Section-I

A. Review of Structures

1. Need for Structures

Need: Structures are required to group related variables (data) or dependent variables as single entity. Dependent variables are the variables where the value of one variable may influence the values of other variables.

Example: Let us illustrate the need for structures. Consider the example of date. A date can be programmatically represented by three different integer variables taken together. Say,

```
int d, m, y;        // Here 'd', 'm', and 'y' represent the day, the month, and the year, respectively.
```

The above three dependent variables are not grouped together in the code, but actually they belong to the same group called DATE.

Illustration for Dependency: Consider a function next_day() that accepts the addresses of the three integers that represent a date and changes their values to represent the next day.

```
Void next_day(int *,int *,int *);        //function to calculate the next day
```

Suppose, *d=31; m=12; y=1999;*

This statement represents 31st December, 1999 and if the function is called as: *next_day(&d,&m,&y);* then 'd' will become 1, 'm' will become 1, and 'y' will become 2000.

It is observed that, 'd', 'm', and 'y' actually belong to the same group. A change in the value of one may change the value of the other two.

Problem without Grouping: The DATE variables need to be placed in a same group because the members of the wrong group may be accidentally sent to the function *next_day(…);*.

```
Group-1:        d1=28; m1=2;  y1=1999;      //28th February, 1999
Group-2:        d2=19; m2=3;  y2=1999;      //19th March, 1999
Function call:  next_day(&d1,&m1,&y1);      //OK.
Function call:  next_day(&d1,&m2,&y2);      // Incorrect set passed!
```

Thus, there is a need of a language construct which groups the related variables together as a single entity.

Solution to the above Problem with Arrays: Suppose the next_day() function accepts an array as a parameter. Its prototype will be: *void next_day(int *);*

Let us declare date as an array of three integers.

```
int date[3];

date[0]=28; date[1]=2; date[2]=1999;
```

Now, let us call the function as follows: *next_day(date);*

The values of 'date[0]', 'date[1]', and 'date[2]' will be correctly set to 1, 3, and 1999, respectively. Although this method seems to work, it certainly appears unconvincing.

Solution to the above Problem with Structures: Create a data type called ***date*** itself using structures.

```
struct date    //a structure to represent dates
{
    int d, m, y;
};
```

Now, the *next_day()* function will accept the address of a variable of the structure date as a parameter. Accordingly, its prototype will be as follows: *void next_day(struct date *);*

```
struct date d1;
d1.d=28; d1.m=2; d1.y=1999;
next_day(&d1);
```

'd1.d', 'd1.m', and 'd1.y' will be correctly set to 1, 3, and 1999, respectively.

Definition: Structure is a programming construct that puts together the variables that should be together as a single entity.

Use of Structures: Library programmers use structures to create new data types. Application programs and other library programs use these new data types by declaring variables of newly created data type.

2. Creating a New Data Type Using Structures

Creation of a new data type using structures is a three-step process that is executed by the programmer (often by library programmer).

Step 1: Put the structure definition and the prototypes of the associated functions in a header file, as shown below.

```
/*Beginning of date.h*/
struct date
{
    int d,m,y;
};
void next_day(struct date *);          //get the next date
void get_sys_date(struct date *);   //get the current system date
/*
Prototypes of other useful and relevant functions to work upon variables of the date structure
*/
/*End of date.h*/
```

Step 2: As shown in the above code, put the definition of the associated functions in a source code and create a library.

```
/*Beginning of date.c*/
#include "date.h"
void next_day(struct date * p)
{
   //calculate the date that immediately follows the one represented by *p and set it to *p.
}
```

```
void get_sys_date(struct date * p)
{
  //determine the current system date and set it to *p
}
/*
Definitions of other useful and relevant functions to work upon variables of the date structure
*/
/*End of date.c*/
```

Step 3: Provide the header file and the library, in whatever media, to other programmers who want to use this new data type.

Note: Creation of a structure and creation of its associated functions are two separate steps that together constitute one complete process.kii

3. Using Structures in Application Programs

The steps to use this new data type are as follows:

Step 1: Include the header file provided by the library programmer in the source code.

```
/*Beginning of dateUser.c*/
#include "date.h"
void main( )
{
  ....
  ....
}
/*End of dateUser.c*/
```

Step 2: Declare variables of the new data type in the source code (i.e. in main function).

```
/*Beginning of dateUser.c*/
#include "date.h"
void main( )
{
  struct date d;
    ....
    ....
}
/*End of dateUser.c*/
```

Step 3: As shown in code below, embed calls to the associated functions by passing these variables in the source code.

```
/*Beginning of dateUser.c*/
#include "date.h"
void main()
{
   struct date d;
```

```
    d.d=28; d.m=2; d.y=1999;
    next_day(&d);
      . . . .
      . . . .
}
/*End of dateUser.c*/
```

Step 4: Compile the source code to get the object file.

Step 5: Link the object file with the library provided by the library programmer to get the executable or another library.

B. Procedure-Oriented Programming Systems

This is the foregoing pattern of programming which divides the code into functions/procedures. Here, the focus is on procedures. This programming pattern is, therefore, a feature of the procedure- oriented programming system. Data is passed from one function/procedure to another to be read from or written into.

Strength: In the procedure-oriented programming system, procedures are dissociated from data and are not a part of it. Instead, they receive variables(e.g. structure) or their addresses and work upon them. The code design is centered around procedures or functions.

Limitation: The drawback with this programming pattern is that the data is not secure. It can be manipulated by any procedure. Associated functions that were designed by the programmer do not have the exclusive rights to work upon the data. Functions are not a part of the data definition itself.

Scenario-1*:* Suppose the library programmer has defined a structure and its associated functions which are being used by an application program. The application program might modify the structure variables, not through the functions, but by some code inadvertently written in the application program itself. Compilers that implement the procedure-oriented programming system do not prevent unauthorized functions from accessing/manipulating structure variables.

Scenario-2: Consider a large application (around 25,000 lines of code) in which variables of the date structure have been used quite extensively. During testing, it is found that, the result is incorrect!.The faulty piece of code that is causing this bug can be anywhere in the program. Therefore, debugging will involve a visual inspection of the entire code (of 25000 lines!) and will not be limited to the associated functions only.

The situation becomes especially grave, if the execution of the code that is likely to corrupt the data is conditional. For example,

```
if(<some condition>)
  d.m++;   //d is a variable of date structure… d.m may become 13
```

Scenario-3: Let us assume that, there exists a compiler for procedure oriented languages. The complier enables the library programmer to assign exclusive rights to the associated functions for accessing the data members of the corresponding structure. If a function which is not one of the intended associated functions accesses the data members of a structure variable, a compile-time error will result. Thus, the application that arises out of a successful compile will be the outcome of a piece of code that is free of any unauthorized access to the data members of the structure variables used therein.

Observation: It is the lack of data security of procedure-oriented programming systems that led to object- oriented programming systems (OOPS).

C. Object-Oriented Programming Systems

OOP tries to model real-world objects. Most real-world objects have internal parts and interfaces that enable us to operate them. These interfaces perfectly manipulate the internal parts of the objects. They also have the exclusive rights to do so.

Scenario: Take the case of a simple LCD projector (a real-world object). It has a fan and a lamp. There are two switches—one to operate the fan and the other to operate the lamp. However, the operation of these switches is necessarily governed by rules. If the lamp is switched on, the fan should automatically switch itself on. Otherwise, the LCD projector will get damaged. For the same reason, the lamp should automatically get switched off; if the fan is switched off. In order to cater to these conditions, the switches are suitably linked with each other. The interface to the LCD projector has the exclusive rights to operate the lamp and fan.

Features: OOPS, with the help of a new programming constructs and new keywords, associated functions of the data structure can be given exclusive rights to work upon its variables. In other words, all other pieces of code can be prevented from accessing the data members of the variables of this structure.

Compilers that implement OOP, enable data security by diligently enforcing this prohibition. They do this by throwing compile-time errors against pieces of code that violate the prohibition.

In OOP, library programmers can ensure a guaranteed initialization of data members of structure variables to the desired values. For this, application programmers do not need to write code explicitly.

Two more features are incidental to OOPS. They are: (a) Inheritance (b) Polymorphism

Inheritance: Inheritance allows one structure to inherit the characteristics of an existing structure. In inheritance, data and interface may both be inherited. The parent structure can be given the general common characteristics while its child structures can be given the more specific characteristics. This allows code reusability by keeping the common code in a common place—the base structure. Otherwise, the code would have to be replicated in all of the child structures, which will lead to maintenance nightmares. Inheritance also enables code extensibility by allowing the creation of new structures that are better suited to our requirements as compared to the existing structures.

Polymorphism: Polymorphism is the phenomena by virtue of which the same entity can exist in two or more forms. In OOPS, functions can be made to exhibit polymorphic behaviour. Functions with different set of formal arguments can have the same name. Polymorphism is of two types: static and dynamic.

D. Comparison of C++ with C

C++ is an extension of C language. It is a proper superset of C language. This means that a C++ compiler can compile programs written in C language.

Apart from the keywords that implement the common programming constructs, C++ provides a number of additional keywords and language constructs that enable it to implement the object-oriented paradigm.

Example: Redefining the Date structure (date) in C++

```
/*Beginning of Date.h*/
class Date   //class instead of structure
{
    private:
       int d,m,y;
    public:
       Date();
       void get_sys_date();       //associated functions appear within the class definition
       void next_day();
};
/*End of Date.h*/
```

The following differences can be noticed between Date structure in C and C++:

- The **keyword class** has been used instead of struct.
- Two new keywords—private and public—appear in the code.
- Apart from data members, the class has a constructor which is member function. A class constructor is a function that has the same name as the class which is a member in the class. Incidentally, it has no return type specified.

E. Console Input/Output in C++

1. Console Output

The output functions in C language, such as printf(), can be included in C++ programs because they are anyway defined in the standard library. However, there are some more ways of outputting to the console in C++. The C++ tokens/constructs for this are:

a. **cout**: It is actually an object of the class ostream_withassign. It stands as an alias for the console output device, that is, the monitor.
b. **The << symbol**: It is originally the left shift operator, has had its definition extended in C++. In the output context, it operates as the insertion operator. It is a binary operator. It takes two operands. The operand on its left must be some object of the ostream_withassign class. The operand on its right must be a value of some fundamental data type. The value on the right side of the insertion operator is 'inserted' into the stream headed towards the device associated with the object on the left.
c. **The file iostream.h:** This file needs to be included in the source code to ensure successful compilation because the object cout and the insertion operator have been declared in that file.
d. **endl:** The object endl inserts a new line into the output stream.

Example-1: Outputting in C++ with Inserting new line by 'endl'

```
#include <iostream.h>
void main()
{
    int x, y;
    x=10; y=20;
```

```
    cout<<x;          //outputting to the console
    cout<<endl;       //inserting a new line by endl
    cout<<y;
}
```

Example-2: Outputting with cascading insertion operator for variables & constants

```
#include<iostream.h>
void main()
{
    int x; float y;
    x=10; y=2.2;
    cout<<x<<endl<<y;                          //cascading the insertion operator
    cout<<10<<endl<<"Hello World\n"<<3.4;       //outputting constants with \n and endl
}
```

2. Console Input

The input functions in C language, such as scanf(), can be included in C++ programs because they are anyway defined in the standard library. However, we do have some more ways of inputting from the console in C++.The C++ tokens/constructs for this are:

a. **Cin:** It is actually an object of the class istream_withassign. It stands as an alias for the console input device, that is, the keyboard.
b. **The >> symbol:** It is originally the right-shift operator, has had its definition extended in C++. In the input context, it operates as the extraction operator. It is a binary operator and takes two operands. The operand on its left must be some object of the istream_withassign class. The operand on its right must be a variable of some fundamental data type. The value for the variable on the right side of the extraction operator is extracted from the stream originating from the device associated with the object on the left.
c. **The file iostream.h:** It needs to be included in the source code to ensure successful compilation because the object cin and the extraction operator have been declared in that file.

Note: Just like the insertion operator, the extraction operator works equally well with variables of all fundamental types as its right-hand operand. It does not need the format specifiers that are needed in the scanf() family of functions.

Example-1: Inputting in C++ with cascaded extraction operator

```
#include<iostream.h>
void main()
{
    int x, y;
    char ch;
    cout<<"Enter two numbers\n";
    cin>>x>>y;                          //cascading the extraction operator
    cout<<"Enter a character: ";
    cin>>c;                             //Inputting with extraction operator
    cout<<"You entered  "<<x<<" , "<<y<< " and "<<ch;
}
```

F. Variables in C++

Variables in C++ can be declared anywhere inside a function and not necessarily at its very beginning.

Example: Declaring variables in C++

```
#include<iostream.h>
void main()
{
    int x;
    x=10;
    cout<<"Value of x= "<<x<<endl;
    int * iPtr;                          //declaring a variable in the middle of a function
    iPtr=&x;
    cout<<"Address of x= "<<iPtr<<endl;
}
```

G. Function Prototyping

Function prototyping is necessary in C++. A prototype describes the function's interface (signature) to the compiler. It tells the compiler the return type of the function as well as the number, type, and sequence of its formal arguments. The general syntax of function prototype is as follows:

return_type function_name(argument_list);

For example, *int add(int, int);*

This prototype indicates that the add() function returns a value of integer type and takes two parameters both of integer type.

Note: Since a function prototype is also a statement, a semicolon must follow it. Providing names to the formal arguments in function prototypes is optional. Even if such names are provided, they need not match those provided in the function definition.

Example: Function prototyping

```
#include<iostream.h>
int add(int,int);          //function prototype
void main()
{
    int x,y,z;
    cout<<"Enter a number: ";
    cin>>x;
    cout<<"Enter another number: ";
    cin>>y;
    z=add(x,y);            //function call
    cout<<z<<endl;
}
int add(int a,int b)       //function definition
{
    return (a+b);
}
```

Importance of Function Prototyping:

a) The return value of a function is handled correctly.

b) Correct number and type of arguments are passed to a function.

Illustration: The prototype tells the compiler that the add() function returns an integer-type value. Thus, the compiler knows how many bytes have to be retrieved from the place where the add() function is expected to write its return value and how these bytes are to be interpreted.

In the absence of prototypes, the compiler will have to assume the type of the returned value. Suppose, it assumes that the type of the returned value is an integer. However, the called function may return a value of an incompatible type (say a structure type).

c) The function's prototype also tells the compiler that the add() function accepts two parameters. If the program fails to provide such parameters, the prototype enables the compiler to detect the error. A compiler that does not enforce function prototyping will compile a function call in which an incorrect number and/or type of parameters have been passed. Run- time errors will arise as in the foregoing case.

NOTE: Since the C++ compiler necessitates function prototyping, it will report an error against the function call because no prototype has been provided to resolve the function call. Thus, function prototyping guarantees protection from errors arising out of incorrect function calls.

d) Finally, function prototyping produces automatic-type conversion wherever appropriate.

Example: Suppose, a function expects an integer-type value but a value of double type is wrongly passed. During runtime, the value in only the first four bytes of the passed eight bytes will be extracted. This is obviously undesirable. However, the C++ compiler automatically converts the double-type value into an integer type. This is because it inevitably encounters the function prototype before encountering the function call and therefore knows that the function expects an integer-type value.

H. Reference Variables in C++

A reference variable is nothing but a reference for an existing variable. It shares the memory location with an existing variable. The syntax for declaring a reference variable is as follows:

<data-type> & <ref-var-name>=<existing-var-name>;

Example-1: if 'x' is an existing integer-type variable, a reference variable named 'iRef' may be defined with the statement is as follows:

Int & iRef=x;

iRef is a reference to 'x'. *This means that although iRef and 'x' have separate entries in the OS*, their addresses are actually the same. Thus, a change in the value of 'x' will naturally reflect in iRef and vice versa.

Note-1: Reference variables must be initialized at the time of declaration. Reference variables are variables in their own right. They just happen to have the address of another variable. After their creation, they function just like any other variable.

Example-2: Role of reference variables

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

```
{
    int x;
    x=10;
    cout<<x<<endl;
    int & iRef=x;                    //iRef is a reference to x
    iRef=20;                         //same as x=10;
    cout<<x<<endl;
    x++;                             //same as iRef++;
    cout<<iRef<<endl;
}
```

Example-3: Reading the value of a reference variable

```
#include<iostream.h>
void main()
{
    intx,y;
    x=10;
    int & iRef=x;
    y=iRef;                  //same as y=x;
    cout<<y<<endl;
    y++;                     //x and iRef unchanged
    cout<<x<<endl<<iRef<<endl<<y<<endl;
}
```

Note-2: A reference variable can be a function argument and thus change the value of the parameter that is passed to it in the function call.

Example-4: Reference Variables as Function parameters

```
#include<iostream.h>
void increment(int&);        //formal argument is a reference to the passed parameter
void main()
{
    int x; x=10;
    increment(x);
    cout<<x<<endl;
}
void increment(int& r)
{
    r++;                     //same as x++;
}
```

Note-3: Functions can return by reference also.

Example-5: Returning by reference

```
#include<iostream.h>
int& larger(int&, int&);      // function prototype with two reference parameters & reference return
```

```
int main()
{
    intx,y;
    x=10; y=20;
    int& r=larger(x,y);
    r=-1;
    cout<<x<<endl<<y<<endl;
}

int& larger(int& a, int& b)
{
    if(a>b)          //return a reference to the larger parameter
       return a;
    else
      return b;
}
```

In the above program code, 'a' and 'x' refer to the same memory location while 'b' and 'y' refer to the same memory location. From the larger() function, a reference to 'b', that is, reference to 'y' is returned and stored in a reference variable 'r'. The larger() function does not return the value 'b' because the return type is int& and not int. Thus, the address of 'r' becomes equal to the address of 'y'. Consequently, any change in the value of 'r' also changes the value of 'y'.

Note-4: Returning the reference of a local variable

```
#include<iostream.h>
int & abc();
void main()
{
    abc()=-1;
}

Int & abc()
{
    int x;
    return x;      //returning reference of a local variable
}
```

The problem with the above program is that when the abc() function terminates, 'x' will go out of scope. Consequently, the statement abc()=-1; leads to run-time errors.

I. Function Overloading

C++ allows two or more functions to have the same name. For this, however, they must have different signatures. *Signature of a function means the number, type, and sequence of formal arguments of the function*. In order to distinguish amongst the functions with the same name, the compiler expects their signatures to be different. Depending upon the type of parameters that are passed to the function call, the compiler decides which of the available definitions will be invoked. For this, function prototypes should be provided to the compiler for matching the function calls.

Example-1: Function overloading

```
#include<iostream.h>
int add(int,int);          //first prototype
int add(int,int,int);      //second prototype

void main()
{
    int x, y;
    x=add(10,20);                  //matches first prototype
    y=add(30,40,50);               //matches second prototype
    cout<<x<<endl<<y<<endl;
}

int add(int a, int b)
{
    return(a+b);
}

int add(int a,int b,int c)
{
    return(a+b+c);
}
```

The two function prototypes at the beginning of the program tell the compiler the two different ways in which the add() function can be called. When the compiler encounters the two distinct calls to the add() function, it already has the prototypes to satisfy them both. Thus, the compilation phase is completed successfully. During linking, the linker finds the two necessary definitions of the add() function and, hence, links success fully to create the executable file.

The compiler decides which function is to be called based upon the number, type, and sequence of parameters that are passed to the function call. When the compiler encounters the first function call,

```
x=add(10,20);
```

it decides that the function that takes two integers as formal arguments is to be executed. Accordingly, the linker then searches for the definition of the add() function where there are two integers as formal arguments.

Function overloading is also known as function polymorphism because, just like polymorphism in the real world where an entity exists in more than one form, the same function name carries different meanings.

Function polymorphism is static in nature because the function definition to be executed is selected by the compiler during compile time itself. Thus, an overloaded function is said to exhibit static polymorphism.

Section-II

A. More on Structures

1. Limitations of Structures in C

The Date structure and its accompanying functions may be perfect. However, there is absolutely no guarantee that the client programs will use only these functions to manipulate the members of variables of the structure.

Example-1: Undesirable manipulation of structures not prevented in C

```
struct Date d1;
setDate(&d1);              //assign system date to d1.
printf("%d",d1.month);
d1.month = 13;             //undesirable but unpreventable!!
```

Notice that the absence of a facility to bind the data and the code that can have the exclusive rights to manipulate the data can lead to difficult-to-detect run-time bugs. C does not provide the library programmer with the facilities to encapsulate data, to hide data, and to abstract data.

2. Structures in C++

The C++ compiler provides a solution to this problem. Structures (the struct keyword) have been redefined to allow member functions also.

Example-2:C++ allows member functions in structures

```
#include<iostream.h>
struct Distance
{
    int iFeet;
    float fInches;
    void setFeet(int x)          // write feet – mutator
    {iFeet=x;}
    int getFeet()                // read feet - accessor
    {return iFeet;}
    void setInches(float y)       //write inches
    {fInches=y;}
    float getInches()             //read inches
    {return fInches;}
}
void main()
{
    Distance d1,d2; d1.setFeet(2); d1.setInches(2.2); d2.setFeet(3); d2.setInches(3.3);
    cout<<d1.getFeet()<<""<<d1.getInches()<<endl;
    cout<<d2.getFeet()<<""<<d2.getInches()<<endl;
}
```

The struct keyword has actually been redefined in C++. The functions have also been defined within the scope of the structure definition. This means that not only the member data of the structure can be accessed through the variables of the structures; but also the member functions can be invoked.

Member functions are invoked in much the same way as member data are accessed, that is, by using the variable-to-member access operator (dot operator). In a member function, one can refer directly to members of the object for which the member function is invoked. For example, as a result of the second line of the main() function in example, it is d1.iFeet that gets the value of 2. On the other hand, it is d2.iFeet that gets the value of 3 when the fourth line is invoked.

Each structure variable contains a separate copy of the member data within itself. However, only one copy of the member function exists.

Note-1: In the above example, note that the member data of structure variables can still be accessed directly. The following line of code illustrates this.

```
d1.iFeet=2;       //legal!!
```

3. Private and Public Members

Specifying member functions as public but member data as private gives the exclusive rights to the functions to work upon the data. Example-2 may be re-written as:

Example-3: Making members of structures private

```
#include<iostream.h>
struct Distance
{
private:
    int iFeet;
    float fInches;
public:
    void setFeet(int x)              // write feet – mutator
    {iFeet=x;}
    int getFeet()                    // read feet - accessor
    {return iFeet;}
    void setInches(float y)          //write inches
    {fInches=y;}
    float getInches()                //read inches
    {return fInches;}
}
void main()
{
    Distance d1, d2;
    d1.setFeet(2); d1.setInches(2.2);
    d2.setFeet(3); d2.setInches(3.3);
    d1.iFeet++;          // error – private member is accessed by non-member function i.e. main()
```

```
    cout<<d1.getFeet()<<""<<d1.getInches()<<endl;
    cout<<d2.getFeet()<<""<<d2.getInches()<<endl;
}
```

The presence of private & public keywords tells the compiler that iFeet and fInches are private data members of variables of the structure Distance and the member functions are public. Thus, values of iFeet and fInches of each variable of the structure Distance can be accessed/modified only through member functions of the structure and not by any non-member function in the program. Any attempt to violate this restriction is prevented by the compiler because that is how the C++ compiler recognizes the private keyword. Since the member functions are public, they can be invoked from any part of the program.

The keywords private and public are also known as ***access modifiers or access specifiers*** because they control the access to the members of structures. The member functions have not been placed under any access modifier. Therefore, they are public members by default.

Example-4: Structure members are public by default

```
struct Distance
{
    void setFeet(int x)       //public by default
    {
        iFeet=x;
    }
    int getFeet()             //public by default
    {
        return iFeet;
    }
    void setInches(float y)   //public by default
    {
        fInches=y;
    }
    float getInches()         //public by default
    {
        return fInches;
    }
    private:
        int iFeet;
        float fInches;
};
```

B. Object Oriented Programming - Basics

1. Class in C++

C++ introduces a new ***keyword class*** as a substitute for the keyword struct. In a structure, members are public by default.

Syntax: *class class_name*

```
{
    // member definitions
};
```

The class members are private by default. This is the only difference between the class keyword and the struct keyword. Thus, the structure Distance can be redefined by using the class keyword as shown in below:

Example-1: Class members are private by default

```
class Distance
{
    int iFeet;              //private by default
    float fInches;          //private by default
 public:
    void setFeet(int x)
    {
        iFeet=x;
    }
    int getFeet()
    {
        Return iFeet;
    }
    void setInches(float y)
    {
        fInches=y;
    }
    float getInches()
    {
        return fInches;
    }
};
```

2. Objects

Variables of classes are known as objects. An object of a class occupies the same amount of memory as a variable of a structure that has the same data members. Introducing member functions does not influence the size of objects.

Syntax: *class_name* $Obj_1, Obj_2, \ldots Obj_n$; where Obj_1, Obj_2 … are objects

Example-1: Size of a class object is equal to that of a structure variable with identical data members

```
#include<iostream.h>
struct A
{
    char a; int b; float c;
};
```

```
class B     //a class with the same data members
{
    char a; int b; float c;
};
void main()
{
    cout<<sizeof(A)<<endl<<sizeof(B)<<endl;
}
```

3. Scope Resolution Operator

It is possible and usually necessary for the library programmer to define the member functions outside their respective classes. The scope resolution operator makes this possible.

Syntax:

```
return_typeclass_name :: member_function(arguments)
{
    // member function definition
}
```

Example-1: The scope resolution operator

```
class Distance
{
     int iFeet;
     float fInches;
  public:
     void setFeet(int);          //prototype only
     int getFeet();              //prototype only
     void setInches(float);      //prototype only
     float getInches();          //prototype only
};
void Distance::setFeet(intx)     //definition
{
  iFeet=x;
}
int Distance::getFeet()          //definition
{
  return iFeet;
}
void Distance::setInches(float y)  //definition
{
  fInches=y;
}
float Distance::getInches()        //definition
{
  return fInches;
}
```

We can observe that the member functions have been only prototyped within the class; they have been defined outside. The scope resolution operator signifies the class to which they belong. The class name is specified on the left-hand side of the scope resolution operator. The name of the function being defined is on the right-hand side.

4. Creating Libraries using Scope Resolution Operator

Creating a new data type in C++ using classes is also a three-step process that is executed by the library programmer.

Step 1: Place the class definition in a header file named "Distance.h".

```
class Distance
{
    int iFeet;
    float fInches;
public:
    void setFeet(int);          //prototype only
    int getFeet();              //prototype only
    void setInches(float);      //prototype only
    float getInches();          //prototype only
};
```

Step 2: Place the definitions of the member functions in a C++source file(the library source code). A file that contains definitions of the member functions of a class is known as the **implementation file** of that class. Compile this implementation file and put in a library.

```
#include "Distance.h"
void Distance::setFeet(int x)      //definition
{
    iFeet=x;
}
int Distance::getFeet()            //definition
{
    return iFeet;
}
void Distance::setInches(float y)  //definition
{
    fInches=y;
}
float Distance::getInches()        //definition
{
    return fInches;
}
```

Step 3: Provide the header file and the library, in whatever media, to other programmers who want to use this new data type.

5. Using Classes in Application Programs

The steps followed by programmers for using this new data type are:

Step 1: Include the header file provided by the library programmer in their source code.

```
#include "Distance.h"
void main()
{
   ....
   ....
}
```

Step 2: Declare variables of the new data type in their source code.

```
#include "Distance.h"
void main()
{
    Distance d1, d2;
        ....
        ....
}
```

Step 3: Embed calls to the associated functions by passing these variables in their source code.

```
#include<iostream.h>
#include "Distance.h"
void main()
{
    Distance d1,d2;
    d1.setFeet(2); d1.setInches(2.2);
    d2.setFeet(3); d2.setInches(3.3);
    cout<<d1.getFeet()<<""<<d1.getInches()<<endl;
    cout<<d2.getFeet()<<""<<d2.getInches()<<endl;
}
```

Step 4: Compile the source code to get the object file.

Step 5: Link the object file with the library provided by the library programmer to get the executable or another library.

6. The 'this' pointer

The facility to create and call member functions of class objects is provided by the C++ compiler. The compiler does this by using a unique pointer known as ***the 'this' pointer***.

The 'this' pointer is always a constant pointer. The 'this' pointer always points at the object with respect to which the function was called.

The members of the invoking object are referred to when they are accessed without any qualifiers in member functions. It should also be obvious that multiple copies of member data exist (one inside each object) but only one copy exists for each member function.

It is evident that, the **this pointer** should continue to point at the same object—the object with respect to which the member function has been called—throughout its life time. For this reason, the compiler creates it as a constant pointer.

Example-1: Code Conversion by the C++ complier

Before Compilation:

```
int Distance::getFeet()
{
    return iFeet;
}
```

After Compilation:

```
int getFeet(Distance *const this)
{
    return this->iFeet;
}
```

Note: The accessibility of the implicit object is the same as that of the other objects passed as parameters in the function call and the local objects inside that function.

Example-2: Illustration of implicit 'this' pointer

```
class Distance
{
        :  :    :
    Distance add(Distance);
};
Distance Distance::add(Distancedd)
{
    Distance temp;
    temp.iFeet=iFeet+dd.iFeet;          //legal to access both temp.iFeet and dd.iFeet
    temp.fInches=fInches+dd.fInches;    //ditto
    return temp;
}
#include<iostream.h>
void main()
{
    Distance d1,d2,d3;
    d1.setFeet(1); d1.setInches(1.1);
    d2.setFeet(2); d2.setInches(2.2); d3=d1.add(d2);
    cout<<d3.getFeet()<<"'-"<<d3.getInches()<<"\"'\n";
}
```

Summary: we can

- Declare a class
- Define member data and member functions
- Make members private and public
- Declare objects and call member functions with respect to objects

The advantage is that, library programmers can now derive from this arrangement is epitomized in the following observation: An executable file will not be created from a source code in which private data members of an object have been accessed by non-member functions. Once again, the importance of compile-time errors over run-time errors is emphasized.

7. Data Abstraction

Data abstraction is a virtue by which an object hides its internal operations from the rest of the program. It makes it unnecessary for the client programs to know how the data is internally arranged in the object. Thus, data abstraction enforces the programmers to define member functions which include complete logic and suitable constructors to initialize member data.

Example-1: The library programmer, who has designed the Distance class, wants to ensure that the 'fInches' portion of an object of the class should never exceed 12.

If a value larger than 12 is specified by an application programmer while calling the Distance::setInches()function, the logic incorporated within the definition of the function should automatically increment the value of iFeet and decrement the value off Inches by suitable amounts. A modified definition of the Distance::setInches() function is as follows.

```
void Distance::setInches(float y)
{
    fInches=y;
    if(fInches>=12)
    {
        iFeet+=fInches/12;
        fInches-=((int)fInches/12)*12;
    }
}
```

Here, we notice that an application programmer need not send values less than 12 while calling the Distance::setInches()function. The default logic within the Distance::setInches() function does the necessary adjustments. This is an example of data abstraction.

Note: The library programmer would like to add a function to the Distance class that gets called automatically whenever an object is created and sets the values of the data members of the object properly. Such a function is the constructor. Thus Data abstraction is effective due to data hiding and perfect definitions of the member functions are guaranteed to achieve their objective because of data hiding.

8. Explicit Address Manipulation

An application programmer can manipulate the member data of any object by explicit address manipulation.

Example-1: Explicit address manipulation

```
#include "Distance.h"
#include<iostream.h>
void main()
{
    Distance d1;
```

```
    d1.setFeet(256); d1.setInches(2.2);
    char * p=(char *)&d1;              //explicit address manipulation
    *p=1;                              //undesirable but unpreventable
    cout<<d1.getFeet()<<""<<d1.getInches()<<endl;
}
```

However, such explicit address manipulation by an application programmer cannot be prevented.

9. The Arrow Operator

Member functions can be called with respect to an object through a pointer pointing at the object. The arrow operator (->) does this.

Example-1: Accessing members through pointers

```
#include<iostream.h>
#include "Distance.h"
void main()
{
    Distance d1;                   //object
    Distance * dPtr;               //pointer
    dPtr=&d1;                      //pointer initialized
    dPtr->setFeet(1);              /*Same as d1.setFeet(1) and d1.setInches(1.1)*/
    dPtr->setInches(1.1);          //through pointers
    cout<<dPtr->getFeet()<<""<<dPtr->getInches()<<endl;    //Same as d1.getFeet() and d1.getInches()
}
```

The definition of the arrow(->) operator has also been extended in C++ to take not only data members on its right, but also member functions as its right-hand side operand. If the operand on its right is a data member, then the arrow operator behaves just as it does in C language. However, if it is a member function of a class where a pointer of the same class type is its left-hand side operand, then the compiler simply passes the value of the pointer as an implicit leading parameter to the function call. Thus, the statement,

```
    dPtr->setFeet(1);
```

after conversion becomes

```
    setFeet(dPtr,1);
```

10. Calling One Member Function from Another

One member function can be called from another.

Example-1: Calling one member function from another

```
class A
{
    int x;
public:
    void setx(int);
    void setxindirect(int);
};
```

```
void A::setx(int p)
{
    x=p;
}
void A::setxindirect(int q)
{
    setx(q);
}
void main()
{
    A a1, a2;
    a1.setxindirect(1);          //member calls another member
    a2.setx(2);
}
```

C. Member Functions and Member Data

1. Overloaded Member Functions

Member functions can be overloaded just like non-member functions.

Example-1: Overloaded member functions

```
#include<iostream.h>
class A
{
  public:
    void show();
    void show(int);          //function show() overloaded!!
};
void A::show()
{
    cout<<"Hello\n";
}
void A::show(int x)
{
    for(int i=0;i<x;i++)
        cout<<"Hello\n";
}
void main()
{
    A a1;
    a1.show();                //first definition called
    a1.show(3);               //second definition called
}
```

Function overloading enables us to have two functions of the same name and same signature in two different classes.

Example-2: Member functions of two different classes to have the same name

```
class A
{
  public:
    void show();
};
class B
{
  public:
    void show();
};
```

A function of the same name show() is defined in both the classes—'A' and 'B'. The signature also appears to be the same. But with the 'this' pointer, the signatures are actually different.

*void show(A * const);*

*void show(B * const);*

2. Default Values for Formal Arguments of Member Functions

Default values can be specified for formal arguments of member functions in the prototype.

Syntax: *return_type mem_fun_name(arg1=val, …..);*

Example-1: Default values to arguments of member functions

```
#include<iostream.h>
class A
{
public:
    void show(int=1);
};
void A::show(int p)
{
    for(int i=0;i<p;i++)
        cout<<"Hello\n";
}
void main()
{
    A a1;
    a1.show();          //default value taken
    a1.show(3);          //default value overridden
}
```

3. Inline Member Functions

a) Inline Functions: Inline functions are used to increase the speed of execution of the executable files. An inline function is a function whose compiled code is 'inline' with the rest of the program. That is, the compiler replaces the function call with the corresponding function code. Within line code, the program does not have to jump to another location to execute the code and then jump back. Inline functions, thus, run a little faster than regular functions.

b) Inline Member Functions: Member functions are made inline by either of the following two methods.

- By defining the function within the class itself.
- By only prototyping and not defining the function within the class. The function is defined outside the class by using the scope resolution operator. The definition is prefixed by the **inline keyword**. As in non-member functions, the definition of the inline function must appear before it is called. Hence, the function should be defined in the same header file in which its class is defined.

Syntax*: inline retun_type mem_fun_name(args)*

```
{
        // definition of the member function
}
```

Example-1: Inline member functions

```
class A
{
  public:
    void show();
};
inline void A::show()      //definition in header file itself
{
    //definition of A::show() function
}
```

4. Constant Member Functions

Assume that, the library programmer desires that one of the member functions of his/her class should not be able to change the value of member data. This function should be able to merely read the values contained in the data members, but not change them. If the programmer declares the function as a constant function, and thereafter attempts to change the value of a data member through the function, the compiler throws an error.

Member functions are specified as constants by suffixing the prototype and the function definition header with the **const keyword**. The modified prototypes and definitions of the member functions of the class Distance are illustrated below:

Syntax: *return_type mem_function_name(args) const; // for prototype*

```
return_type mem_function_name(args) const        // for member definition
{
    // member function definition
}
```

Example-1: Constant member functions

```
class Distance
{
    int iFeet;
    float fInches;
  public:
    void setFeet(int);
    int getFeet() const;                    //constant function
```

```
    void setInches(float);
    float getInches() const;                //constant function
    Distance add(Distance) const;           //constant function
};

void Distance::setFeet(int x)
{
    iFeet=x;
}

int Distance::getFeet() const               //constant function
{
    iFeet++;                                //ERROR!!
    return iFeet;
}

float Distance::getInches() const           //constant function
{
    return fInches;
}
    :       :       :                       // other member definitions
```

For constant member functions, the memory occupied by the invoking object is a read-only memory. How does the compiler manage this? For constant member functions, the 'this' pointer becomes a constant pointer to a constant instead of only a constant pointer.

Note that, only constant member functions can be called with respect to constant objects. Non-constant member functions cannot be called with respect to constant objects. However, constant as well as non-constant functions can be called with respect to non-constant objects.

5. Mutable Data Members

A mutable data member is never constant. It can be modified inside constant functions also. Prefixing the declaration of a data member with the **keyword mutable** makes it mutable.

Syntax: *mutable data-type member_name;*

Example-1: Mutable data members

```
class A
{
    int x;          //non-mutable data member
    mutable int y;  //mutable data member
 public:
    void abc() const        //a constant member function
    {
        x++;    //ERROR: cannot modify a non-constant data member in a constant member function
        y++;    //OK: can modify a mutable data member in a constant member function
    }

    Void def()              //a non-constant member function
    {
        x++;    //OK: can modify a non-constant data member in a non-constant member function
        y++;    //OK: can modify a mutable data member in anon-constant member function
```

```
    }
};
```

6. Friend Functions & Classes

A class can have global non-member functions and member functions of other classes as friends. Such functions can directly access the private data members of objects of the class.

a) Friend Non-Member Functions

A friend function is a non-member function that has special rights to access private data members of any object of the class of whom it is a friend.

A friend function is prototyped within the definition of the class of which it is intended to be a friend. The prototype is prefixed with the **keyword friend**. Since it is a non-member function, it is defined without using the scope resolution operator. Moreover, it is not called with respect to an object.

Syntax: *friend return_type mem_fun_name(args);*

Example-1: Friend functions

```
class A
{
    int x;
 public:
    friend void abc(A&);    //prototype of the friend function
};
void abc(A& AObj)  //definition of the friend function
{
    AObj.x++;       //accessing private members of the object
}
void main()
{
    A a1;
    abc(a1);
}
```

A few important points about the friend functions:

- Friend keyword should appear in the prototype only and not in the definition.
- Since it is a non-member function of the class of which it is a friend, it can be prototyped in either the private or the public section of the class.
- A friend function takes one extra parameter as compared to a member function that performs the same task. This is because it cannot be called with respect to any object. Instead, the object itself appears as an explicit parameter in the function call.
- We need not and should not use the scope resolution operator while defining a friend function.

b) Friend Classes

A class can be a friend of another class. Member functions of a friend class can access private data members of objects of the class of which it is a friend.

Syntax: *friend class frnd_class_name;*

If class B is to be made a friend of class A, then the statement

friend class B;

should be written within the definition of class.

```
class A
{
    friend class B;          //declaring B as a friend of A
    /* rest of the class A */
};
```

It does not matter whether the statement declaring class B as a friend is mentioned within the private or the public section of class A. Now, member functions of class B can access the private data members of objects of class A.

Example-2: Effect of declaring friend class

```
class B;        /*forward declaration… necessary because definition of class B is after the statement that
                declares class B a friend of class A. */
class A
{
     int x;
  public:
    void setx(const int=0);
    int getx() const;
    friend class B;
};
class B
{
   int y;
  public:
   int gety() const;
   void test_friend();
};
void B::test_friend()
{
   A a;                  // Object of class A
   a.setx(5);
   y=a.x;                // Access to private data 'x' from friend function
}
void A::setx(const int a)
{
  x=a;
}
int B::gety()const
{
  return y;
}
#include <iostream.h>
void main()
{
   B b1;
   b1.test_friend();
```

```
    cout << b1.gety() << endl;
}
```

As we can see, member functions of class B are able to access private data member of objects of the class A although they are not member functions of class A. Friendship is not transitive.

c) Friend Member Functions

How can we make some specific member functions of one class friendly to another class? For making only B::test_friend() function a friend of class A, replace the line

```
    friend class B;
```

in the declaration of the class A with the line:

```
    friend void B::test_friend();
```

Prototype Syntax at Class: *friend ret_type class_name_of_frnd_func::frnd_func_name();*

The modified definition of the class A is:

```
class A
{
    /* rest of the class A */
        :       :       :
    friend void B::test_friend();
};
```

However, in order to compile this code successfully, the compiler should first see the definition of the class B. Otherwise, it does not know that test_friend() is a member function of the class B. This means that we should put the definition of class B before the definition of class A.

However, a pointer of type A* is a private data member of class B. So, the compiler should also know that there is a class A before it compiles the definition of class B. This problem of circular dependence is solved by forward declaration. This is done by inserting the line

```
    class A;                    //Declaration only! Not definition!!
```

before the definition of class B. Now, the declarations and definitions of the two classes appear as follows:

Example-3: Forward defining a class that requires a friend

```
class B
{
    int y;
  public:
    int gety() const;
    void test_friend();
};
class A
{
    int x;
  public:
    void setx(const int=0);
    int getx() const;
```

```
    friend class B;
};
```

d) Friends as Bridges

Friend functions can be used as bridges between two classes. Suppose there are two unrelated classes whose private data members need a simultaneous update through a common function. This function should be declared as a friend to both the classes.

Example-4: Friends as bridges

```
class B;                    //forward declaration

class A
{
        :       :                               /* rest of the class A */
    friend void ab(const A&, const B&);
};

class B
{
        :       :                               /* rest of the class B */
    friend void ab(const A&, const B&);
};
```

7. Static Members

a) Static Member Data

Static data members hold global data that is common to all objects of the class. Examples of such global data are:

- Count of objects currently present
- Common data accessed by all objects, etc.

Static data members are members of the class and not of any object of the class, that is, they are not contained inside any object. We prefix the declaration of a variable within the class definition with the ***keyword static*** to make it a static data member of the class.

Syntax for defining static member: *static data_type member_name;*

Example-1: Declaring a static data member

```
class Account
{
    static float interest_rate;         //a static data member
        :       :       :               // rest of the class Account
};
```

A statement declaring a static data member inside a class will obviously not cause any memory to get allocated for it. Moreover, memory for a static data member will not get allocated when objects of the class are declared. This is because a static data member is not a member of any object.

Therefore, we must not forget to write the statement to define (allocate memory for) a static member variable. Explicitly defining a static data member outside the class is necessary. Otherwise, the linker produces an error.

Syntax for allocating space to static member: *data_type class_name::member_name;*

The following statement allocates memory for interest_rate member of class Account.

float Account::interest_rate;

The above statement initializes interest_rate to zero. If some other initial value (say 4.5) is desired instead, the statement should be rewritten as follows.

float Account::interest_rate=4.5;

Making static data members private prevents any change from non-member functions as only member functions can change the values of static data members. Introducing static data members does not increase the size of objects of the class. Static data members are not contained within objects. There is only one copy of the static data member in the memory.

Example-2: Static data members are not a part of objects

```
#include <iostream.h>
class A
{
    int x; char y; float z;
    static float s;
};
float A::s=1.1;
void main()
{
    cout<<sizeof(A)<<endl;
}
```

Note-1: Static data members can be of any type. Static data members of integral type can be initialized within the class itself if the need arises.

Example-3: Initializing integral static data members within the class itself

```
class Account
{
    static int nameLength=30;
    static char name[nameLength];
        :       :       :           // other members
};
int A::nameLength;
char A::name[nameLength]="The Rich and Poor Bank";
    :   :   :                       // other member definitions
```

Example-4: Non-integral static data members cannot be initialized within the class

```
class Account
{
    static char name[30]="The Rich and Poor Bank";   //error!!
        :       :       :                           // other members
};
```

Note-2: Member functions can refer to static data members directly.

Example-5: Accessing static data members from non-static member functions

```
class Account
{
    static float interest_rate;
public:
    void updateBalance();
        :       :       :               // other members
};

float Account::interest_rate=4.5;

void Account::updateBalance()
{
    if(end_of_year)
        balance+=balance*interest_rate/100;
}
    :   :       :       :               // other member definitions
```

Object to Member Access: The object-to-member access operator (dot) can be used to refer to the static data member of a class with respect to an object.

Syntax: *object_name.static_mem_name;*

The class name with the scope resolution operator can do this directly.

Syntax: *class_name::static_mem_name;*

For example:

```
f=a1.interest_rate;          //a1 is an object of the class Account
f=Account::interest_rate;
```

Example-6: Static data members can be of the same type as their class

```
class A
{
    static A a1;     //OK : static
    A * APtr;        //OK : pointer
    A a2;            //ERROR!! : non-static
};
```

Example-7: A static data member can appear as the default argument in the member functions

```
class A
{
    static int x;
    int y;
  public:
    void abc(int=x);        //OK
    void def(int=y);        //ERROR!! : object required
};
```

Note-3: A static data member can be declared to be a constant. In that case, the member functions will be able to only read it but not modify its value.

b) Static Member Functions

Static member functions access and/or modify static data members of the class. Prefixing the function prototype with the **keyword static** specifies it as a static member function. However, the keyword static should not reappear in the definition of the function.

Syntax: *static ret-type static_mem_fun_name(arguments);*

Example-8: Static member function

```
class Account
{
    static float interest_rate;
  public:
    static void set_interest_rate(float);
        :       :       :                       // other members
};
float Account::interest_rate= 4.5;
void Account::set_interest_rate(float p)
{
    interest_rate=p;
}
```

Static member functions do not take the 'this' pointer as a formal argument. Therefore, accessing non-static data members through a static member function results in compile-time errors. Static member functions can access only static data members of the class. Static member functions can still be called with respect to objects.

```
    a1.set_interest_rate(5);      //a1 is an object of the class Account
```

8. Objects and Functions

Objects can appear as local variables inside functions. They can also be passed by value or by reference to functions. Finally, they can be returned by value or by reference from functions.

Example-1: Returning class objects

```
class Distance
{
  public:
    Distance add(Distance);
    :       :       :               // other members
};
Distance Distance::add(Distancedd)
{
    Distance temp;
    temp.iFeet=iFeet+dd.iFeet;
    temp.setInches(fInches+dd.fInches);
    return temp;
}
 :      :       :               // other member definitions
```

```
#include<iostream.h>
void main()
{
    Distance d1,d2,d3;
    d1.setFeet(5); d1.setInches(7.5);
    d2.setFeet(3); d2.setInches(6.25);
    d3=d1.add(d2);
    cout<<d3.getFeet()<<""<<d3.getInches()<<endl;
}
```

Example-2: Returning class objects by reference

```
class Distance
{
    :       :       :                   // other members
    Distance& larger(Distance&, Distance&);
};

Distance& larger(Distance& dd1, Distance& dd2)
{
    if(dd1.getFeet()>dd2.getFeet)
        return dd1;
    else
        return dd2;
}
    :       :       :           // other member definitions

#include<iostream.h>
void main()
{
    Distance d1,d2;
    d1.setFeet(5); d1.setInches(7.5);
    d2.setFeet(5); d2.setInches(6.25);
    Distance& d3=larger(d1,d2);
    d3.setFeet(0); d3.setInches(0.0);
    cout<<d1.getFeet()<<»«<<d1.getInches()<<endl;
    cout<<d2.getFeet()<<»«<<d2.getInches()<<endl;
}
```

9. Objects and Arrays

a) Arrays of Objects: We can create arrays of objects.

Example-1: Array of objects

```
#include<iostream.h>
#define SIZE 3

void main()
{
    Distance dArray[SIZE];
    int a;
    float b;
    for(int i=0;i<SIZE;i++)
    {
```

```
        cout<<"Enter the feet : "; cin>>a; dArray[i].setFeet(a);
        cout<<"Enter the inches : "; cin>>b; dArray[i].setInches(b);
    }
    for(int i=0;i<SIZE;i++)
        cout<<dArray[i].getFeet()<<""<<dArray[i].getInches()<<endl;
}
```

b) Arrays Inside Objects

An array can be declared inside a class. Such an array becomes a member of all objects of the class. It can be manipulated/accessed by all member functions of the class.

Example-2: Arrays inside objects

```
class A
{
    int iArray[SIZE];
  public:
    void setElement(unsigned int, int);
    int getElement(unsigned int);
};
void A::setElement(unsigned int p, int v)
{
    if(p>=SIZE)
        return;        //better to throw an exception
    iArray[p]=v;
}
int A::getElement(unsigned int p)
{
    if(p>=SIZE)
        return –1;      //better to throw an exception
    return iArray[p];
}
```

10. Namespaces

Namespaces enable the C++ programmer to prevent pollution of the global namespace that leads to name clashes. The term 'global namespace' refers to the entire source code which also includes all the directly and indirectly included header files.

a) Need of Namespaces in Programs

Suppose a class with the same name is defined in two header files.

```
/*Beginning of A1.h*/
class A
{
};
/*End of A1.h*/
/*Beginning of A2.h*/
class A//a class with an existing name
{
};
```

```
/*End of A2.h*/
```

Now, let us include both these header files in a program and see what happens if we declare an object of the class.

```
#include "A1.h"
#include "A2.h"
void main()
{
    A AObj;              //ERROR: Ambiguity error due to multiple definitions of A
    :   :
}
```

The simultaneous use of both classes in a program is facilitated by enclosing the two definitions of the class in separate namespaces.

b) Defining a Namespace

Syntax:
```
namespace namespace_name
    {
        // class definitions
    }
```

Example-1: Enclosing classes in namespaces prevents pollution of the global namespace

```
/*Beginning of A1.h*/
namespace A1              //beginning of a namespace A1
{
    class A
    {
    };
}                         //end of a namespace A1
/*End of A1.h*/

/*Beginning of A2.h*/
namespace A2              //beginning of a namespace A2
{
    class A
    {
    };
}                         //end of a namespace A2
/*End of A2.h*/
```

Now, the two definitions of the class are enveloped in two different namespaces. The corresponding namespace, followed by the scope resolution operator, must be prefixed to the name of the class while referring to it anywhere in the source code. Thus, the ambiguity encountered in the above listing can be overcome.

Syntax of referring a namespace: *namespace_name::class_name Object_name*

```
#include "A1.h"
#include "A2.h"
void main()
{
    A1::A AObj1;          //OK: AObj1 is an object of the class defined in A1.h
```

```
    A2::A AObj2;            //OK: AObj2 is an object of the class defined in A2.h
}
```

c) The 'using' directive

Qualifying the name of the class with that of the namespace can be cumbersome. The using directive enables us to make the class definition inside a namespace visible so that qualifying the name of the referred class by the name of the namespace is no longer required.

Syntax: *using namespace namespace_name;*

Example-2: The using directive makes qualifying of referred class names by names of enclosing namespaces unnecessary.

```
#include "A1.h"
#include "A2.h"
void main()
{
    using namespace A1;
    A AObj1;                //OK: AObj1 is an object of the class defined in A1.h
    A2::A AObj2;            //OK: AObj2 is an object of the class defined in A2.h
}
```

d) Namespace Aliases

Some namespaces have long names. Qualifying the name of a class that is enclosed within such a namespace, with the name of the namespace, is cumbersome.

```
namespace a_very_very_long_name
{
    class A
    {
    };
}
void main()
{
    a_very_very_long_name::AA1;   //cumbersome long name
}
```

Assigning a suitably short alias to such a long namespace name solves the problem.

Syntax: *namespace alias_name=namespace-name;*

Example-3: Providing an alias for a namespace

```
namespace a_very_very_long_name
{
    class A
    {
    };
}
namespace x = a_very_very_long_name;     //declaring an alias
void main()
{
    x::A A1;            //convenient short name
```

```
}
```

Note: Suppose an alias has been used at a number of places in the source code. Changing the alias declaration so that it stands as an alias for a different namespace will make each reference of the enclosed class refer to a completely different class.

Suppose an alias X refers to a namespace 'N1'.

```
namespace X = N1;          //declaring an alias
```

Further, suppose that this alias has been used extensively in the source code. If the declaration of alias X is modified as follows ('N2' is also a namespace)

```
namespace X = N2;          //modifying the alias
```

then, all existing qualifications of referred class names that use X would now refer to class A that is contained in namespace 'N2'. Of course, the lines having such references would compile only if both of the namespaces, 'N1' and 'N2', contain a class named A, and if these two classes have the same interface.

11. Nested Classes

A class can be defined inside another class. Such a class is known as a nested class (inner class). The class that contains the nested class is known as the enclosing (outer) class. Nested classes can be defined in the private, protected, or public portions of the enclosing class.

Example-1: Nested classes (private and public)

```
class A
{
    class B
    {
        /* definition of class B */
    };
    /* definition of class A */
};
```

```
class A
{
  public:
    class B
    {
        /* definition of class B*/
    };
    /* definition of class A */
};
```

Need: A nested class is created if it does not have any relevance outside its enclosing class. By defining the class as a nested class, we avoid a name collision. The size of objects of an enclosing class is not affected by the presence of nested classes.

Example-2: Size of objects of the enclosing class

```
class A
{
    int x;
  public:
    class B
    {
        int y;
    };
};
void main()
{
```

```
    cout<<sizeof(int)<<endl;
    cout<<sizeof(A)<<endl;
}
```

a) Defining Member Functions of a Nested Class

Member functions of a nested class can be defined outside the definition of the enclosing class. This is done by prefixing the function name with the name of the enclosing class followed by the scope resolution operator. This, in turn, is followed by the name of the nested class followed again by the scope resolution operator.

Syntax: *ret_type Enclosing_class : : Nested_class : : Nested_member_fun(args)*

```
        {
                // function definition
        }
```

Example-3: Defining member functions of nested classes

```
class A
{
  public:
    class B
    {
       public:
          void BTest();         //prototype only
    };
    /* definition of class A */
};
void A::B::BTest()
{
    //definition of A::B::BTest() function
}
:     :      :   // other member definitions
```

b) Defining Nested Class Outside the Enclosing Class

A nested class may be only prototyped within its enclosing class and defined later. Again, the name of the enclosing class followed by the scope resolution operator is required.

Syntax: *class enlosing_class : : nested_class*

```
        {
                // inner class definitions
        };
```

Example-4: Defining a nested class outside the enclosing class

```
class A
{
    class B;          //prototype only
};
class A::B
{
    /* definition of the class B */
```

```
};
```

c) Creating Objects of a Nested Class

Objects of the nested class are defined outside the member functions of the enclosing class in much the same way (by using the name of the enclosing class followed by the scope resolution operator).

Syntax: *Enclosing_class : : Nested_class Inner_Object;*

Example: *A::B B1;*

However, the above line will compile only if class B is defined within the public section of class A. Otherwise, a compile-time error will result.

d) Declaring Objects of a Nested Class in the member Functions of Enclosing Class

An object of the nested class can be used in any of the member functions of the enclosing class without the scope resolution operator. Moreover, an object of the nested class can be a member of the enclosing class. In either case, only the public members of the object can be accessed unless the enclosing class is a friend of the nested class.

Example-5: Declaring objects of the nested class in the member functions of the enclosing class

```
class A
{
    class B
    {
      public:
        void BTest();    //prototype only
    };
    B B1;                // Nested class object
  public:
    void ATest();
};
void A::ATest()
{
    B1.BTest();
    B B2;                //Nested class object inside member function of enclosing class
    B2.BTest();
}
```

Note: Member functions of the nested class can access the non-static public members of the enclosing class through an object, a pointer, or a reference only.

Example-5: Accessing non-static members of the enclosing class in member functions of the nested class.

```
class A
{
  public:
    void ATest();
    class B
    {
        public:
          void BTest(A&);
```

```
        void BTest1();
    };
};

void A::B::BTest(A& ARef)
{
    ARef.ATest();           //OK
}

void A::B::BTest1()
{
    ATest();                //ERROR!!
}
```

It can be observed that an error is produced when a direct access is made to a member of the enclosing class through a function of the nested class. After all, creation of an object of the nested class does not cause an object of the enclosing class to be created. By default, the enclosing class and the nested class do not have any access rights to each other's private data members. They can do so only if they are friends to each other.

Section-III

A. Constructors

The constructor gets called automatically for each object that has just got created. It appears as member function of each class, whether it is defined or not. It has the same name as that of the class. It may or may not take parameters. It does not return anything (not even void). The prototype of a constructor is

Prototype Syntax: *class name (parameter list);*

Constructors guarantees initialization of member data of a class. Domain constraints on the values of data members can also be implemented via constructors. The compiler embeds a call to the constructor for each object when it is created.

Example-1: Constructor gets called automatically for each object when it is created

```
class A
{
    int x;
  public:
    void setx(const int=0);
    int getx();
};

void main()
{
    A A1;               //object declared … constructor called
}
```

The statement in the function main() in Example-1 is transformed into the following statements.

```
A A1;       //memory allocated for the object (4 bytes)
A1.A();     //constructor called implicitly by compiler
```

Similarly, the constructor is called for each object that is created dynamically in the heap by the ***new operator.***

```
A * APtr;
APtr = new A;       //constructor called implicitly by compiler
```

Note-1: Unlike their name, constructors do not actually allocate memory for objects. They are member functions that are called for each object immediately after memory has been allocated for the object.

1. Zero-argument Constructor

We can and should define our own constructors if the need arises. If we do so, the compiler does not define the constructor. The constructor is a non-static member function which is called for an object. It, therefore, takes the 'this' pointer as a leading formal argument just like other non-static member functions. This means that the members of the invoking object can be accessed from within the definition of the constructor.

Example-1: Constructor gets called automatically for each object when it is created

```
class A
{
    int x;
  public:
    A();                    //custom constructor prototype
    void setx(const int=0);
    int getx();
};

A::A()                      //custom constructor definition
{
    cout<<"Constructor of class A called\n";
}
/* definitions of the rest of the functions of class A */

#include<iostream.h>
void main()
{
    A A1;
    cout<<"End of program\n";
}
```

Example-2: A user-defined constructor to implement domain constraints on the data members of a class

```
class Distance
{
  public:
    Distance();             //our own constructor
    /* rest of the class Distance */
};

Distance::Distance()        //our own constructor
{
    iFeet=0;
    fInches=0.0;
```

```
}
/* definitions of the rest of the functions of class Distance */

#include<iostream.h>
void main()
{
    Distance d1;            //constructor called
    cout<<d1.getFeet()<<""<<d1.getInches();
}
```

Now, due to the presence of the constructor within the class Distance, there is guaranteed initialization of the data of all objects of the class Distance.

The constructor that does not take any arguments and is called the ***zero-argument constructor***. The constructor provided by default by the compiler also does not take any arguments. Therefore, the terms ***'zero-argument constructor' and 'default constructor'*** are used interchangeably.

Example-3: A STRING class

Let us call the class String. It will have two data members. Both these data members will be private. The first data member will be a character pointer. It will point at a dynamically allocated block of memory that contains the actual character array. The other data member will be a long unsigned integer that will contain the length of this character array.

```
class String
{
    char * cStr;
    long unsigned int len;
  public:
    String();           //prototype of the constructor
    /* rest of the class String */
};

String::String()        //definition of the constructor When an object is created …
{
    cStr=NULL;          //…nullify its pointer and…
    len=0;              //…set the length as zero.
}
```

2. Parameterized Constructor

Constructors take arguments and can, therefore, be overloaded. Suppose, for the class Distance, the library programmer decides that while creating an object, the application programmer should be able to pass some initial values for the data members contained in the object.

Example-1: A user-defined parameterized constructor—called by creating an object in the stack

```
class Distance
{
  public:
    Distance();             // Zero argument constructor
    Distance(int,float);    //prototype of the parameterized constructor
    /* rest of the class Distance */
};
```

```
Distance::Distance()
{
    iFeet=0;
    fInches=0.0;
}

Distance::Distance(int p, float q)
{
    iFeet=p;
    setInches(q);
}

/* definitions of the rest of the functions of class Distance */
#include<iostream.h>
void main()
{
    Distance d1(1,1.1);      //parameterized constructor called
    cout<<d1.getFeet()<<""<<d1.getInches();
}
```

Example-2: A user-defined parameterized constructor—called by creating an object in the heap

```
#include<iostream.h>
void main()
{
    Distance * dPtr;
    dPtr = new Distance(1,1.1);        // parameterized constructor called Output
    cout<<dPtr->getFeet()<<""<<dPtr->getInches();
}
```

Note-1: We must remember that, if the parameterized constructor is provided and the zero-argument constructor is not provided, the compiler will not provide the default constructor. In such a case, the following statement will not compile.

```
Distance d1;              //ERROR: No matching constructor
```

Note-2: Just like in other member functions, the formal arguments of the parameterized constructor can be assigned default values. But in that case, the zero-argument constructor should be provided. Otherwise, an ambiguity error will arise when we attempt to create an object without passing any values for the constructor.

```
class Distance
{
  public:
    //Distance();       zero-argument constructor commented out default values given
    Distance(int=0,float=0.0);
    /* rest of the class Distance */
};
```

If we write, *Distance d1;* an ambiguity error arises if the zero-argument constructor is also defined. This is because both the zero-argument constructor and the parameterized constructor can resolve this statement.

Example-2: Parameterized Constructor for Strings

Let us now create a parameterized constructor for the class String. We will also assign a default value for the argument of the parameterized constructor. The constructor would handle the following statements.

```
String s1("abc");
    OR
char * cPtr = "abc";
String s1(cPtr);
    OR
char cArr[10] = "abc";
String s1(cArr);
```

In each of these statements, we are essentially passing the base address of the memory block in which the string itself is stored to the constructor.

Let us now define the constructor that produces these effects. We must realize that 'p' (the formal argument of the constructor) should be as follows:

```
    const char * const
```

First, it should be a constant pointer because throughout the execution of the constructor, it should continue to point at the same memory block. Second, it should be a pointer to a constant char because even in advertently, the library programmer should not dereference it to change the contents of the memory block at which it is pointing. Additionally, we would like to specify a default value for 'p' (NULL), so that there is no need to separately define a zero-argument constructor.

```
#include<string.h>
#include<iostream.h>
class String
{
    char * cStr;
    long unsigned int len;
  public:
    String(const char * const p = NULL);       //no zero-argument constructor
    const char * getString();
    /* rest of the class String */
};
String::String(const char * const p)
{
    if(p==NULL)         //if default value passed…
    {
        cStr=NULL;    //…nullify
        len=0;
    }
    else          //…otherwise…
    {
        len=strlen(p);
        cStr=new char[len+1];      //…dynamically allocate a separate memory block
        strcpy(cStr,p);              //…and copy into it
    }
}
```

```
const char * String::getString()
{
    return cStr;
}
/* definitions of the rest of the functions of class String */

void main()
{
    String s1("abc");              //pass a string to the parameterized constructor
    cout<<s1.getString()<<endl;    //display the string
}
```

3. Copy Constructor

The copy constructor is a special type of parameterized constructor. As its name implies, it copies one object to another. It is called when an object is created and equated to an existing object at the same time. The copy constructor is called for the object being created. The pre-existing object is passed as a parameter to it. The copy constructor member-wise copies the object passed as a parameter to it into the object for which it is called.

If we do not define the copy constructor for a class, the compiler defines it for us. But in either case, a call is embedded to it under the following three circumstances.

- When an object is created and simultaneously equated to another existing object, the copy constructor is called for the object being created. The object to which this object was equated is passed as a parameter to the copy constructor.

  ```
  A A1;          //zero-argument/default constructor called
  A A2=A1;       //copy constructor called
     or
  A A2(A1);      //copy constructor called
     or
  A * APtr = new A(A1);        //copy constructor called
  ```

  Here, the copy constructor is called for 'A2'and for 'Aptr' while 'A1' is passed as a parameter to the copy constructor in both cases.

- When an object is created as a non-reference formal argument of a function. The copy constructor is called for the argument object. The object passed as a parameter to the function is passed as a parameter to the copy constructor.

  ```
  void abc(A);
  A A1;                  //zero-argument/default constructor called
  abc(A1);               //copy constructor called
  void abc(A A2)
  {
      /* definition of abc() */
  }
  ```

  Here again the copy constructor is called for 'A2' while 'A1' is passed as a parameter to the copy constructor.

When an object is created and simultaneously equated to a call to a function that returns an object. The copy constructor is called for the object that is equated to the function call. The object returned from the function is passed as a parameter to the constructor.

```
A abc()
{
    A A1;           //zero-argument/default constructor called
    /* remaining definition of abc() */
    return A1;
}
A A2=abc();         //copy constructor called
```

Once more, the copy constructor is called for 'A2' while 'A1' is passed as a parameter to the copy constructor.

Note: The default copy constructor does exactly what it is supposed to do—it copies. The statement

```
A A2=A1;
```

is converted as follows:

```
A A2;         //memory allocated for A2
A2.A(A1);     //copy constructor is called for A2 and A1 is passed as a parameter to it
```

Example-1: Copy Constructor for Strings

```
#include<iostream.h>
#include<string.h>
class String
{
    char * cStr;
    long unsigned int len;
  public:
    String(const String&);       //our own copy constructor
    /* rest of the class String */
    explicit String(const char * const p = NULL);
    const char * getString();
};
String::String(const String& ss) //our own copy constructor
{
    if(ss.cStr==NULL)            //if passed object's pointer is NULL…
    {
        cStr=NULL;               //… then nullify the invoking object's pointer too
        len=0;
    }
    else                         //otherwise…
    {
        len=ss.len;
        cStr = new char[len+1];    //…dynamically allocate a separate memory block
        strcpy(cStr,ss.cStr);      //…and copy into it
    }
}

String::String(const char * const p)
{
```

```
    if(p==NULL)                    //if default value passed…
    {
        cStr=NULL;                 //…nullify len=0;
    }
    else                           //…otherwise…
    {
        len=strlen(p);
        cStr=new char[len+1];      //…dynamically allocate a separate memory block
        strcpy(cStr,p);            //…and copy into it
    }
}
const char * String::getString()
{
    return cStr;
}
void main()
{
    String s1("abc");
    String s2=s1;
    cout<<s1.getString()<<endl;
    cout<<s2.getString()<<endl;
}
```

B. Destructors

The destructor gets called for each object that is about to go out of scope. It appears as a member function of each class whether we define it or not. It has the same name as that of the class but prefixed with a tilde sign. It does not take parameters. It does not return anything (not even void). The prototype (syntax) of a destructor is:

~ <class name> ();

Need: The need for a function that guarantees deinitialization of member data of a class and frees up the resources acquired by the object during its life time (also termed as clean up). Destructors fulfill this need. For example, closing the files that are opened or disconnecting the database etc.

The destructor will also be called for an object that has been dynamically created in the heap just before the **delete operator** is applied on the pointer pointing at it.

```
A * APtr;
APtr = new A;          //object created … constructor called
. . . .
. . . .
delete APtr;           //object destroyed … destructor called
```

The last statement is transformed into

```
APtr->~A();            //destructor called for *APtr
delete APtr;           //memory for *APtr released
```

First, the destructor is called for the object that is going out of scope. Thereafter, the memory occupied by the object itself is deallocated. The second last statement above is transformed into *~A(APtr);*

Unlike its name, the destructor does not 'destroy' or deallocate memory that an object occupies. It is merely a member function that is called for each object just before the object goes out of scope (gets destroyed).

Example-1: Destructor gets called for each object when the object is destroyed

```
#include<iostream.h>
class A
{
    int x;
  public: A();
    void setx(const int=0);
    int getx();
    ~A();                  //our own destructor
};
A::A()
{
    cout<<"Constructor of class A called\n";
}
A::~A()                    //our own destructor
{
    cout<<"Destructor of class A called\n";
}
/*definitions of the rest of the functions of class A */
void main()
{
    A A1;
    cout<<"End of program\n";
}
```

Review Questions

1. Explain the need of structures in programming with a suitable example. Write the steps in creating a new data type using structures.
2. Bring out the salient features of procedure oriented programming and object oriented programming systems.
3. Explain the C++ tokens/constructs used for console Input and Output with examples.
4. What are function prototypes? Write the syntax of a function prototype and explain the importance of function prototypes in C++.
5. What is Reference Variable? Explain with an example, how reference variables are passed and returned from functions?
6. Explain function overloading with a suitable example in C++.
7. How structures in C++ are different from structures in C?
8. Write the structure of a C++ class. How class is different from structure in C++?
9. Explain with an example, the need of a scope resolution operator in C++. Write the steps in creating libraries using scope resolution operator.
10. Explain the role of 'this' pointer with an example.

11. Define the term 'Data Abstraction' by taking suitable example.
12. Explain with an example:
    (i) Arrow operator (iii) Inline member functions (iii) Constant member functions
    (ii) Mutable member functions (iv) Default values for formal arguments of member functions
13. Explain the role of friend classes and friend functions in C++ with suitable examples.
14. Explain the need, definition and accessing static data members of a class with examples.
15. Explain with an example, static member functions.
16. Explain passing and retuning of objects to member functions.
17. Explain, how array of objects and object contain an array is defined in C++ with examples to each.
18. Explain the need and defining a namespaces in building applications in C++.
19. Explain with an example:
    (i) 'using' directive (ii) Namespace aliases (ii) Destructors and its need
20. Explain the role of nested classes in program development. Explain with example the ways of defining nested classes and their member functions.
21. With syntax and an example, explain the role constructors in programming.
22. With suitable syntax and examples, explain THREE types of constructors.
23. Design a class "COMPLEX" with the following members:
    - Constructor for complex number initialization, print complex number, read a complex number, add two complex numbers (TWO methods – one member with implicit object and other member with TWO explicit object parameters).

    Write a main function to create few objects of class COMPLEX and add TWO complex numbers.

****