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

using namespace std;

class media
{
    protected:
        char title[50];
        float price;
    public:
        media(char *s, float a)
        {
            strcpy(title, s);
            price = a;
        }
        virtual void display() { } // empty virtual function
};

class book: public media
{
    int pages;
    public:
        book(char *s, float a, int p): media(s, a)
        {
            pages = p;
        }
        void display();
};

(Contd)
```
class tape : public media
{
    float time;
public:
    tape(char * s, float a, float t): media(s, a)
    {
        time = t;
    }
    void display();
};

void book :: display()
{
    cout << "\n Title: " << title;
    cout << "\n Pages: " << pages;
    cout << "\n Price: " << price;
}

void tape :: display()
{
    cout << "\n Title: " << title;
    cout << "\n play time: " << time << "mins";
    cout << "\n price: " << price;
}

int main()
{
    char * title = new char[30];
    float price, time;
    int pages;

    // Book details
    cout << "\n ENTER BOOK DETAILS\n";
    cout << " Title: "; cin >> title;
    cout << " Price: "; cin >> price;
    cout << " Pages: "; cin >> pages;

    book book1(title, price, pages);

    // Tape details
    cout << "\n ENTER TAPE DETAILS\n";
    cout << " Title: "; cin >> title;
    cout << " Price: "; cin >> price;
    cout << " Play time (mins): "; cin >> time;

    (Contd)
taped tape1(title, price, time);

media* list[2];
list[0] = &book1;
list[1] = &tape1;

cout << "\n MEDIA DETAILS"

cout << "\n ......BOOK......"
list[0] -> display(); // display book details

cout << "\n ......TAPE......"
list[1] -> display(); // display tape details

result 0;

PROGRAM 9.13

The output of Program 9.13 would be:

ENTER BOOK DETAILS
Title: Programming_in_ANSI_C
Price: 88
Pages: 400

ENTER TAPE DETAILS
Title: Computing_Concepts
Price: 90
Play time (mins): 55

MEDIA DETAILS
......BOOK......
Title: Programming_in_ANSI_C
Pages: 400
Price: 88

......TAPE......
Title: Computing_Concepts
Play time: 55mins
Price: 90

Rules for Virtual Functions
When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements:
1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

### 9.7 Pure Virtual Functions

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder. For example, we have not defined any object of class media and therefore the function display() in the base class has been defined 'empty'. Such functions are called "do-nothing" functions.

A "do-nothing" function may be defined as follows:

```cpp
virtual void display() = 0;
```

Such functions are called *pure virtual* functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redefine it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called *abstract base classes*. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.
SUMMARY

 Polynomial simply means one name having multiple forms.

 There are two types of polymorphism, namely, compile time polymorphism and run
time polymorphism.

 Functions and operators overloading are examples of compile time polymorphism. The
overloaded member functions are selected for invoking by matching arguments, both
type and number. The compiler knows this information 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 or static binding or static linking. It means that
an object is bound to its function call at compile time.

 In run time polymorphism, an appropriate member function is selected while the program
is running. C++ supports run time polymorphism with the help of virtual functions. It
is called late or dynamic binding because the appropriate function is selected dynamically
at run time. Dynamic binding requires use of pointers to objects and is one of the
powerful features of C++.

 Object pointers are useful in creating objects at run time. It can be used to access the
public members of an object, along with an arrow operator.

 A this pointer refers to an object that currently invokes a member function. For example,
the function call a.show() will set the pointer 'this' to the address of the object 'a'.

 Pointers to objects of a base class type are compatible with pointers to objects of a
derived class. Therefore, we can use a single pointer variable to point to objects of base
class as well as derived classes.

 When a function is made virtual, C++ determines which function to use at run time
based on the type of object pointed to by the base pointer, rather than the type of the
pointer. By making the base pointer to point to different objects, we can execute different
versions of the virtual function.

 Run time polymorphism is achieved only when a virtual function is accessed through
a pointer to the base class. It cannot be achieved using object name along with the dot
operator to access virtual function.

 We can have virtual destructors but not virtual constructors.

 If a virtual function is defined in the base class, it need not be necessarily redefined in
the derived class. In such cases, the respective calls will invoke the base class function.

 A virtual function, equalled to zero is called a pure virtual function. It is a function
declared in a base class that has no definition relative to the base class. A class containing
such pure function is called an abstract class.
Key Terms

- Abstract base classes
- 'address of' operator
- argument object
- arrays of pointers
- arrow operator
- base address
- base object
- base pointer
- call back function
- class hierarchy
- compile time
- compile time polymorphism
- dereference operator
- Derived object
- do-nothing function
- dot operator
- dynamic binding
- early binding
- function overloading
- function pointer
- Implicit argument
- indirection operator
- invoking object
- late binding
- new operator
- Null pointers
- object pointer
- operator overloading
- placeholder
- pointers
- pointer arithmetic
- pointers to functions
- polymorphism
- pure virtual function
- run time
- run time polymorphism
- static binding
- static linking
- this pointer
- virtual constructors
- virtual destructors
- virtual function
- void pointers

Review Questions

9.1 What does polymorphism mean in C++ language?
9.2 How is polymorphism achieved at (a) compile time, and (b) run time?
9.3 Discuss the different ways by which we can access public member functions of an object.
9.4 Explain, with an example, how you would create space for an array of objects using pointers.
9.5 What does this pointer point to?
9.6 What are the applications of this pointer?
9.7 What is a virtual function?
9.8 Why do we need virtual functions?
9.9 When do we make a virtual function "pure"? What are the implications of making a function a pure virtual function?
9.10 State which of the following statements are TRUE or FALSE.
   (a) Virtual functions are used to create pointers to base classes.
   (b) Virtual functions allow us to use the same function call to invoke member functions of objects of different classes.
   (c) A pointer to a base class cannot be made to point to objects of derived class.
   (d) this pointer points to the object that is currently used to invoke a function.
   (e) this pointer can be used like any other pointer to access the members of the object it points to.
   (f) this pointer can be made to point to any object by assigning the address of
       the object.
   (g) Pure virtual functions force the programmer to redefine the virtual function
       inside the derived classes.

Debugging Exercises

9.1 Identify the error in the following program.

```cpp
#include <iostream.h>

class Info
{
    char *name;
    int number;
  public:
    void getInfo()
    {
        cout << "Info::getInfo ";
        getName();
    }

    void getName()
    {
        cout << "Info::getName ";
    }
};
```
class Name: public Info
{
    char *name;
public:
    void getName()
    {
        cout << "Name::getName ";
    }
};

void main()
{
    Info *p;
    Name n;
    p = n;
    p->getInfo();
}

/*
9.2 Identify the error in the following program.

#include <iostream.h>
class Person
{
    int age;
public:
    Person()
    {
    }
    Person(int age)
    {
        this.age = age;
    }
    Person& operator < (Person &p)
    {
        return age < p.age ? p: *this;
    }
    int getAge()
    {
        return age;
    }

Copyrighted material
9.3 Identify the error in the following program.

```cpp
#include "iostream.h"

class Human
{
public:
    Human()
    {
    }

    virtual ~Human()
    {
        cout << "Human::~Human";
    }
};

class Student: public Human
{
public:
    Student()
    {
    }
    ~Student()
    {
        cout << "Student::~Student";
    }
};
```
};

void main()
{
    Human *H = new Student();
    delete H;
}

9.4 Correct the errors in the following program.

class test
{
    private:
        int m;
    public:
        void getdata()
        {
            cout << "Enter number:";
            cin >> m;
        }
        void display()
        {
            cout << m;
        }
};

main()
{
    test T;
    T->getdata();
    T->display();

test *p;
p = new test;
p.getdata();
(*p).display();
}

9.5 Debug and run the following program. What will be the output?

#include <iostream.h>
class A
{
    protected:
int a, b;
public:
  A(int x = 0, int y)
  {
    a = x;
    b = y;
  }
  virtual void print();
};

class B: public A
{
  private:
    float p, q;
  public:
    B(int m, int n, float u, float v)
    {
      p = u;
      q = v;
    }
    B() {p = q = 0;}
    void input(float u, float v);
    virtual void print(float);
};
void A::print(void)
{
  cout << A values: a = "<< b ="n; 
}
void B::print(float)
{
  cout << B values: u = "<< v ="n; 
}
void B::input(float x, float y)
{
  p = x;
  q = y;
}
main()
{
  A a1(10, 20), *ptr;
  B b1;
  b1.input(7.5, 3.142);
  ptr = &a1;
  ptr->print();
  ptr = &b1;
  ptr->print();
}
Programming Exercises

9.1 Create a base class called shape. Use this class to store two double type values that could be used to compute the area of figures. Derive two specific classes called triangle and rectangle from the base shape. Add to the base class, a member function get_data() to initialize base class data members and another member function display_area() to compute and display the area of figures. Make display_area() as a virtual function and redefine this function in the derived classes to suit their requirements.

Using these three classes, design a program that will accept dimensions of a triangle or a rectangle interactively, and display the area.

Remember the two values given as input will be treated as lengths of two sides in the case of rectangles, and as base and height in the case of triangles, and used as follows:

\[
\text{Area of rectangle} = x \times y \\
\text{Area of triangle} = \frac{1}{2} \times x \times y
\]

9.2 Extend the above program to display the area of circles. This requires addition of a new derived class 'circle' that computes the area of a circle. Remember, for a circle we need only one value, its radius, but the get_data() function in the base class requires two values to be passed. (Hint: Make the second argument of get_data() function as a default one with zero value.)

9.3 Run the above program with the following modifications:

(a) Remove the definition of display_area() from one of the derived classes.

(b) In addition to the above change, declare the display_area() as virtual in the base class shape.

Comment on the output in each case.