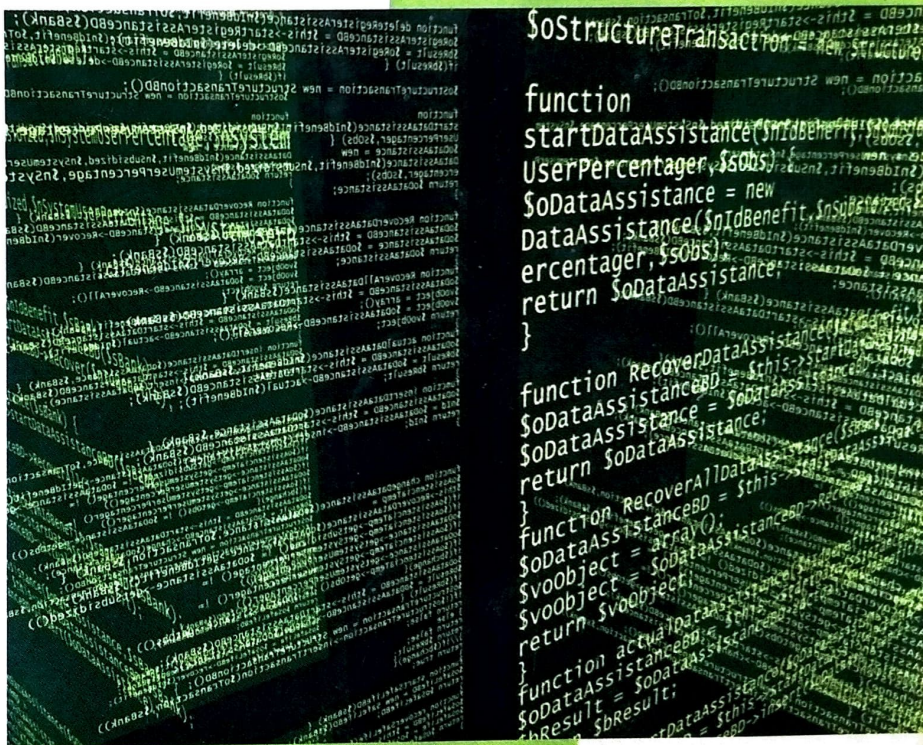
PEARSON

Object Oriented Programming with C++



Subhash K U

More on Classes and Objects

3

3.1. INTRODUCTION

In the previous chapter we learnt about differences between structures in C and structures in C++. We also learnt about the basic difference between structures in C++ and Classes in C++. We discussed the concept of Classes and objects and the relationship between them. We found that, the data members of the class are accessible and can be modified only by the member-functions (and also friend functions which is discussed in detail in the later sections) of the class. In this chapter we shall learn more about member functions and their types, data members and their types, friend functions and friend classes and namespaces. The study of these concepts helps the programmer learn to do programming better and also help him/her equip himself/herself with the basic but highly necessary tools of C++ programming language. Let us start discussing with the following topic.

3.2. MEMBER FUNCTIONS AND DATA MEMBERS

Member functions and data members of a class can be made to work in several ways as the need for it arises. It will be nice to put the member functions and data members into various categoris and learn them one by one.

Types of Member functions:

- Nested Member functions
- Overloaded member functions
- Constant member functions
- Member functions with default arguments
- Inline member functions
- Static Member functions

Types of Data Members:

- Constant data members
- Mutable data members
- Static data members

Let us discuss them one by one in detail.

Nested Member Functions :

A member function call can be embedded within another member function. That is, in other words, a member function can be called from within another member function definition. Such member functions are called nested member functions. The following program 3.1 helps in understanding this concept much better.

Program 3.1

```
/*3.1.cpp */
#include <iostream>
using namespace std;
class simple
{
      int x;
   public:
      void set_data( int );
      int get_data( void );
      void message( char * );
};
void simple :: set_data( int a )
{
   x = a;
   message( "Setting" );          //Nested Member function
}

int  simple :: get_data( void )
{
   message( "Getting" );         // Nested Member function
   return x;
}
void simple :: message( char *s )
{
   cout << s << endl;
}
int main( )
{
   simple e;
   e.set_data( 5 );
   cout << e.get_data( ) << endl;
   return 0;
}
```

Output:

```
setting
set to 5
```

From the above program, we can observe that, the member functions 'set_data( )' and 'get_data( )' of class 'simple' are called from the 'main( )' function (a non-member function of the class simple) using the object of the same class (i.e., class 'simple') through object to member access operator(.). This is because, the 'main( )' being the non-member function of the class 'simple', it cannot access the member function of class 'simple' directly without referring it with the help of the class object. But in the function definitions of 'set_data( )' and 'get_data( )', a call is being made to the function 'message( )', which is also a member function of class 'simple' without referring it with an object. This is because, when the function 'set_data( )' and 'get_data( )' are called in 'main( )' with reference to an existing object, the pointer to the object is passed implicitly as a formal argument to the called function by the compiler. This address is stored in the 'this' pointer (as discussed earlier in 2nd chapter). Therefore, in the function definition of 'set_data( )' and 'get_data( )', the member function 'message( )' is referred using the 'this' pointer. And because, the 'message( )' is also a member function of the class 'simple', the function is easily dereferenced using 'this' pointer.

During compilation of the above program, once the compiler ascertains that no access is made to the private members of the class 'simple' by any non-member functions, the original C++ source code is converted to the code as shown below:

```
#include <iostream>
using namespace std;
class simple
{
    int x;
    void set_data( simple * const, int );
    int get_data( simple * const );
    void message( char * );
};
void simple :: set_data( simple * const this, int a )
{
  this->x = a;
  this->message( "Setting" );
}
int simple :: get_data( simple * const this)
```

```
    this->message( "Getting" );
    return this->x;
}
void simple :: message(  char *s )
{
    cout << s << endl;
}
int main( )
{
    simple e;
    set_data(&e, 5 );
    cout << get_data( &e) << endl;
    return 0;
}
```

Hence, while calling a member function within another member function, it is not referred with an object of a class. In the above program, the function 'message( )' is declared in the public section. The above discussion is also true when the function 'message( )', is placed in the private section of a class. The point to be noted here is that, when the member functions are defined outside the class, scope resolution operator has to be used as shown above.

Overloaded Member Functions:

In the 1st chapter, we had learnt how to write overloaded functions. But we had not dealt with classes and objects. The same technique is used to create overloaded member functions of a class also. Two or more functions of a class or a different class which share a common name but have different parameter list are called overloaded member functions.

Types of overloaded functions are as follows:

a. Overloaded member functions within a single class:

Program 3.2

```
/*3.2.cpp*/
class attitude
{
    public:
        void display( void );
        void display( int );
};
void attitude :: display( void )
{
    cout << "Attitude" << endl;
```

```
}
void attitude :: display( int d )
{
cout << "positive A + T+ T + I +T + U + D + E =  " << d << "%
        success\n";
}
int main( )
{
   attitude a1;
   a1.display(void);
   a1.display( 100 );
   return 0;
}
```

Output:

```
Attitude
positive A + T+ T + I +T + U + D + E = 100% success
```

In the above example, 'display( )' is an overloaded member function of the class 'attitude'. As evident from the above program, the function 'display( )' is called once with a 'void' parameter, and next time with an 'int' parameter. As can be observed from the above program 3.2 , first time, the function call

```
   a1.display( void );
```

will be mapped to the following function definition,

```
void attitude :: display( void )
{
   cout << "Attitude" << endl;
}
```

Second time, the function call

```
   a1.display( 100 );
```

will be mapped to the following function definition,

```
void attitude :: display( int d )
{
   cout << "positive A + T+ T + I +T + U + D + E =  " << d << "%
           success\n";
}
```

It can be analyzed from the above program 3.2 that, in case of over loaded member functions the compiler decides which function to call after thoroughly examining the number and type of arguments matching the function call and function definition. Hence the output.

It is suggested for the reader to go through the 1st chapter on the topic 'Concept of function Overloading' and compare the topic with the present topic for better understanding. Hence overloaded member functions can have same function name but should differ in their number or type of parameter list.

b. Overloaded member functions of two different classes :

Program 3.3

```
/*3.3.cpp*/
#include <iostream>
using namespace std;
class attitude1
{
   public:
      void display( double );
};
class attitude2
{
   public:
      void display( double );
}
void attitude1 :: display( double d)
{
   cout << "Negative attitude is the key for " << d << "% failure"
        << endl;
}
void attitude2 :: display( double d )
{
   cout << "Positive attitude is the key for " << d << "% success"
        << endl;
}
int main( )
{
   attitude1  a1;
   attitude2  a2;
   double x;
   cout << "Enter the number" << endl;
   cin >> x ;
   a1.display( x );
   a2.display( x );
   return 0;
}
```

Output:

```
Enter the number
100.00
Negative attitude is the key for 100.00% failure
Positive attitude is the key for 100.00% success
```

In the above program, two different classes i.e. 'class attitude1' and 'class attitude2', both have 'display( )' as their member functions with having parameter list of the same data type, i.e. double. The functions in both the classes have the same name and signature. Signature refers to the function name and the argument/parameter list of a function. Function signature does not include the return type of the function. And this is possible, because of the concept of function overloading in C++. Here 'display( )' is the overloaded member function in two different classes. Though it seems the signature of 'display( )' is same, it is not true. With the knowledge of 'this' pointer as learnt from the 2nd chapter, we know that, during compilation, the address of the object 'a1' (object of 'class attitude1') is passed to the member function 'display( )' of the class 'attitude1' . That is, in main, 'a1.display(x)' becomes 'display(&a1, x)'. In the same way, the address of the object 'a2' (object of 'class attitude2') is passed to the member function display 'display( )' of the class 'attitude2'.That is, in 'main( )', 'a2.display(x);' becomes 'display (&a2, x);'. With this we can easily say that, though the 'display( )' function looks like having the same signature, during compilation the signature changes. With this signature change, function overloading takes place.

Member functions with default arguments:

In C++, the member functions of a class assigns default values for some or all of the formal arguments which does not have a corresponding matching actual arguments in the function calls and those default values are known as default arguments to member functions. An example for this is given below in the program 3.4.

Program 3.4

```
/*3.4.cpp*/
#include <iostream>
using namespace std;
class addition
{
   public:
      void add( int , int = 2 );
};
void addition :: add( int a, int  b )
{
```

```
    return a + b;
}
int main( )
{
    addition a1;
    a1.add( 5, 6 );
    a1.add( 5 );
}
```

Output:

```
11
7
```

In the above program, in main function, the member function 'add()' of class 'addition' is called twice, once with two parameters, and second time with only one parameter. First time, the values '5' and '6' are passed as formal arguments to the function definition. Here the default value '2' in the function prototype is ignored, because the value '6' is being assigned to the formal arugment 'b' of the member function 'add()' and hence the result produced is '11'(addition of 5 and 6). Second time, only one parameter is passed to the member function definition. Here the default value '2' specified in the function prototype, is used for the second formal argument 'b' and hence the result produced is 7(addition of 5 and 2). So from the program 3.4, it is clear that, when sufficient argument values are not available for the formal parameters of function definition, the default arguments present in the function prototype is taken as the default value for those parameters. It is to be noted that, the default values for the arguments are always assigned in right to left fashion in the function prototype. Refer chapter 1 for more details on default arguments.

Inline member functions :

An inline member function is a function that is expanded inline with the main program at the point of invocation when it is invoked. This is something very similar to using a macro. To cause a compiler to replace the member function call with its corresponding function code rather than calling the function definition(i.e, to make a function definiton inline with the function call), the function definition has to be preceded with the keyword 'inline'.

There are two ways by which member functions can be made inline. They are:

- *Defining the member function outside the class and prototyping the member function within the class. Here the function definition should precede by 'inline' keyword.*

Program 3.5

```
/*3.5.cpp*/
#include <iostream>
using namespace std;
class sample
{
   private:
      int x, y;
   public:
      void initialize( int, int );
      void display( );
};
inline void sample :: initialize( int i, int j )
{
   x = i;
   y = j;
}
inline void sample :: display( )
{
   cout << "x = " << x << "y = " << y << endl;
}
int main( )
{
   sample s ;
   s.initialize( 20, 30 ) ;
   s.display( ) ;
   return 0;
}
```

Output :

```
x = 20
y = 30
```

- *Automatic inline function :*

Here the member functions are defined within the class making the function automatically inline. Here, it is not necessary to precede the function definition with an inline keyword.

Program 3.6

```
/*3.6.cpp*/
#include <iostream>
using namespace std;
class sample
```

```
{
  private:
     int x;
     int y;
  public :
     void initialize( int i, int j )//automatic inline function
     {
        x = i;
        y = j;
     }
     void display( )//automatic inline function
     {
        cout << "x = " << x << "y = " << y << endl;
     }
} ;
int main( )
{
   sample s ;
   s.initialize( 20, 30 ) ;
   s.display( ) ;
   return 0 ;
}
```

Output :

```
x = 20
y = 30
```

Inline functions help reducing the overhead involved in calling the function definition, like pushing arguments on to the stack, saving the various registers, etc. But using inline functions for very large functions at many places in the source code increases the program size, which is undesirable. So inline functions should be used for small functions which are used rarely, and should be used only when required. Note that, inline functions should be always placed, before it is called. Refer 1st chapter for more details on inline functions.

Constant member functions:

Constant member functions are those functions which are denied permission to change the values of the data members of their class. To make a member function constant, the qualifier "const" is appended to the function prototype and also to the function definition header. The syntax is as follows.

i. For function declaration or function prototype within a class.

```
<data_type> <function_name>( ) const;
```

where,

```
data_type          → return type of the function
function_name      → name of the function
const              → keyword to make a function constant
```

Example:

```
int get_data( ) const;
```

ii. For function definition within the class declaration.

```
<data_type><function_name>( ) const
{
   // function definition
}
```

where,

```
data_type          → return type of the function
function_name      → name of the function
const              → keyword to make a function constant
```

Example:

```
int get_data( ) const //constant member function
{
   //function definition
}
```

iii. For function definition outside the class.

```
<data_type> <class_name> :: <function_name>( ) const //constant member
function
{
   //function definition
}
```

where,

`data_type`	→	return type of the function
`class_name`	→	name of the class to which the function is a member
`function_name`	→	member function of the class class_name.
`const`	→	keyword to make a function constant

Example:

```
int sample :: get_data( ) const //constant member function definition
                                //outside class
```

```
{
      //Function definition
}
```

If an attempt is made by the constant member functions to alter the values of data members of the class, the compiler generates an error message. The memory occupied by the invoking object for a constant member function is "read-only memory". During compilation, the object's address is passed as a "constant pointer to a constant", that is,

```
const <object_name> * const this
```

to the function definition. The above statement says that, the value pointed to by the pointer as well as the pointer itself cannot be changed. But recall how the object's address was being passed while invoking non-constant member function. It was just passed as a constant pointer, that is as,

```
<object_name> * const this
```

This is the difference between normal member function and constant member function.

Let us examine few examples to understand the concept much better.

Program 3.7

```
/*3.7.cpp*/

/* Not a constant member function. */
#include <iostream>
using namespace std;

class sample
{
      int x;
   public:
      void set_data( int c )
      {
         x = c;
      }
      int get_data( )
      {
         x++;                    // No error while modifying the data
                                 // member
         return x;
      }
};
int main( )
```

```
{
   sample b1;
   b1.set_data( 5 );
   cout << b1.get_data( ) << endl;
   return 0;
}
```

Output :

```
6
```

In the above program 3.7 it can be observed that, when attempt is made to change the value of the data member 'x' of the 'class sample' by the member function 'get_data( )', no error message is generated. The program is compiled successfully. This is because, the member function 'get_data( )' is not a constant member function. That means, during compilation, the address of the object 'b1' is passed as a "constant pointer" to the 'get_data( )' member function and not as "constant pointer to a constant". During compilation, the statement,

```
b1.get_data( );
```

becomes,

```
   get_data( &b1 );
```

and the function definition,

```
int get_data( )
{
   x++;
   return x;
}
```

becomes,

```
int get_data( sample *const this )
{
   (this->x)++;
   return (this->x);
}
```

Let us try the same program with a slight change. That is, let us declare the member function 'get_data( )' as constant. Now let us see what happens when the program 3.8 is compiled.

Program 3.8

```
/*3.8.cpp*/
#include <isotream>
using namespace std;
class sample
```

```
{
      int x;
   public:
      void set_data( int c )
      {
         x = c;
      }
      int get_data( ) const //constant member function
      {
         x++;// Error while attempting to modify the data memb
         return x;
      }
}
int main( )
{
   sample b1;
   b1.set_data( 5 );
   cout << b1.get_data( ) << endl;
   return 0;
}
```

Now it can be observed that, an error will be generated if the member fu 'get_data( )' tries to modify the value the data member 'x' of the class sample. T because the member function 'get_data( )' is trying to do a write operation on a 're memory area' allocated for the object 'b1' while invoking the constant member fu which is not permissible. In other words, during compilation, the address of the 'b1' is passed as a "constant pointer to a constant" to the 'get_data( )' member fu During compilation, the statement,

```
b1.get_data( );
```

becomes,

```
get_data( &b1 );
```

and the function definition,

```
int get_data( )
{
   x++;
   return x;
}
```

becomes,

```
int get_data( const sample *const this )
{
   (this->x)++;
   return (this->x);
}
```

Let us try another program, whose output is quite self-explainable. The only difference with the previous program and the next one (program 3.9) is that, the function is defined outside the class using scope resolution operator.

Program 3.9

```
/*3.9.cpp */
#include <iostream>
using namespace std;
class sample
{
      int x;
   public:
      void set_data( int );
      int get_data( ) const; //Prototyping constant member function
}
void sample :: set_data( int c )
{
   x = c;
}
int sample :: get_data( ) const //constant member function definition
                                //outside class
{
   x++; //Error while attempting to modify the data member's value
   return x;
}
int main( )
{
   sample b1;
   b1.set_data( 5 );
   cout << b1.get_data( ) << endl;
   return 0;
}
```

Now, before discussing the static member function, let us first understand what are ;

a. Constant data members
b. Mutable data members
c. Static data members

- **Constant data members:**

The data members whose value cannot be changed throughout the execution of the program is known as constant data members. The constant data members are declared by preceding the qualifier const as shown below.

```
const int x = 6;
```

Any attempt to change the value of x will flash an error. The memory allocated for constant data member is read only memory. Let us write a simple program to understand constant data members.

Program 3.10

```
/*3.10.cpp*/
#incude <iostream>
using namespace std;
int main( )
{
   const int x = 6;
   x++ ;              // Error
   cout << x << endl;
   return 0;
}
```

Now, let us discuss, few differences in using 'const' between C++ and C. In C++ constant data members can be used in expression like,

```
const int SIZE = 25;
int director[SIZE];
```

But this is not allowed in C. In C++, a variable declared as const is local to the file in which it is declared. But in C, a constant variable is global, which means, another file other than in which it is defined can also access the constant variable using the 'extern' keyword. To make a constant variable local to a file in C, we need prefix the constant variable declaration with 'static' keyword. In C++, the constant data member has to compulsorily be initialized to some value at the time of its creation, failing which it will generate a compile time error. But this is not the case in C.

- **Mutable data members:**

Whenever a member function is made constant, as studied earlier, using the const qualifier as the suffix to its function definition header and to its function prototype, then, the function cannot modify the data member of its class. But, if the need arises, such that, the constant member functions has to modify the value of the data member, then the data member has to be declared by prefixing the keyword 'mutable' as shown below.

```
mutable int x = 5;
```

Consider the below program 3.11.

Program 3.11

```
/*3.11.cpp*/
#include <iostream>
using namespace std;
class X
{
        int a;
        mutable int b;
   public :
      void xyz( ) const
      {
         a++;          // Error: cannot modify the data member
         b++;         // Legal : can modify mutable data member
      }

};
int main( )
{
   X  x1;
   x1.xyz( );
   return 0;
}
```

In the program 3.11, the 'a' is a normal data member, 'b' is the mutable data member and 'xyz( )' is the constant member function of the class 'X'. We have learnt that, the constant member function cannot modify the value of the data member of a class. Hence in our program, when 'xyz( )' tries to modify the data member 'a', an error is generated. But the same function can modify the mutable data member 'b' of the class. Hence to modify a data member of a class using the constant member function, that particular data member should be made mutable. In other words, the data member declaration should be preceded by the keyword 'mutable'.

Static data members:

Static data members are those members whose members are accessed by all the objects of a class. Static data member is not owned by any object of a class, but is essentially a data member of a class. Every object of a class can access the static data member. There is only one copy of a static data member created for a class which can be accessed by all the objects of that class. The scope of a static data member is

within a class, but its lifetime is the entire program. A data member can be made stat by preceding the keyword 'static' to the data member declaration as shown below.

```
static int a;
```

The static data member is initialized to zero when it is created. static variables a used to maintain values common to the entire class.

Program 3.12

```
/*3.12.cpp*/
#include <iostream>
using namespace std;
class simple
{
      static int a;
      int num;
   public:
      void get_data( int c )
      {
         num = c;
         a++;
      }
      void count( void )
      {
         cout << a << endl;
      }
};
int simple :: a; // memory allocated for static data member
int main( )
{
   simple x, y;
   x.count ( );
   y.count ( );
   x.get_data(1);
   y.get_data(2);
   x.count( );
   y.count( );
   return 0;
}
```

Output:

```
0
0
2
2
```

From the output of the above program 3.12, it can be easily understood that, if the value of the static data member is changed using an object, that is effected to the other objects also. This proves that, each object of a particular class shares a copy of static data member.

The static data members does not form a part of the class size. This is because the static data members are created separately and not when the object is created. It can be proved with the below program 3.13.

Program 3.13

```
/* 3.13.cpp */
#include <iostream>
using namespace std;
class A
{
   public :
      static int a; // static data member
      int b;
}
int A :: a;
int main( )
{
   A obj;
   cout << sizeof( obj ) << endl;
   return 0;
}
```

Output:

```
4
```

From the above program it can be easily proved that static data members does not form a part of the class size.

In the program 3.12, note the statement,

```
int sample :: a;
```

This is the definition of the static data member. The memory is not allocated for the static data member when the object is created. This is because the static data member is not a part of any of the class objects. Here the data member is defined outside the class declaration using the scope resolution operator. Hence the static data members does not form a part of the class size. The static data member can be initialized while defining it. It is as shown below.

```
class A
{
   static int a;
};
int A :: a = 3;     // or int A :: a(3);
```

Static member functions:

Static member functions are those functions of a particular class used to access or mo the static members(functions or variables) of the same class. To make a member funct static, its prototype should be prefixed with the keyword 'static'. static member functions be called without using an existing object. That is, it can be called using the class name should be noted that, the static member function can be called with respect to objects a The static member function is used in the below program 3.14.

Program 3.14

```
/* 3.14.cpp */
#include <iostream>
using namespace std;
class sample
{
      static int a;
   public:
      static void initialize( int x ) //static member function
                                      //definition
      {
         a = x;
      }
      void display( )
      {
         cout << a << endl;
      }
};
int sample :: a;        //Static data member definition
int main( )
{
   sample s;
   sample :: initialize ( 1 ); // invoking static member function
   s.display( );
   return 0;
}
```

Output:

```
1
```

In the above program 3.14, 'initialize()' is the static member function which is invoked using the class name and scope resolution operator using the statement,

```
sample :: initialize(1);
```

The static member function definition gets invoked and the value of the static data member is modified by this member function by a value 1. The value of the static data member is displayed through the member function 'display()' of the class 'sample'.

As told earlier the static member function can also be invoked using the class objects. It is really a nice idea to analyze a program of such sort as shown below.

Program 3.15

```
/* 3.15.cpp */
#include <iostream>
using namespace std;
class sample
{
      static int a;
   public:
      static void initialize( int x ) //static member function
                                       //definition
      {
         a = x;
      }
      void display( )
      {
         cout << a << endl;
      }
};
int sample :: a;          //Static data member definition
int main( )
{
   sample s1, s2;
   s1.initialize ( 1 ); // invoking static member function
   s1.display( );
   s2.display( );
   s2.initialize( 10 );
   s1.display( );
   s2.display( );
   return 0;
}
```

Output:

```
1
1
10
10
```

In the above program, the static member function 'initialize( )' is called using the objects s1 and s2. This code works absolutely fine. Keep in mind that, a single copy of the static data member is shared by all the objects of a class. So the modifying the static data member using one object, will be reflected for the other objects also. This point is proved in the above example program 3.15. Observe that in the above program, the static data member is being changed using the statement,

```
s1.initialize ( 1 ); // using object s1
```

and the changed value is printed on the monitor, using the statement,

```
s2.display( );        // using object s2.
```

Hence the output shows that the change done using the object 's1' is also available for display using the object 's2'.

It should be noted that, the static member functions are used to access static data members only. Static member functions cannot be used to access the non-static data member. Attempt to modify or acces the non-static data members by a static data member function generates a compilation error. Let us prove this by an example program 3.16.

Program 3.16

```
/* 3.16.cpp */
#include <iostream>
using namespace std;
class sample
{
      static int a;
      int b;
   public:
      static void initialize( int x ) //static member function
                                      //definition
      {
         a = x;
         b++;// Error
      }
      void display( )
      {
```

```
        cout << a << endl;
    }
};
int sample :: a;//Static data member definition
int main( )
{
    sample s;
    sample :: initialize ( 1 ); // invoking static member function
    s.display( );
    return 0;
}
```

In the above program, 'b' is a data member of class sample, which is being accessed by a static member function. As this is not allowed, the compiler generates an error message.

3.3. FRIEND FUNCTIONS

We know that, a non-member function of a class cannot have access to the private data of that class. But sometimes if such a need arises, where in, the private data members of a class should be accessed by non-member functions, C++ allows the non-member function to be made as a friend of that class. Now the non-member function, which is declared as a friend of a particular class, can have access to the private data members of that class to which it has been declared as a friend. To make an outside function friendly to the class, we have to declare the non-member function as a friend of the class by preceding the 'friend' keyword before the function prototype inside the class as shown below.

Syntax:

```
friend <data_type> <function_name>( );
```

where,

`friend` → a keyword to make a function friend to a class

`data_type` → return type of the friend function

`function_name` → name of the friend function

Example:

```
class class_name
{
    ..............
    ..............
    public:
```

```
        friend void print( ); //friend function
    };
```

Lets discuss the different types of friend functions.

- **Friend non-member functions:**

When an outside function, which does not belong to any of the other class, in other words, which is not a member any other class, is made as friend of a particular class, it is known as friend non-member functions. It is well depicted in the below program 3.17

Program 3.17

```
/*3.17.cpp*/
#include <iostream>
using namespace std;
class sample
{
      int a;
   public:
      friend void change( sample & ) ; //Prototype of friend
                                       //function.
};
void change( sample &x ) // friend function definition
{
   x.a = 5; //accessing private data members of the object
}
int main( )
{
   sample A;
   change( A );
   return 0;
}
```

In the above program, it is to be observed that, 'change( )' is a non-member function which does not belong to any of the class and hence it is defined outside the class without being referred to any class and scope resolution operator is also not used. Moreover, it is called without using the object to member access operator in the main, and the reason behind this is quite obvious. That is, the friend function is not a member function of any class. The function 'change( )' after being made as a friend of class 'sample', it can now have access to all the private data members of the class 'sample'. This is evident from the statement

```
x.a = 5;
```

in the program 3.17.cpp. But it is also very important to note that, while calling the friend function to access the data members of a particular class object , we have to pass the relevant class object as reference to the function definition.

- **Friend member functions:**

  If a member function of class say, 'dimple' wants to access the private data members of another class say 'simple', then the member function of class 'dimple' has to be declared as the friend of the class 'simple' within the class 'simple' declaration. This type of friend functions are known as friend member functions. In this case the friend function is declared in a different manner as shown in the example program 3.18.

Program 3.18

```
/*3.18.cpp */
#include <iostream>
using namespace std;
class simple;      //Forward declaration
class dimple
{
   public:
      void set_data( simple &, int );
};
class simple
{
   private :
      int x;
   public:
      int get_data( void );
      friend void dimple :: set_data( simple &, int );
};
void dimple :: set_data( simple &a, int b )
{
   a.x = b; // accessing data member of class simple
}
int simple :: get_data( void )
{
   return x;
}
int main( )
{
   simple e;
   dimple f;
   f.set_data( e, 5 );
```

```
    cout << e.get_data( ) << endl;
    return 0;
}
```

Output:

```
5
```

In the above program, 'set_data( )' is the member function of class 'dimple', which is declared as a friend of class 'simple'. The statement in the above program 3.18,

```
friend void dimple :: set_data( simple &, int );
```

declares the function 'set_data( )' of class 'dimple', as the friend function of class 'simple'. Here scope resolution operator is required, because, the friend function 'set_data( )', is a member function of class 'dimple'. From now on, the member function 'set_data( )' of class dimple can have access to the private data members of the class 'simple'. This is evident from the statement,

```
a.x = b;
```

in the program 3.18.

The following line in the above program 3.18,

```
class simple;
```

is known as forward declaration. If this is not done, then the compiler flashes error while compiling the declaration statement,

```
void set_data( simple &, int );
```

as the class 'simple' would have not been defined. Hence, to avoid this, we have just declared the class 'simple' before defining it. Now the compiler understands that the class 'simple' will be defined further.

- **Functions friend to more than one class:**

  An independent function belonging to none of the class can be declared as the friend of more than one class. Here, the friend function can access the private data members of all the class, to which it is declared as a friend.

Program 3.19

```
/*3.19.cpp*/
#include <iostream>
using namespace std;
class A;          //forward declaration
class X
```

```
{
      int x;
   public :
      void set_value( int i )
      {
         x = i;
      }
      friend void maximum( X, A);
};
class A
{
      int a;
   public:
      void set_value( int i )
      {
         a = i;
      }
      friend void maximum( X, A );
};
void maximum( X m, A  n)
{
   if( m.x >= n.a )
      cout << m.x;
   else
      cout << n.a;
}
int main( )
{
   A a ;
   a.set_value( 1 ) ;
   X x ;
   x. set_value( 2 ) ;
   maximum( x, a ) ;
   return 0 ;
}
```

Output :

2

In the above program, 'maximum( )' is a function which does not belongs to any of the classes; neither class 'A' nor class 'X'. But it has been made as a friend function of both the classes, which means to say that, now the friend function 'maximum( )', can have access to all the private members of both the classes as shown in the program 3.19

above. The function definition of the friend function 'maximum( )' in the program 3.19, as shown below,

```
void maximum( X m, A  n)
{
   if( m.x >= n.a )
      cout << m.x;
   else
      cout << n.a;
}
```

justifies that, the friend function can have access to both the private data members of both the classes.

It is to be noted that in the above program, the objects of both the classes 'A' and 'X' have to be passed to the friend function 'maximum( )'.

Characteristics of Friend functions:

- The friend function can be invoked without the help of any object.
- The friend function is not in the scope of the class to which the function has been declared as friend.
- The friend functions usually have objects as their arguments.
- The friend function can be declared either in the private or public section of the class.
- The friend function has to be preceded by the keyword 'friend'.
- Unlike member functions, the friend function cannot access the member names directly. The data members of the class has to be accessed using an object name and object to member access operator(dot operator (.)).

3.4. FRIEND CLASS

A class can be made as a friend of another class. Let us discuss in detail. If, a class 'Y' is a friend of class 'X', then, all the member functions of class 'Y' can access the private members of class 'X'. But the member functions of class 'X' is restricted in accessing the private members of the class 'Y'. To declare class 'Y' as a friend of class 'X', then the following statement has to be included in either the private or public section of class 'X'.

```
friend class Y ;
```

The friend class example is given below in the program 3.20.

Program 3.20

```
/*3.20.cpp */
#include <iostream>
class Y;              //Forward declaration
class X
{
     int x;
     int y;
  public:
     void show_data( );
     friend class Y;
};
class Y
{
  public:
     void change_data( X &, int );
     void change_other_data( X &, int );
};
void Y :: change_data( X &c, int q )
{
   c.x = q;
}
void Y :: change_other_data( X &d, int p )
{
   d.y = p;
}
void X :: show_data( )
{
   cout << x << endl << y << endl;
}
int main( )
{
   X x1 ;
   Y y1 ;
   y1.change_data( x1, 5 ) ;
   y1.change_other_data( x1, 6 ) ;
   x1.show_data( );
   return 0;
}
```

Output

```
5
6
```

As evident from the above program 3.20, class 'Y' can access all the private member of class 'X'. But the reverse is not true. Friend classes are very rarely used in practice.

Now, suppose we have three classes, class A, class B and class C. class B is a friend of class A. And class C is a friend of class B. The question is "can the member functions of class C access the private members of class A ?" The answer is "No". It is because, only the member functions of class B can access the private members of class A. But not its friend classes. That is "The access rights is not transferable to its friend class". This is very well understood by analyzing the program 3.21 shown below.

Program 3.21

```
/* 3.21.cpp */
#include <iostream>
using namespace std;
class B;                        //Forward declaration
class C;                        //Forward declaration
class A
{
   friend class B;
   int x;
};
class B
{
   public:
      friend class C;
};
class C
{
   public:
      int set_other_data( A &, int ); //Error
};
int C :: set_other_data( A &c, int q );
{
   c.x = q; //Error
   return c.x;
}
int main( )
{
   A a1;
   B b1;
   C c1;
   cout << c1.set_other_data( a1, 5 ) << endl;
   return 0;
}
```

3.5. ARRAY OF CLASS OBJECTS

The declaration of array of objects is very much similar to declaration of array of structures. As an array can be of any data type, we can have arrays of variables that are of the class type. Such variables are called as "Array of objects". Array of objects are greatly used while dealing with applications pertaining to database. The various ways of declarations of arrays of objects are presented here.

Syntax 1:

```
class <class_name>
{
   private :
   ..........
   public:
   ..........
};
class <class_name> <object_name[SIZE]>;
```

where,

`object_name` → is an array name which indicates the object name

`SIZE` → array subscript that indicates the number of objects to be created

Syntax 2:

```
class <class_name>
{
   private:
   ..................
   public:
   ..................
} <object_name[SIZE]>;
```

Syntax 3:

```
class
{
   ..................
}<object_name[SIZE]>;
```

Example:

```
class employee
{
      char name[20];
      float salary;
```

```
    public:
       void get_details( void );
       void display_details( void );
};
```

For the above class declaration, the following statement;

```
employee director[5];
```

forms an array of 5 objects named director. Each object have a copy of the private data members and all objects share a single copy of member functions. We know that, the array indexing starts from 0th position. The statement,

```
director[0].display_details( ) ;
```

will display the details of the 1st director of class employee. The statement,

```
director[1].display_details( );
```

will display the details of the 2nd director of class employee. Similarly, the statement,

```
director[i].display_details( );
```

will display the details of the (i + 1)th director of class employee. Let us work out an example program as shown below.

Program 3.22

```
/* 3.22.cpp */
#include <iostream>
using namespace std;
class employee
{
      char name[40];// character array within a class; a data member
      float salary;
   public:
      void get_details( );
      void display_details( );
};
void employee :: get_details( )
{
   cout << "Enter name = " ;
   cin >> name;
   cout << "Enter salary = " ;
   cin >> salary;
}
void employee :: display_details( )
{
   cout << "Name = " << name << endl;
   cout << "Salary = " << salary << endl;
}
```

```
int main( )
{
   int j;
   employee director[3];          //Array of 3 objects

   for( j = 0; j < 3; j++ )
   {
      cout << "Details of director" << j +1 << endl;
      director[j].get_details( );
   }
   cout << endl;
   for( j = 0; j < 3; j++ )
   {
      cout << "The entered details of director" << j + 1 << endl;
      director[j].display_details( );
   }
   return 0;
}
```

Output :

```
Details of director1
Enter name = Rajesh ( enter )
Enter salary = 50000.00 ( enter )
Details of director2
Enter name = Subhash ( enter )
Enter salary = 40000.00( enter )
Details of director3
Enter name = Sumesh ( enter )
Enter salary = 30000.00( enter )

The entered details of director1
Name = Rajesh
Salary = 50000.00

The entered details of director2
Name = Subhash
Salary = 40000.00

The entered details of director3
Name = Sumesh
Salary = 30000.00
```

In the above program 3.21, as can be observed, the statement,

```
employee director[3];
```

creates three objects named director[0], director[1], director[2] of class employee. Here in this example program 3.22, we just enter the name and salary of each director using the member function 'get_details( )' and display the entered details using the member function 'display_details( )'. The idea behind this program is to show the readers how the array of objects are created. The remaining part of the program is quite simple to trace.

Pictorially, the array of objects looks like as shown in figure 3.1 in the memory.

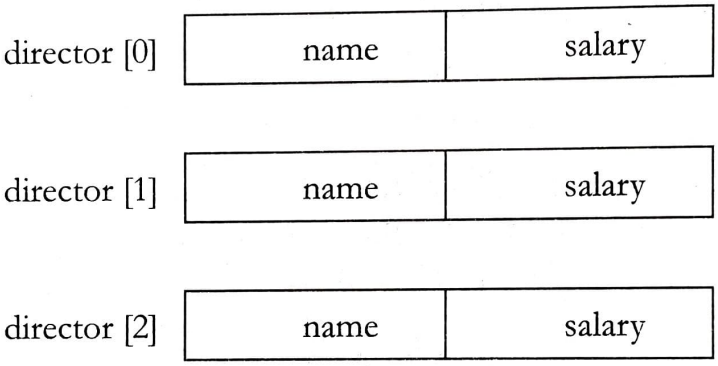


Figure 3.1

3.6. PASSING CLASS OBJECTS TO FUNCTIONS

Class objects can be passed as arguments to functions, like any other data type. The objects can be passed in three ways. They are;

- Passing objects by value
- Passing objects by reference
- Passing objects by pointers

Let us understand each with an example.

Passing objects by value :

Here, only a copy of the object is passed to the function definition. The modification on the objects made in the called function will not be reflected in the calling function. This is evident from the program 3.23.

Passing Objects by reference:

Here, when the objects are passed to the function definition, the formal arguments of the called function, shares the memory location of the actual arguments. Hence the change made on the objects in the called function will be reflected in the calling function also.This is evident from the program 3.23.

Passing objects by pointers:

Here, a pointer to an object is passed. The members of the objects passed are accessed by the arrow operator (->). The change of object in the called function will be reflected in the calling function also. This is evident from the program 3.23.

Program 3.23

```
/*3.23.cpp */
#include <iostream>
using namespace std;
class A
{
   public:
      int a;
      void set( A , int ); //call by value
      void set( int,  A & ); // call by reference
      void set( A *, int );// call by pointer
};
void A :: set( A x, int p )
{
   x.a = p;
}
void A :: set ( int q, A &y )
{
   y.a = q;
}
void A :: set( A *z, int r )
{
   z->a = r;
}
int main( )
{
    A a1;
   a1.a = 10;
   cout << "Before passing a = " << a1.a << endl;
   a1.set( a1, 5 ); //by value
   cout << "After passing by value a = " << a1.a << endl;
   a1.set( 8, a1 ); // by reference
   cout << "After passing by reference a = " << a1.a << endl;
   a1.set( &a1, 6 );// by pointer
   cout << "After passing by pointer a = " << a1.a << endl;
   return 0;
}
```

Output:

```
Before passing a = 10
After passing by value a = 10
After passing by reference a = 8
After passing by pointer a = 6
```

As can be observed from the above program 3.23, an overloaded member functio 'set( )' is used to assign a value to the data member 'a' of the class 'A'. As learned earlie an overloaded function can have same name, but should differ either in the number, typ or order of passing the arguments to the function definition. Here, each time, the 'set(function is called with different types of arguments and also differ in the order in which th arguments are passed to the function definition. First time, the statement,

```
a1.set( a1, 5 );
```

invokes the 'set( )' function, where only a copy of the object is passed to the functio definition. In this function definition, a new copy of an object 'x' is created, and its valu is being changed. This change does not affect the original copy of the object. Once th control comes out of this function definition, the memory occupied by the new object deallocated.

Second time, the statement,

```
a1.set( 8, a1 ) ;
```

invokes its appropriate function definition. This time, the object 'a1' is passed as reference. This means that, in the function definition, 'y' is made as a reference to the origin object 'a1'. So changing the value of the 'y' will be reflected in the original object also. It because, 'y' becomes an alias to the existing object 'a1'. In other words, 'y' is another nam referring to the same memory location occupied by the original object 'a1' .

Third time, the statement,

```
a1.set( &a1, 6 );
```

invokes the appropriate function definition. This time, the address of the object 'a1' passed to the function definition, which is stored in another pointer variable in that definitio In our program code, the address of the object is stored in a pointer variable 'z' which is o type 'class A'. The change in data member through this pointer will be reflected in the origin copy of the object 'a1'.

The output can be analyzed easily with reference to the above explanation.

3.7. RETURNING OBJECTS FROM FUNCTIONS

We are very much familiar with returning variables of built in data types. In the same way, functions can also return objects(class variables) itself to its caller. The syntax is very much similar to those that return variable of other data type to the caller. We have to prefix the type (class name)of the object while declaring the function.

Syntax:

```
<class_name> <function_name>( );
```

The above syntax statement shows how to prototype a function returning an object where,

`class_name` → data type of the object returned by the function

`function_name` → name of the function

Let us write an example program to understand the concept clearly.

Program 3.24

```
/*3.24.cpp*/
#include <iostream>
using namespace std;
class student
{
     int m1, m2;
  public:
     void get( )
     {
        cout << "Enter the marks m1" << endl;
        cin >> m1;
        cout << "Enter the marks m2" << endl;
        cin >> m2;
     }
     void show( )
     {
        cout << "Marks in m1 = " << m1 << endl;
        cout << "Marks in m2 = " << m2 << endl;
     }
     student  increased( student  );
};

student student :: increased( student s )
```

```
{
    student st;
    st.m1 = m1 + s.m1;
    st.m2 = m2 + s.m2;
    return ( st );
}

int main( )
{
    student s1, s2, s3;
    cout << "Enter the student's marks" << endl;
    s1.get( );
    cout << "Enter the increased marks" << endl;
    s2.get( );
    s3 = s1.increased( s2 );
    s3.show( );
    return 0;
}
```

Output:

```
Enter the student's marks
Enter the marks m1
90
Enter the marks m2
85
Enter the increased marks
Enter the marks m1
5
Enter the marks m2
6
Marks in m1 = 95
Marks in m2 = 91
```

In the above program 3.24, 'increased( )' is the member function, which passes an object 's2' of class 'student' as argument to the function definition, and the function definition returns an object of the same class type. The remaining part of the program is self-explainable.

3.8. NESTED CLASSES

A class embedded within another class is called a nested class. The class which is nesting a class is known as enclosing class or outer class, and the nested class is known as inner class. Nested classes can be defined in the private, public or protected(discussed

inheritance chapter) section of the enclosing class. The name of the nested class or inner class is inside the local scope of its enclosing class. Nested classes are very rarely used.

The general syntax of class declaration looks like this:

```
class enclosing_class_name
{
   private :
      ..................
   public :
      class nested_class_name
      {
         private :
            ..................
         public :
            ..................
      };
}
```

The syntax for creating objects for both the class looks like this:

i. For creating an object of enclosing class(in a normal way)

```
<enclosing_class_name> <enclosing_object_name>;
```

where,

`enclosing_class_name` → type of the object created(enclosing class type)

`enclosing_object_name` → name of the object created for enclosing class of type enclosing_class_name

ii. For creating object of nested class

```
<enclosing_class_name>::<nested_class_name> <nested_object_name>;
```

where,

`enclosing_class_name` → name of the enclosing class embedding the inner class

`nested_class_name` → type of the object created for nested class(nested class type)

`nested_object_name` → object of the nested class of type `nested_class_name`

An example program is given below.

Program 3.25

```
/*3.25.cpp*/
#include <iostream>
using namespace std;
class enclose    // outer class
{
   private
      int a;
      double b;
   public:
      void set_enclose( );
      void get_enclose( ) ;
      class inner     // inner class
      {
         private
            int c;
            double d;
         public:
            void set_inner( )
            {
               c = 5;
               d = 2.5;
            }
            void get_inner( )
            {
               cout << c << endl;
               cout << d << endl;
            }
      };
};
void enclose ::set-enclose( )
{
   a = 9;
   b = 10.12;
}
void enclose :: get_enclose( )
{
   cout << a << endl;
   cout << b << endl;
}
int main( )
{
   enclose e;
   e.set_enclose( );
   e.get_enclose( );
```

```
    enclose :: inner i;
    i.set_inner( );
    i.get_inner( );
    return 0;
}
```

Output:

```
9
10.12
5
2.5
```

In the above program, class 'enclose' is the outer class or enclosing class and class 'inner' is the inner class or nested class. It is evident that, the class 'inner' is within the scope of the class 'enclose'. If desired there can be a another class 'inner' outside the class 'enclose'. This does not have a conflict with the inner class 'inner' because, the nested class has its scope only withing the enclosing class 'enclose'. The object for the nested class is defined by following its enclosing class scope. It is shown in the above program as :

```
enclose :: inner i;
```

One important thing to be noted is that, the size of the objects of an enclosing class is not affected by the presence of the nested class. It is proved by the following example program 3.26.

Program 3.26

```
/*3.26.cpp*/
#include <iostream>
using namespace std;
class enclose
{
      int a;
   public:
      class nest
      {
         int b;
      };
};
int main( )
{
   cout << sizeof( enclose ) << endl;
   return 0;
}
```

Output:

Now, one question arises in our mind: How to define the member function of a nested class outside the enclosing class? It is quite simple. The following program 3.27 will help us understand the answer for the raised question above.

Program 3.27

```
/*3.27.cpp */
#include <iostream>
using namespace std;
class A  //outer class or enclosing class
{
   public:
      class B //Nested class or inner class
      {
            int b;
         public:
            void print( ); //member function of nested class
      };
};
void A :: B :: print( ) //defining member function of nested class
{
   cout <<"I am subhash\n";
}
int main( )
{
   A :: B b; //creating object of nested class
   b.print( );
   return 0;
}
```

Output:

```
I am Subhash
```

3.9. NAMESPACES

C++ introduces a new keyword 'namespace' to define a scope that could hold global identifiers. It enables the programmer to prevent pollution of global namespace that leads to name clashes. The best example of namespace scope is the C++ standard library, where all the classes, functions and templates are declared within the namespace 'std'. Hence while writing a C++ program, we usually include the directive

```
using namespace std;
```

The statement 'using namespace std;' specifies that the members defined in the 'std' namespace will be used in the program very frequently. A namespace defines the scope. The general syntax of using namespace in a program is shown below.

```
namespace name_of_namespace
{
   //declarations of variables, functions, classes etc.
}
```

It is to be noted that, there is no semicolon(;) after the closing brace, which forms the difference between defining a class and defining a namespace.

Example:

```
namespace check
{
      int x;
      void show( int y )
      {
         cout << y << endl;
      }
}
```

Here, the variable 'x' and function 'show( )' are inside the scope defined by the namespace 'check'. The value for x can be assigned using the qualifying namespace name followed by a scope resolution operator as shown below:

```
check :: x = 10;
```

Here, x is said to be qualified by the namespace name check. But always using the qualifier to access the variables becomes quite difficult. Hence in such situations, we use a 'using' directive to simplify their access.The syntax is as follows:

```
using namespace name_of_namespace;
```

Example:

```
using namespace check;
x = 2;                    // allowed
show( 3 );              // allowed
```

Here, all the members declared within the namespace check can be accessed directly without using the qualifying namespace name and scope resolution operator.

Check another example.

```
using namespace check :: x;
```

```
x = 200;                    // allowed
show(4 );                   // not allowed
```

Here only the specified member, i.e, 'x' can be accessed. Let us work out on some sample programs.

Program 3.28

```
/*3.28.cpp*/
#include <iostream>
using namespace std;
namespace subhash
{
   int a = 6;
   void display( );
   {
      cout << "Hello world" << endl;
   }
}
int main( )
{
   using namespace subhash;
   cout <<  a << endl;
   display( );
   return 0;
}
```

Output:

```
6
Hello World
```

In the above program 3.28, variable 'a' and function 'display( )' is defined in the scope 'subhash'. 'subhash' scope is created using the keyword namespace. Hence to use the variable 'a' and function 'display( )' in main( )'s scope, we need to use the statement

```
using namespace subhash;
```

if this statement is not used, an error is flashed. This is because, the variable 'a' and function 'display( )' are not available outside the scope of 'subhash' . The other way of doing the same is by preceding the variable name 'a' and function 'display( )' with namespace name followed by the scope resolution operator as follows.

Program 3.29

```
/* 3.29.cpp */
#include <iostream>
using namespace std;
```

```
namespace subhash
{
   int a = 6;
   void display( )
   {
      cout << "Hello world" << endl;
   }
}
int main( )
{
   cout <<  subhash :: a << endl;
   subhash :: display( );
   return 0;
}
```

Output:

```
6
Hello world
```

Now, let us write another program which produces errors. This helps the reader to understand this concept more better.

Program 3.30

```
/* 3.30.cpp */
#include <iostream>
using namespace std;
namespace subhash
{
   int a = 6;
   void display( )
   {
      cout << "Hello world" << endl;
   }
}
int main( )
{
   cout << a << endl;          //Error
   display( );//Error
   return 0;
}
```

Here in this program, we are trying to access the value of variable 'a' and also trying to call the function 'display( )' from 'main( )' function without using the namespace name. And hence this program produces the error, because 'a' and 'display( )' has been

accessed outside the scope of 'subhash'. It is left as an exercise for the reader to analyze the program 3.31 and 3.32.

Program 3.31

```
/*3.31.cpp*/
#include <iostream>
using namespace std;
namespace func
{
   int division( int x, int y )
   {
      return ( x / y ) ;
   }
   int add( int , int  ) ;
}
int func :: add( int x, int y )
{
   return ( x + y ) ;
}
int main( )
{
   using namespace func ;
   cout << division( 2 , 1 ) << endl ;
   cout << add( 2 , 1 ) << endl;
   return 0;
}
```

Output:

```
2
3
```

Program 3.32

```
/*3.30.cpp*/
#include <iostream>
using namespace std;
namespace subhash
{
   int a = 6;
   void display( )
   {
      cout << "Hello world" << endl;
   }
}
int main( )
```

```
{
   using namespace subhash :: a;
   cout <<  a << endl;       // Works fine
   display( );              //Error
   return 0;
}
```

Nested Namespaces :

C++ allows nesting of a namespace within another namespace as shown in the example below:

Example:

```
namespace A
{
   int x;
   namespace B
   {
      int y;
   }
}
```

The variables of nested namespaces can be accessed, using the following statements

```
A :: B :: y = 2;
   Or
using namespace A;
B :: y = 2;
```

Let us practice few example programs and analyze their outputs.

Program 3.33

```
/*3.33.cpp*/
#include <iostream>
using namespace std;
namespace n1
{
   double a = 3.14;
   int b = 3;
   namespace n2
   {
      double y = 1.2;
   }
}
int main( )
{
```

```
    cout << n1 :: a << endl;
    cout << n1 :: b << endl;
    cout << n1 :: n2 :: y << endl;
    return 0 ;
}
```

Output :

```
3.14
3
1.2
```

Program 3.34

```
/* 3.34.cpp */
#include <iostream>
using namespace std ;
namespace n1
{
   float a = 100.99;
   int b = 90;
   namespace n2
   {
      float c = 80.88;
   }
}
namespace n3
{
   int b = 35;
   int n = 42;
}
int main( )
{
   using namespace n1;
   cout << a << endl;
   cout << b << endl;
   cout << n2 :: c << endl;
   using namespace n3 :: n;
   cout << n << endl;
   cout << n3 :: b << endl;
   return 0;
}
```

Output:

```
100.99
90
```

```
80.88
42
35
```

Unnamed Namespaces:

Unnamed namespaces are also called anonymous namespaces. As the name indicated, unnamed namespaces are those namespaces which do not have a name. The members of unnamed namespace occupy the global scope and are accessible in all scopes following the declaration in the file. This allows you to create identifiers that are known only within the scope of single file. The general syntax is as follows.

```
namespace
{
   //declaration of variables, functions, classes etc.
}
```

Example:

Consider two files f1 and f2 as shown in the figure 3.2.

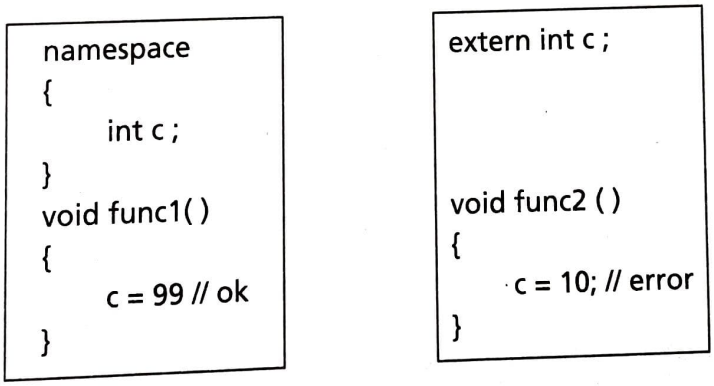


Figure 3.2

In the file f1, by declaring the variable 'c' with an unnamed namespace, the c's scope is restricted to only that file and access of 'c' in that file can be made without any qualifier. As the above diagram shows, another file f2, is trying to access the variable 'c' of f1, using extern. But, as the variable 'c' in file f1 is shielded by namespace, access of the variable 'c' outside of the file is not allowed, when both the files are tried to link. The linker flashes a link time error.

Let us write a program which has a unnamed namespace in it.

Program 3.35

```
/*3.35.cpp */
#include <iostream>
using namespace std;
```

```
namespace n1
{
   double a = 3.14;
   int b = 3;
   namespace n2
   {
      double y = 1.2;
   }
}
namespace                    // unnamed namespace
{
   int z = 1;
}
int main( )
{
   cout << n1 :: a << endl;
   cout << n1 :: b << endl;
   cout << n1 :: n2 :: y << endl;
   cout <<  z << endl ;
   return 0 ;                                          // z is global
}
```

Output :

```
3.14
3
1.2
1
```

Suppose we need to have two classes with the same name in a program, we can h
classes defined within two different scopes in the same program using namespace
classes are embedded within two different namespace scope. While creating the obje
these two classes, the normal object creation syntax should be preceded by the nam
name followed by the scope resolution operator as shown below. Let us look at an e
program.

Program 3.36

```
/* 3.36.cpp */
#include <iostream>
namespace X1
```

```
{
   class A
   {
      //class declaration or definition
   };
}
namespace X2
{
   class A
   {
      //class declaration or definition
   };
}
int main( )
{
   X1 :: A obj1;  // object creation of class A in namespace X1
   X2:: A obj2; // object creation of class A in namespace X2
   //other operations
}
```

3.10. Summary

- There are different types of member functions and data members.
- Member functions can be nested into another member function definition.
- Constant member functions are those member functions which cannot alter the value of their class data members.
- Static member functions can access only static data members.
- static data members are shared by all objects of a class.
- mutable data members are those data members which can be modified by the constant member functions.
- friend functions can access the data members of the class to which it is declared as a friend.
- namespace is a keyword in C++, which is used to create a new scope within a program and is also used to prevent pollution of the global namespace scope that leads to name clashes.

3.11. EXERCISES :

1. What are nested member functions ? Give example .
2. What are constant member functions? Give example.
3. What is the use of 'mutable' keyword ? Give example.
4. What is the purpose of static member functions ?
5. Why do we need static data members ? What is the syntax to define a static da member ?
6. Think of a real time application where static data member can be used.
7. How can we make a constant member function to change the value of its class's da members ?
8. Explain in brief the concept of 'friend' in C++ . Why is friend class used ?
9. Describe briefly the different ways to pass the class objects to function definitions
10. Can we return objects from functions ? Support your answer with example.
11. Explain the concept of 'namespace' in C++ with various examples.

* * * * *