
Chapter Table of Contents

Chapter 4.2

Virtual Functions

Aim.....	221
Instructional Objectives.....	221
Learning Outcomes.....	221
4.2.1 Introduction.....	222
4.2.2 Virtual Functions	223
(i) Normal Member Functions Accessed with Pointers	224
(ii) Virtual Member Function Accessed with Pointers.....	225
(iii) Late Binding.....	226
(iv) Abstract Classes and Pure Virtual Functions	226
(v) Virtual Base Classes.....	228
Self-assessment Questions.....	231
4.2.3 Friend Functions	232
(i) Purpose of Friend Function	232
(ii) Defining Friend Functions.....	232
(iii) Friend Classes	234
Self-assessment Questions.....	235
4.2.4 Static Functions	235
(i) Accessing Static Functions	236
(ii) Numbering the Objects	237
Self-assessment Questions.....	237
4.2.5 The “this” Pointer.....	238
(i) Returning Values using This Pointer.....	239
Self-assessment Questions.....	240
Summary	241
Terminal Questions.....	242
Answer Keys.....	243
Activity.....	244
Case Study	244
Bibliography.....	246
e-References	246
External Resources	246
Video Links	247



Aim

To equip the students with the skills to write programs using virtual functions in C++ programming



Instructional Objectives

After completing this chapter, you should be able to:

- Describe virtual functions with example
- Explain Abstract class and pure virtual functions
- Illustrate the purpose of Friend function and static function
- Describe virtual base classes and friend classes
- Explain this pointer



Learning Outcomes

At the end of this chapter, you are expected to:

- Elaborate the role of virtual function with programming example
- Differentiate normal and virtual member function accessed by pointers
- Explain properties of abstract class and pure virtual functions
- List the advantages of friend class and write a code to demonstrate friend function
- Demonstrate a C++ program on this pointer

4.2.1 Introduction

Suppose there are member functions with the same name in base class and derived class. The virtual functions give programmer capability to call member function of different class by a same function call depending upon different context. This feature in C++ programming is known as polymorphism which is one of the important features of OOP.

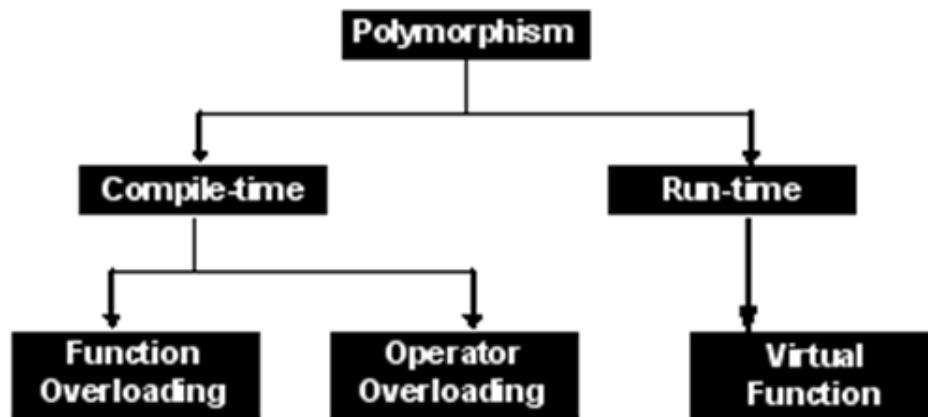


Figure 4.2.1: Polymorphism in C++

Figure 4.2.1 shows the different types of polymorphism in C++. The word polymorphism refers to expressing same function in multiple forms. *i.e.*, in object oriented programming, a method or object with same name will behave differently with different parameters.

There are two types of polymorphism:

1. Compile-time Polymorphism
2. Run-time Polymorphism

Compile-time polymorphism refers to a type of polymorphism where an object will behave differently with different parameters that are passed to it at the compile time of the program. It is also called as static polymorphism or static binding or overloading.

Run-time polymorphism refers to a type of polymorphism where an object will behave differently with different parameters that are passed to it at the run time of the program. It is also called as dynamic polymorphism or dynamic binding or overriding.

A base class and a derived class may have the same function. We write code to access that function using pointer of base class. Then, the function in the base class is executed, even if the object of derived class is referenced with that pointer variable.

4.2.2 Virtual Functions

Virtual function is a type of function C++ which is based on the concept of run-time polymorphism. It is present in a base class and its existence is then overridden in the derived class which then make the compiler to accept parameters at run time which is also called as late binding or dynamic n binding.

Virtual Keyword is used to make a member function of the base class Virtual.

The following program illustrates the concept of virtual functions.

```
1  #include <iostream>
2  using namespace std;
3  class A
4  {
5  public:
6  virtual void display()    /* Virtual function */
7  { cout<<"Content of base class.\n"; }
8  };
9
10 class B1 : public A
11 {
12 public:
13 void display()
14 { cout<<"Content of first derived class.\n"; }
15 };
16
17 class B2 : public A
18 {
19 public:
20 void display()
21 { cout<<"Content of second derived class.\n"; }
22 };
23 int main()
24 {
25     A *a;
26     B1 b1;
27     B2 b2;
28
29     /* a->display(); // we cannot use this code here because the function of base class is virtual. */
30
31     a = &b1;
32     a->display();    /* calls display() of class derived B1 */
33     a = &b2;
34     a->display();    /* calls display() of class derived B2 */
35     return 0;
36 }
```

Output:

```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Content of first derived class.
Content of second derived class.
sh-4.3$
```

(i) Normal Member Functions Accessed with Pointers

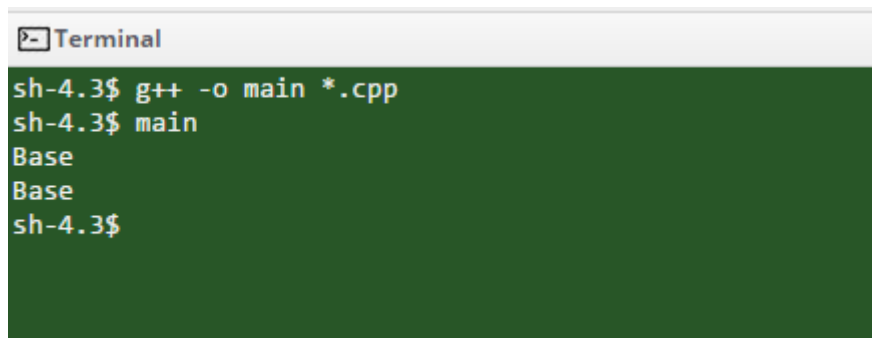
Similar to how we have pointers attached to our variables and functions in our programs, we can also have make pointers to access member functions in C++ programming.

The program given below illustrates how normal member functions are accessed with pointers

```
1  #include <iostream>
2  using namespace std;
3  class Base //base class
4  {
5  public:
6  void show() //normal function
7  { cout<< "Base\n"; }
8  };
9  class Derv1 : public Base //derived class 1
10 {
11 public:
12 void show()
13 { cout<< "Derv1\n"; }
14 };
15
16 class Derv2 : public Base //derived class 2
17 {
18 public:
19 void show()
20 { cout<< "Derv2\n"; }
21 };
22
23 int main()
24 {
25     Derv1 dv1;           //object of derived class 1
26     Derv2 dv2;           //object of derived class 2
27     Base* ptr;           //pointer to base class
28     ptr = &dv1;          //put address of dv1 in pointer
29     ptr->show();          //execute show()
30     ptr = &dv2;          //put address of dv2 in pointer
31     ptr->show();          //execute show()
32     return 0;
33 }
```

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

Output:



```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Base
Base
sh-4.3$
```

(ii) Virtual Member Function Accessed with Pointers

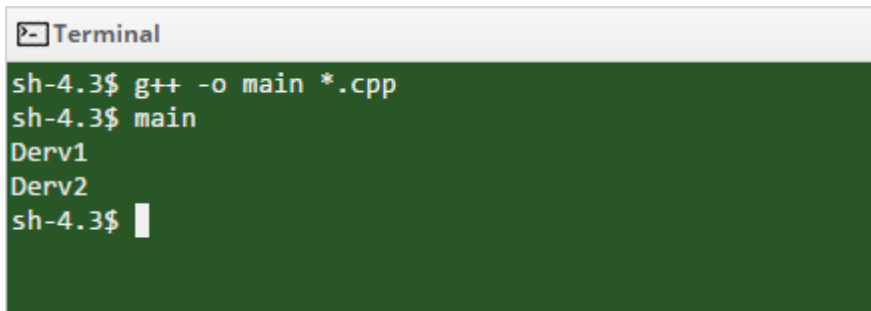
We can use pointers to access virtual member functions similar to how we have used pointers for normal member functions in the earlier sub-section 4.2.2.(i). With this feature, we can achieve run-time polymorphism that results into dynamic behaviour of our C++ programs.

Now, we will use the keyword virtual for the above program.

```
1  #include <iostream>
2  using namespace std;
3
4  class Base //base class
5  {
6  public:
7  virtual void show() //virtual function
8  { cout<< "Base\n"; }
9  };
10
11 class Derv1 : public Base           //derived class 1
12 {
13 public:
14 void show()
15 { cout<< "Derv1\n"; }
16 };
17
18 class Derv2 : public Base           //derived class 2
19 {
20 public:
21 void show()
22 { cout<< "Derv2\n"; }
23 };
24
25 int main()
26 {
```

```
27 Derv1 dv1;           //object of derived class 1
28 Derv2 dv2;           //object of derived class 2
29 Base* ptr;            //pointer to base class
30 ptr = &dv1;           //put address of dv1 in pointer
31 ptr->show();           //execute show()
32 ptr = &dv2;           //put address of dv2 in pointer
33 ptr->show();           //execute show()
34 return 0;
35 }
```

Output:



```
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Derv1
Derv2
sh-4.3$
```

As seen in the output, instead of executing the member functions of the base class as seen in earlier section 4.2.2 (i), it executes the member functions of the derived classes. This happens because we have declared the show() functions of base class as virtual which makes the program to behave in this manner and it then executes the member functions of the derived class.

(iii) Late Binding

In Late Binding function, call is resolved at runtime. Hence, now compiler determines the type of object at runtime and then binds the function call. Late Binding is a slow process. However, it provides flexibility to the code. Late Binding is also called Dynamic Binding or Runtime Binding.

For example, Virtual functions.

(iv) Abstract Classes and Pure Virtual Functions

Abstract classes are those classes which contains a pure virtual function. The sub-classes which are then created from this abstract classes gets its interface from these abstract classes.

Pure virtual functions are those functions which contains a keyword virtual before a function name which is then initialised to 0, to its declaration. They do not contain any definition.

It becomes mandatory for all the classes to provide definition to the pure virtual function that inherits an abstract class. If not provided, the same classes will also become abstract classes.

Also, pure virtual functions are always defined outside the class definition. If not declared outside, the compiler gives an error.

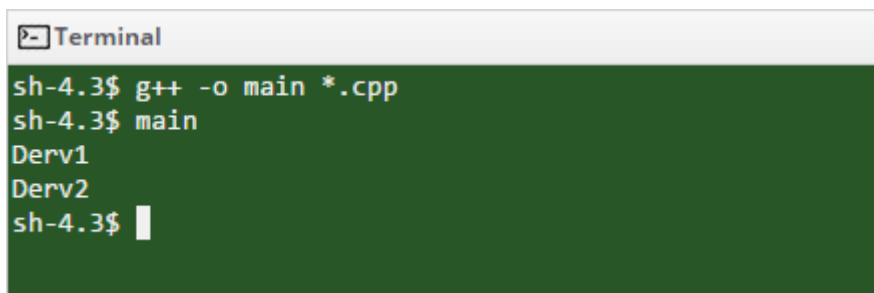
For example,

```
1  #include <iostream>
2  using namespace std;
3  class Base                               //base class
4  {
5  public:
6  virtual void show() = 0; //pure virtual function
7  };
8  class Derv1 : public Base                 //derived class 1
9  {
10 public:
11 void show()
12 { cout<< "Derv1\n"; }
13 };
14 class Derv2 : public Base                 //derived class 2
15 {
16 public:
17 void show()
18 { cout<< "Derv2\n"; }
19 };
20 int main()
21 {
22 // Base bad;                               //can't make object from abstract class
23 Base* arr[2];                             //array of pointers to base class
24 Derv1 dv1;                                //object of derived class 1
25 Derv2 dv2;                                //object of derived class 2
26 arr[0] = &dv1; //put address of dv1 in array
27 arr[1] = &dv2; //put address of dv2 in array
28 arr[0]->show(); //execute show() in both objects
29 arr[1]->show();
30 return 0;
31 }
```

Here the virtual function show() is declared as

```
virtual void show() = 0; // pure virtual function
```

Output:



```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Derv1
Derv2
sh-4.3$
```

The output of the above program is the same as seen in the previous example program in subsection (iii).

(v) Virtual Base Classes

Virtual base classes are those classes that are defined in C++ which helps us to resolve ambiguity when many paths do exist for a class belonging to a same base class. This will then avoid duplicates member sets of child class which was derived from a single base class. And by introducing virtual base classes, the above ambiguity is resolved.

Consider the following program 1:

```
1  class A
2  {
3      public:
4          int i;
5  };
6
7  class B : virtual public A
8  {
9      public:
10         int j;
11 };
12
13 class C: virtual public A
14 {
15     public:
16         int k;
17 };
18
19 class D: public B, public C
20 {
21     public:
22         int sum;
23 };
24
25 int main()
26 {
27     D ob;
28     ob.i = 10; //unambiguous since only one copy of i is inherited.
29     ob.j = 20;
30     ob.k = 30;
31     ob.sum = ob.i + ob.j + ob.k;
32     cout << "Value of i is : " << ob.i << "\n";
33     cout << "Value of j is : " << ob.j << "\n";
34     cout << "Value of k is : " << ob.k << "\n";
35     cout << "Sum is : " << ob.sum << "\n";
36
37     return 0;
38 }
```

Output:

Output:

```
1 Value of i is : 10
2 Value of j is : 20
3 Value of k is : 30
4 Sum is : 60
```

Program 2:

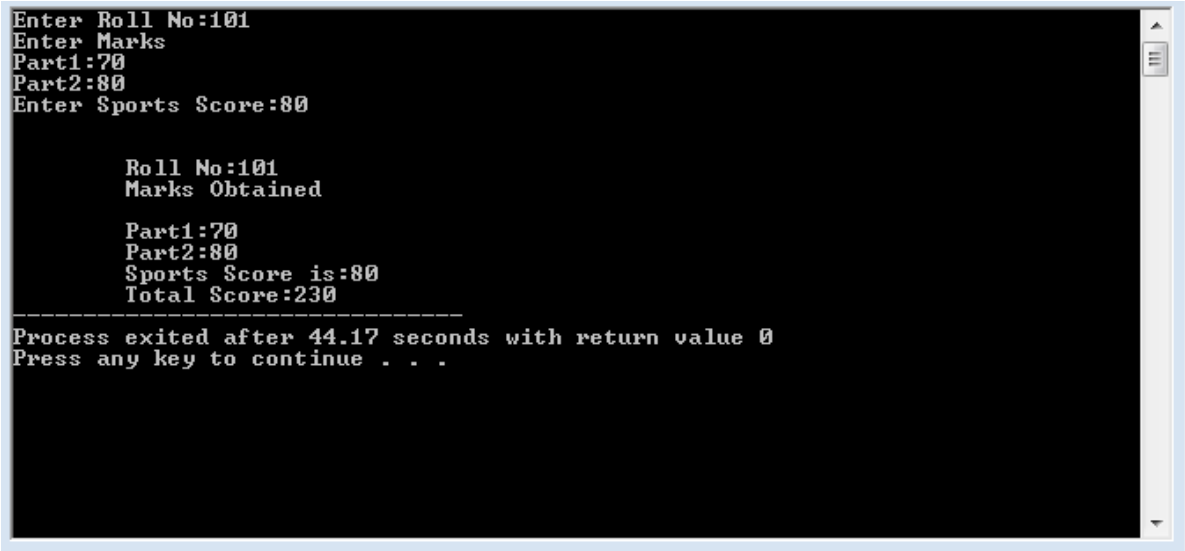
```
1  #include<iostream>
2
3  class student
4  {
5      int rno;
6  public:
7      void getnumber()
8      {
9          cout<<"Enter Roll No:";
10         cin>>rno;
11     }
12     void putnumber()
13     {
14         cout<<"\n\n\tRoll No:"<<rno<<"\n";
15     }
16 };
17
18 class test:virtual public student
19 {
20
21 public:
22     int part1,part2;
23     void getmarks()
24     {
25         cout<<"Enter Marks\n";
26         cout<<"Part1:";
27         cin>>part1;
28         cout<<"Part2:";
29         cin>>part2;
30     }
31     void putmarks()
32     {
33         cout<<"\tMarks Obtained\n";
34         cout<<"\n\tPart1:"<<part1;
35         cout<<"\n\tPart2:"<<part2;
36     }
37 };
38
39 class sports:public virtual student
40 {
41
42 public:
43     int score;
44     void getscore()
45     {
46         cout<<"Enter Sports Score:";
47         cin>>score;
48     }
```

```

49     void putscore()
50     {
51         cout<<"\n\tSports Score is:"<<score;
52     }
53 };
54
55 class result:public test,public sports
56 {
57     int total;
58 public:
59     void display()
60     {
61         total=part1+part2+score;
62         putnumber();
63         putmarks();
64         putscore();
65         cout<<"\n\tTotal Score:"<<total;
66     }
67 };
68
69 int main()
70 {
71     result obj;
72     obj.getnumber();
73     obj.getmarks();
74     obj.getscore();
75     obj.display();
76 }

```

Output:



```

Enter Roll No:101
Enter Marks
Part1:70
Part2:80
Enter Sports Score:80

    Roll No:101
    Marks Obtained

    Part1:70
    Part2:80
    Sports Score is:80
    Total Score:230
-----
Process exited after 44.17 seconds with return value 0
Press any key to continue . . .

```

Returning values using “this” pointer

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,

```
return *this;
```

The statement inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare 2 or more objects inside a member function and return the invoking object as a result of multiple inheritances. So, they should be used with caution.



Self-assessment Questions

- 1) Run-time polymorphism can be achieved with _____.
 - a) Virtual base class
 - b) Virtual function
 - c) Member function
 - d) Pure virtual function
- 2) When a virtual function is redefined by the derived class, it is called _____.
- 3) If a class contains pure virtual function then it is termed as abstract class.
 - a) True
 - b) False

4.2.3 Friend Functions

There could be a situation where we would like two classes to share a particular function. **For example**, consider a case where two classes, manager and scientist have been defined. We would like to use a function `incometax()` to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

(i) Purpose of Friend Function

The concepts of encapsulation and data hiding dictate that non-member functions should not be able to access an object's private or protected data. The policy is, if we are not a member, we cannot get in. However, there are situations where such rigid discrimination leads to considerable inconvenience. Imagine that we want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments and operate on their private data. In this situation, there is nothing like a friend function.

(ii) Defining Friend Functions

A friend function of a class is defined outside that class's scope, but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or a member function, or a class or class template in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword `friend` as follows:

```
class Box
{
    double width;
    public:
    double length;
    friend void printWidth( Box box );
    void setWidth( double wid );
};
```

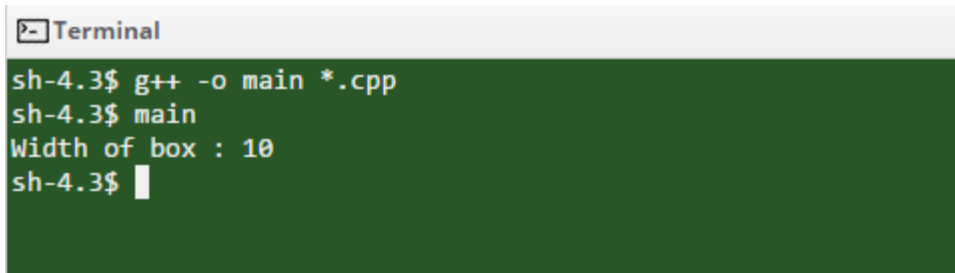
To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne:

```
friend class ClassTwo;
```

Consider the following program:

```
1  #include <iostream>
2  using namespace std;
3  class Box
4  {
5  double width;
6  public:
7  friend void printWidth( Box box );
8  void setWidth( double wid );
9  };
10 // Member function definition
11 void Box::setWidth( double wid )
12 {
13 width = wid;
14 }
15
16 // Note: printWidth() is not a member function of any class.
17 void printWidth( Box box )
18 {
19     /* Because printWidth() is a friend of Box, it can
20     directly access any member of this class */
21     cout<< "Width of box : " <<box.width<<endl;
22 }
23
24 // Main function for the program
25 int main( )
26 {
27     Box box;
28
29     // set box width without member function
30     box.setWidth(10.0);
31
32     // Use friend function to print the width.
33     printWidth( box );
34
35     return 0;
36 }
```

Output:



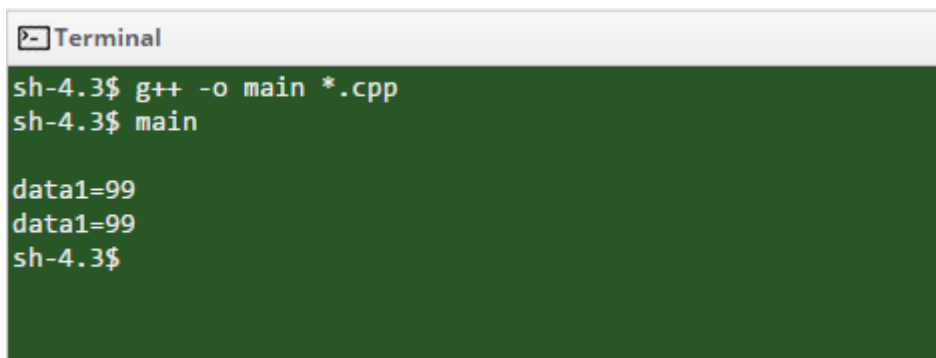
```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Width of box : 10
sh-4.3$
```

(iii) Friend Classes

Friend classes are those classes declared with a keyword friend which helps non-member to access private and protected members of a class if it is declared as a friend of that class. Since, we know that private and protected members of a class cannot be accessed from outside the class which is then made possible by friend classes.

```
1  #include <iostream>
2  using namespace std;
3  class alpha
4  {
5  private:
6  int data1;
7  public:
8  alpha() : data1(99) { }           //constructor
9  friend class beta;               //beta is a friend class
10 };
11 class beta
12 {                                //all member functions can access private alpha data
13 public:
14 void func1(alpha a) { cout << "\ndata1=" << a.data1; }
15 void func2(alpha a) { cout << "\ndata1=" << a.data1; }
16 };
17 int main()
18 {
19 alpha a;
20 beta b;
21 b.func1(a);
22 b.func2(a);
23 cout<< endl;
24 return 0;
25 }
```

Output:



```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main

data1=99
data1=99
sh-4.3$
```

The above program illustrates the use of friend classes which helps to access private and protected members of a class from outside the same classes in which they are declared.

With this, the member functions of beta class are able to access the private data belonging to alpha class.



Self-assessment Questions

- 4) A function can be declared as friend maximum only in two classes.
 - a) True
 - b) False

- 5) Which of the following is false about friend function?
 - a) It can be called / invoked with class object
 - b) It has objects as arguments
 - c) It can have built-in types as arguments
 - d) It does not have this pointer as an argument

- 6) Member functions of one class be friend function of _____ class.

4.2.4 Static Functions

A static data member is not duplicated for each object; rather a single data item is shared by all objects of a class. The STATIC example showed a class that kept track of how many objects of it there were. Let us extend this concept by showing how functions as well as data may be static. Besides showing static functions, our example will model a class that provides an ID number for each of its objects. This allows us to query an object to find out which object it is—a capability *i.e.*, sometimes useful in debugging a program, among other situations. The below program illustrates the operation of destructors.

```
1  #include <iostream>
2  using namespace std;
3  class gamma
4  {
5  private:
6  static int total;           //total objects of this class (declaration only)
7  int id;                    //ID number of this object
8  public:
9  gamma()                   //no-argument constructor
10 {
11     total++;               //add another object
12     id = total;            //id equals current total
13 }
14 ~gamma()                  //destructor
15 {
16     total--;
17     cout<< "Destroying ID number " << id << endl;
18 }
19 static void showtotal()    //static function
20 {
21     cout<< "Total is " << total << endl;
22 }
```

```

23 void showid()           //non-static function
24 {
25     cout<< "ID number is " << id << endl;
26 }
27 };
28 int gamma::total = 0;    //definition of total
29
30 int main()
31 {
32     gamma g1;
33     gamma::showtotal();
34     gamma g2, g3;
35     gamma::showtotal();
36     g1.showid();
37     g2.showid();
38     g3.showid();
39     cout<< "-----end of program-----\n";
40     return 0;
41 }

```

Output:

```

Total is 1
Total is 3
ID number is 1
ID number is 2
ID number is 3
-----end of program-----
Destroying ID number 3
Destroying ID number 2
Destroying ID number 1
-----
Process exited after 0.8162 seconds with return value 0
Press any key to continue . . .

```

(i) Accessing Static Functions

When a function member is declared as static, you make it autonomous of a specific object of the class. A static member function can be called regardless of the fact that no objects of the class exist and the static functions are utilised only by its class name and the scope resolution operator ::.

A dummy object is created which is used to call a member function.

```

gamma dummyObj;           // make an object so we can call function
dummyObj.showtotal();      // call function

```

With scope resolution operator, we then use the name of the class as seen in the next statement.

```

gamma::showtotal();        // more reasonable

```

If showtotal() is not defined as per the above statement, it will not work. Since it then becomes a normal member function.

To access showtotal() using only the class name, we must declare it to be a static member function. static void showtotal()

Now the function can be accessed using only the class name.

(ii) Numbering the Objects

Another function is used in gamma() which prints the ID numbers of the individual members of the class. With numbering of objects, each object thus have a unique number.

The showid () function is used to print the ID of the numbered objects.

It is then called three times in the main function as given below:

```
g1.showid();  
  
g2.showid();  
  
g3.showid();
```

As shown in the above output, we can say that each object have a unique number. *i.e.*, object g1 is numbered as 1, object g2 is numbered as 2 and object g3 is numbered as 3.



Self-assessment Questions

- 7) When a data member is declared _____, there is only one such data value for the entire class, no matter how many objects of the class are created.
- 8) A static data member is not duplicated for each object, rather a single data item is shared by all objects of a class.
 - a) True
 - b) False
- 9) The correct way of defining a Static member function.
 - a) Static void showtotal()
 - b) Void static showtotal()
 - c) Static void showtotal
 - d) Static void showtotal {}

4.2.5 The “this” Pointer

This pointer is a type of pointer which is used to invoke a member function for an object. When a member function is called, this pointer is automatically passed when it is called. With this we can say that, this pointer acts as an implicit argument to all the member functions.

This pointer can only be used for member functions. With this, we can conclude that friends functions cannot have this pointer.

Program:

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Box
6  {
7  public:
8      // Constructor definition
9      Box(double l=2.0, double b=2.0, double h=2.0)
10     {
11         cout << "Constructor called." << endl;
12         length = l;
13         breadth = b;
14         height = h;
15     }
16     double Volume()
17     {
18         return length * breadth * height;
19     }
20     int compare(Box box)
21     {
22         return this->Volume() > box.Volume();
23     }
24 private:
25     double length;    // Length of a box
26     double breadth;   // Breadth of a box
27     double height;    // Height of a box
28 };
29
30 int main(void)
31 {
32     Box Box1(3.3, 1.2, 1.5);    // Declare box1
33     Box Box2(8.5, 6.0, 2.0);    // Declare box2
34
35     if(Box1.compare(Box2))
36     {
37         cout << "Box2 is smaller than Box1" << endl;
38     }
39     else
40     {
41         cout << "Box2 is equal to or larger than Box1" << endl;
42     }
43     return 0;
44 }
```

Output:

```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
sh-4.3$
```

(i) Returning Values using This Pointer

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,

```
return *this;
```

The statement inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare 2 or more objects inside a member function and return the invoking object as a result.



Did you Know?

Object-oriented languages typically manage memory allocation and de-allocation automatically when objects are created and destroyed.



11) “This” pointer can be used to return the value from a member function.

- 12) The pointer _____ acts as an implicit argument to all the member functions.

- a) This b) Variable
c) Function d) Constant



Summary

- Polymorphism simply means one name having multiple forms.
- Functions and operators overloading are examples of compile time polymorphism.
- When a function is made virtual, C++ determines which function to use at runtime. It is based on the type of object pointed to by the base pointer, rather than the type of pointer.
- Runtime polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.
- We can have virtual destructors but not virtual constructors
- A “this” pointer refers to an object that currently invokes a member function.



Terminal Questions

1. Describe virtual functions with example.
2. Explain Abstract class and pure virtual functions.
3. Illustrate the purpose of Friend function and static function.
4. Describe virtual base classes and friend classes.
5. Explain this pointer.



Answer Keys

Self-assessment Questions	
Question No.	Answer
1	b
2	Overriding
3	a
4	a
5	a
6	Another
7	Static
8	a
9	a
10	Object
11	a
12	This



Activity

Activity Type: Online

Duration: 45 Minutes

Description:

Develop and execute a program to calculate bonus of the employees. The class master derives the information from both admin and account classes which intern derives information from class person. Create base and all derived classes having same member functions called get data, display data and bonus. Create a base class pointer that capable of accessing data of any class and calculates bonus of the specified employee. (**Hint:** Use virtual functions).

Case Study

Problem Statement: Let us look at the illustration of the VIRTUAL method.

```
class Base
{
    public:
        void f();
        virtual void vf();
};

class Derived : public Base
{
    public:
        void f();
        voidvf();
};

#include <iostream>
using namespace std;

void Base::f()
{
    cout<< "Base f()" << endl;
}
void Base::vf()
{
    cout<< "Base vf()" << endl;
}
void Derived::f()
{
    cout<< "Derived f()" << endl;
}
void Derived::vf()
{
```

```
        cout<< "Derived vf()" << endl;
    }
int main()
{
    Base b1;
    Derived d1;
    b1.f();
    b1.vf();
    d1.f();
    d1.vf();
    Derived d2;          // Derived object
    Base* bp = &d2; // Base pointer to Derived object
    bp->f();    // Base f()
    bp->vf();   // which vf()?
    return 0;
}
```

Questions:

1. What is the output of the program?
2. What happens If vf() is not declared as virtual in the base class?
3. What happens If vf() is declared as virtual in the base class?

Bibliography



e-References

- programiz.com,(2016). *C++ Virtual Function*. Retrieved 7 April, 2016. from, <http://www.programiz.com/cpp-programming/virtual-functions>
- go4expert.com,(2016). *Virtual, Static and Friend Functions in C++*.Retrieved 7 April, 2016. from, <http://www.go4expert.com/articles/virtual-static-friend-functions-cpp-t29967/>
- cpp.thiyagaraaj.com,(2016). *Simple Program for Virtual Functions Using C++ Programming*. Retrieved 7 April, 2016. from, <http://www.cpp.thiyagaraaj.com/c-programs/simple-program-for-virtual-functions-using-c-programming>



External Resources

- Balaguruswamy, E. (2008). *Object Oriented Programming with C++*. Tata McGraw Hill Publications.
- Lippman. (2006). *C++ Primer (3rd ed.)*. Pearson Education.
- Robert, L. (2006). *Object Oriented Programming in C++ (3rd ed.)*. Galgotia Publications References.
- Schildt, H., & Kanetkar, Y. (2010). *C++ completer (1st ed.)*. Tata McGraw Hill.
- Strousstrup. (2005). *The C++ Programming Language (3rd ed.)*. Pearson Publications



Video Links

Topic	Link
Virtual functions	https://www.youtube.com/watch?v=UaISAX83Yk8
Friend functions	https://www.youtube.com/watch?v=_urmtRk9zQA
Static functions	https://www.youtube.com/watch?v=Ut8LLK__B7E
Abstract Classes	https://www.youtube.com/watch?v=4YhgZ1Uun7Q



Notes:

