```
#include <iostream>
using namespace std;
Notes EC_5_SEM_D_SEC( )
{
        cout <<" MODULE-2 "<<"\n"
        <<" FUNCTIONS,
 CLASSES AND OBJECTS
        "<<"\n"
        << " - GANESH Y "<< "\n"
        << " Dept. of ECE RNSIT ";
return Assignments;
}
```

MODULE -2

FUNCTIONS, CLASSES AND OBJECTS

GANESH Y
Dept. of ECE RNSIT

MODULE -2
Functions, Classes and Objects

SYLLABUS

Functions, classes and Objects: Functions, Inline function, function overloading, friend and virtual functions, Specifying a class, C++ program with a class, arrays within a class, memory allocation to objects, array of objects, members, pointers to members and member functions (Selected Topics from Chap-4,5 of Text1).

Introduction

We know that functions play an important role in C program development. Dividing a program into functions is one of the major principles of top down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

Recall that we have used a syntax similar to the following in developing C programs.

```
void show();  /* Function declaration */
main ()
{
     ......
     show();       /* Function call */
     ......
}
void show()        /* Function definition */
{
     .......
     .......     //Function body
}
```

When the function is called control is transferred to the first statement in function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered.

C++ is no exception. Functions continue to be the building blocks of C++ programs. In fact, C++ bas added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ function can be overloaded to make it perform different tasks depending on the arguments passed to it. Most of these modifications are aimed at meeting the requirements of object oriented facilities.

The Main Function

C does not specify any return type for the `main()` function which is the starting point for the execution of a program. The definition of **`main()`** would look like this:

```
main ()
{
        //main program statements
}
```

This is perfectly valid because the `main()` in C does not return any value. In C++, the `main()` returns a value of type `int` to the operating system. C++, therefore, explicitly defines `main()` as matching one of the following prototypes:

```
int main();
int main(int argc, char* argv[]);
```

The functions that have a return value should use the return statement for termination, The **`main()`** function in C++ is, therefore, defined as follows:

```
int main ()
{
    ..........
    ..........
    return 0;
}
```

Since the return type of functions is `int` by default. the keyword `int` in the `main()` header is optional. Most C++ compilers will generate an error or warning if here is no **`return`** statement.

Many operating systems test the return value (called *exit value)* to determine if there is any problem. The normal convention *is* that an exit value of zero means the program ran successfully. while a nonzero value means there was a problem. The explicit use of a **`return(0)`** statement will indicate that the program was successfully executed.

Function Prototyping

Function ***prototyping*** is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. With function prototyping, a *template* is always used when declaring and defining a function.

When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time

of compilation itself. These checks and controls did not exist in the conventional C functions.

Remember, C also uses prototyping. But it was introduced first in C++ and the success of this feature inspired the ANSI C committee to adopt it.

However, there is a major difference in prototyping between C and C++. While C++ makes the prototyping essential, ANSI C makes it optional, perhaps, to preserve the compatibility with classic C.

Function prototype is a declaration statement in the calling program and is of the following form:

```
type function-name (argument-list);
```

The *argument-list* contains the types and names of arguments that must be passed to the function.

Example:

```
float volume(int x, float y, float z);
```

Note that each argument variable must be declared independently inside the parenthesis. That is, a. combined declaration like

```
float volume(int x, float y,z);
```

is illegal.

In a function declaration, the names of the arguments are *dummy* variables and therefore they are optional. That is, the form

```
float volume(int , float, float);
```

is acceptable at the place of declaration. At this stage. the compiler only checks for the type of arguments when the function is called.

In general, we can either include or exclude the variable names in the argument list of prototypes. The variable names in the prototype just act as placeholders and, therefore, if names are used, they don't have to match the names used in the function call or function definition.

In the function definition, names are required because the arguments must be referenced inside the function. Example:

```
float volume(float a, float b, float c)
{
     float v=a*b*c;
     ..............
     ..............
}
```

The function volume() can be invoked in a program as follows:

```
float cube1= volume(b1,w1,h1); // Function call
```

The variable b1, w1, and h1 are known as the actual parameters which specify the dimensions of cube1. Their types (which have been declared earlier) should match with the types declared in the prototype. Remember, the calling statement should not include type names in the-argument list.

We can also declare a function with an *empty argument list,* as in the following example:

```
void display( );
```

Which is similar to

```
void display(void);
```

However, in C, an empty parenthesis implies any number of arguments. That is, we have foregone prototyping. A C++ function can also have an 'open' parameter list by the use of ellipses in the prototype as shown below:

```
void do_something(...);
```

The general form of a function is

```
ret-type function-name (parameter list)
{
    body of the function
}
```

The parameter declaration list for a function takes this general form:

```
f(type varname1, type varname2, . . . , type varnameN)
```

```
f(int i, int k, int j) /* correct */
f(int i, k, float j) /* incorrect */
```

Call by Value

In traditional C, a function call passes arguments by value. The ***called function*** creates a new set of variables and copies the values of arguments into them. The function does not have access to the actual variables in the calling program and can only work on the copies of values.

```
#include <stdio.h>
int sqr(int x )/* formal parameters */
{
    x = x*x;
    return(x);
}
int main(void)
{
```

```
        int t=10,a;
        a=sqr(t);  /* actual parameters */
        return 0;
    }
```

Call by Reference

Provision of the *reference variables* in C++ permits us to pass parameters to the functions by reference. When we pass arguments by reference. the formal arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data.

```
swap (int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
int main()
{
    int i, j;
    i = 10;
    j = 20;
    swap(i, j);
    return 0;
}
// C style Call by reference using pointers
swap (int *x, int *y)
{
    int temp;
    temp = *x; /* save the value at address x */
    *x = *y;   /* put y into x */
    *y = temp; /* put x into y */
}
int main()
{
    int i, j;
    i = 10;
    j = 20;
    swap(&i, &j); /* pass the addresses of i and j */
    return 0;
}
```

This approach is also acceptable in C++. Note that the call-by-reference method is neater in its approach.

Return by reference

A function can also return a reference. Consider the following function:

```
int & max(int &x, int &y)
{
     if (x > y)
          return x;
     else
          return y;
}
int main()
{
 int m=10,n=8,p;
 p=max(m,n); // p=m=10
 max(m,n)=-1; // returned variable=m=-1
 return 0;
}
```

Since the return type of `max()` is `int &`, the function returns reference to x or y (and not the values). Then a function call such as `max(m,n)` will yield a reference to either m or n depending on their values.

This means that this function call can appear on the left-hand side of an assignment statement as `max(m,n)=-1;`

Inline Functions

One of the objectives of using functions in a program is to save some memory space. Which becomes appreciable when a function is likely to be called many times.

However, every time a function it; called, it takes a lot of extra time in executing a series of instructions for tasks such as *jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function.*

When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use ***macro definitions***, popularly known as ***macros.*** Preprocessor macros are popular in C.

The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called ***inline function***.

An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code (something similar to macros expansion).

The inline functions are defined as follows:

```
inline function-header
{
     function body
}
```

For example

```
inline double cube(double a)
{
     return(a*a*a);
}
```

The above inline function can be invoked by statements like

```
c = cube(3.0);
d = cube(2.5+1.5);
```

If the arguments are expressions such as 2.5 + 1.5, the function passes the value of the expression, 4 in this case. This makes the inline feature far superior to macros.

We should exercise care before making a function inline. The speed benefits of inline functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of inline functions may be lost. **In such cases, the use of normal functions will be more meaningful.**

Usually the functions are made inline when they are small enough to be defined in one or two lines. Example:

```
inline double cube(double a){return(a*a*a);}
```

All inline functions must be defined before they are called.

Remember that the inline keyword merely sends a request, not a command, to the Compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a `switch`, or a `goto` exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain `static` variables.
4. If inline functions are recursive.

Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. However, it makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called. So a trade-off becomes necessary.

```
#include <iostream>
using namespace std;
inline float Mul(float x, floaty)
{
     return (x*y);
}
inline double Div(double p, double q)
{
     return(p/q);
}
int main()
{
     float a =12.345;
     float b = 9.82;
     cout << Mul(a.b) << "\n";
     cout << Div(a.b) << "\n";
     return 0;
}
```

output
121.228
l.25713

Default Arguments

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a ***default value*** to the parameter which does not have a matching argument in the function call.

Default values are specified when the function *is* declared. The compiler looks at the prototype(declaration) to see how many arguments a function uses and alerts the program for possible default values.

```
float amount( float principal , int period, float rate=0.15) ;
```

The default value is specified in a manner syntactically similar to a variable initialization.

The above prototype declares a default value of 0.15 to the argument rate. A subsequent function call like

```
value=amount(5000,7) ; // one argument missing
```

the function use default value of 0.15 for rate. The call

```
value=amount(5000,5,0.12); // no missing argument
```

passes an explicit value of 0.12 to rate

One important point to note is that only the trailing arguments can have default values and therefore we must add defaults from ***right to left***. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
int mul(int i,int j=5,int k=10);      //legal
int mul(int i=5,int j);               //illegal
int mul(int i=0,int j,int k=10);      //illegal
int mul(int i=2,int j=5,int k=10);    //legal
```

Example:

```
#include <iostream>
using namespace std;
void repchar(char='*', int=45); //declaration with
int main()
{
     repchar(); //prints 45 asterisks
     repchar('='); //prints 45 equal signs
     repchar('+', 30); //prints 30 plus signs
     return 0;
}
// displays line of characters
void repchar(char ch, int n) //defaults supplied
{
     for(int j=0; j<n; j++) //loops n times
     cout << ch; //prints ch
     cout << endl;
}
```

Advantages of providing the default arguments are:

1. We can use default arguments to add new parameters to the existing functions.

2. Default arguments can be used to combine similar functions into one.

Example 2:

```
#include<iostream>
using namespace std;
float value(float p, int n, float r=0.15) //prototype + defn
{
     int year = 1;
     float sum = p;
     while (year <= n)
```

```
    {
        sum=sum*(1+r);
        year = year+1;
    }
    return (sum);
}
void printline(char ch='*', int len=40) //prototyp + defn
{
    for(int i=1; i<+len; i++) cout<<ch;
    cout<<"\n";
}
int main()
{
    float amount;
    printline(); //uses default values for arguments
    amount =value(5000.00,5); //default for 3rd argument
    cout<<"\n"<<"final value"<<amount<<"\n";
    printline('='); //default for 2nd  argument
    return 0;
}
```

const Arguments

In C++, an argument to a function can be declared as const as shown below.

```
int strlen(const char *p) ;
int length(const string &s);
```

The qualifier const tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

Recursion

Recursion is a situation where a function calls itself meaning, one of the statements in the function definition makes a call to the same function in which it is present.

It may sound like an infinite looping condition but just as a loop has a conditional check to take the program control out of the loop, recursive function also possesses a base case which returns the program from the current instance of the function to call back to the calling function.

Example 1:

```
//Calculating Factorial of a Number
#include <iostream>
#include <conio.h>
```

```
using namespace std;
long fact (int n)
{
     if(n==0)   //base case
          return 1;
     return (n* fact(n-1)); // recursive function call
}
 int main()
 {
     int num;
     cout<<"Enter a positive integer: ";
     cin>>num;
     cout<<"Factorial of "<<num<< "is"<<fact(num);
     getch();
     return 0;
 }
```

Example 2:

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.

Approach :

Take an example for 2 disks :

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1 : Shift first disk from 'A' to 'B'.

Step 2 : Shift second disk from 'A' to 'C'.

Step 3 : Shift first disk from 'B' to 'C'.

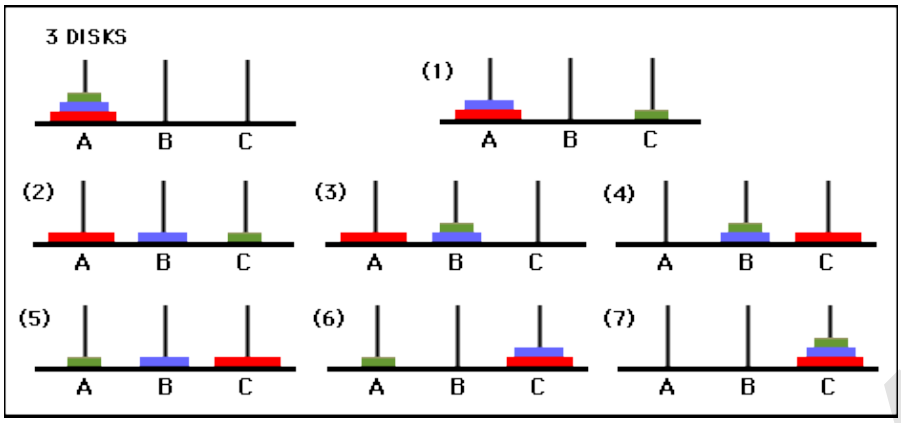
The pattern here is :

Shift 'n-1' disks from 'A' to 'B'.

Shift last disk from 'A' to 'C'.

Shift 'n-1' disks from 'B' to 'C'.

Image illustration for 3 disks:




```
#include <iostream>
#include <conio.h>
using namespace std;
void TOH(int d, char tower1, char tower2, char tower3)
{
     if(d==1)   //base case
     {
        cout<<"\n Shift top disk from tower "<<tower1<<" to tower
"<<tower2;
     return;
     }
     TOH(d-1,tower1,tower3,tower2); // recursive function call
     cout<<"\n Shift top disk from tower "<<tower1<<" to tower "<<
tower2;
     TOH(d-1,tower1,tower3,tower2); // recursive function call
}
 int main()
 {
     int disk;
     cout<<"Enter the no of disks: ";
     cin>>disk;
     if (disk<1)
        cout<<"\nThere are no disks to shift";
     else
     cout<<"\nThere are "<<disk<<"disks in tower1\n";
     TOH(disk,'1','2','3');
     cout <<"\n\n"<<disk<<" disks in tower 1 are shifted to tower
2";
     getch();
     return 0;}
```

Function Overloading

As stated earlier, *overloading* refers to the use of the same thing for different purposes.

C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as *function polymorphism* in OOP.

Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

For example, an overloaded add() function handles different types of data as shown below:

```
//declarations
int add(int a, int b);                    //prototype 1
int add(int a, int b, int c);             //prototype 2
double add(double x, double y);           //prototype 3
double add(int p, double q);              //prototype 4
double add(double p, int q);              //prototype 5

// Function calls
cout << add(5, 10);          //uses prototype 1
cout << add(15, 10.0);       //uses prototype 4
cout << add(12.5, 7.5);      //uses prototype 3
cout << add(5, 10, 15); //uses prototype 2
cout << add(0.75, 5) ;       //uses prototype 5
```

Example:

```
#include<stdlib.h>
#include<iostream>
using namespace std;
int square(int x)
{
     return x*x;
}
float square(float x)
{
     return x*x;
}
int main()
{
```

```
    cout<<square(5)<<endl;
    cout<<square(2.5f);
    return 0;
}
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.

2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

```
char to int
float to double
```

to find a match

3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

```
long square(long n)
double square(double)
```

A function call such as

```
square(10)
```

will cause an error because int argument can be converted to either long or double, thereby creating an ambiguous situation as to which version of square() should be used.

4. If all of the steps fail, then the compiler will try the user defined conversions in combination with integral promotions and built-in conversions to find a unique match. User defined conversions are often used in handling class objects.

Example 2:

```
//function volume() is overloaded three times
#include <iostream>
using namespace std;
// declarations (prototypes)
int volume (int);
double volume (double, int);
long volume (long, int, int);
int main()
{
     cout<< volume(10) <<"\n";
     cout<< volume(2.5,8) <<"\n";
     cout<< volume(100l,75,15) <<"\n";
     return 0;
}

//function definitions
int volume(int s) //cube
{
     return (s*s*s);
}

double volume(double r, int h) // cylinder
{
     return (3.14519*r*r*h);
}
long volume (long l, int b, int h) // rectangular box
{
     return (l*b*h);
}
```

Overloading of the functions should be done with caution. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks.

Sometimes, the default arguments may be used instead of overloading. This may reduce the number of functions to be defined.

Math Library Functions

The standard C++ supports many math functions that can be used for performing certain commonly used calculations. Most frequently used math library functions are summarized in following table.

Commonly used math library functions

Function	Purposes
ceil(x)	Rounds x to the smallest integer not less than x ceil(8.1) = 9.0 and ceil(-8.8) = –8.0
cos(x)	Trigonometric cosine of x (x in radians)
exp(x)	Exponential function e^x.
fabs(x)	Absolute value of x. If x>0 then abs(x) is x If x=0 then abs(x) is 0.0 If x<0 then abs(x) is –x
floor(x)	Rounds x to the largest integer not greater than x floor(8.2) = 8.0 and floor(–8.8 = –9.0
log(x)	Natural logarithm of x(base e)
log10(x)	Logarithm of x(base 10)
pow(x,y)	x raised to power y(x^y)
sin(x)	Trigonometric sine of x (x in radians)
sqrt(x)	Square root of x
tan(x)	Trigonometric tangent of x (x in radians)

note

The argument variables **x** and **y** are of type **double** and all the functions return the data type **double.**

To use the math library functions, we must include the header file **math.h** in conventional C++ and **cmath** in ANSI C++.

Limitations of C Structure

The standard C does not allow the struct data type to be treated like built-in types. For example, consider the following structure:

```
struct complex
{
float x;
float y;
};
struct canplex c1, c2, c3;
```

The complex numbers c1, c2, and c3 can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other. For example,

```
c3 = c1 + c2;
```

is illegal in C.

Another important limitation of C structures is that they do not permit *data hiding.* Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In other words, the structure members are public membe.rs.

Extensions to Structures

In C++, a structure can have both variables and functions as members. It can also declare some of its members as ***'private'*** so that they cannot be accessed directly by the external functions.

In C++, the structure names are stand alone and can be used like any other type names. In other words, the keyword struct can be omitted in the declaration of structure variables.

For example, we can declare the student variable A as

```
student A; // C++ decleration
```

Remember, this is an error in C.

C++ incorporates all these extensions in another user-defined type known as class. There is very little syntactical difference between structures and classes in C++ and, therefore. they can be used interchangeably with minor modifications. Since class is a specially introduced data. type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions.

Note: The only difference between a structure and a class in C++ is that, by default. the members of a class are *private,* while, by default. the members of a structure are *public.*

Specifying a Class

When defining a class, we are creating a new ***abstract data type*** that can be treated like any other built-in data type.

Generally, a class specification has two parts:

1. Class declaration

2. Class function definitions

The class declaration describes type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

```
class class_name
{
     private:
     variable declarations;
     function declarations;
     public:
     variable declarations;
     function declarations;
};
```

The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions.

These functions and variables are collectively called ***class members.*** They are usually grouped under two sections, namely, private and public to denote which of the members are private and which of them are public.

The keywords private and public are known as visibility labels. Note that these keywords are followed by a colon.

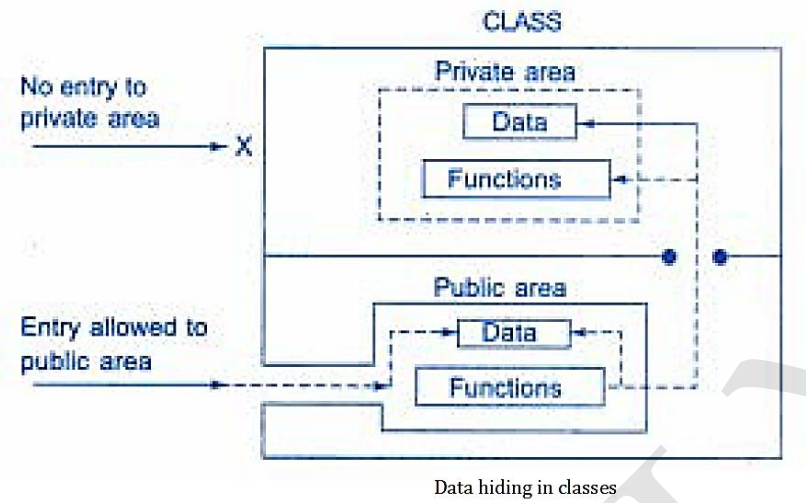
The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also.

The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword private is optional. By default, the members of a class are private.

The variables declared inside the class are known as ***data members*** and the functions are known as ***member functions.***

Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class.

The binding of data and functions together into a single class-type variable is referred to as ***encapsulation.***



Data hiding in classes

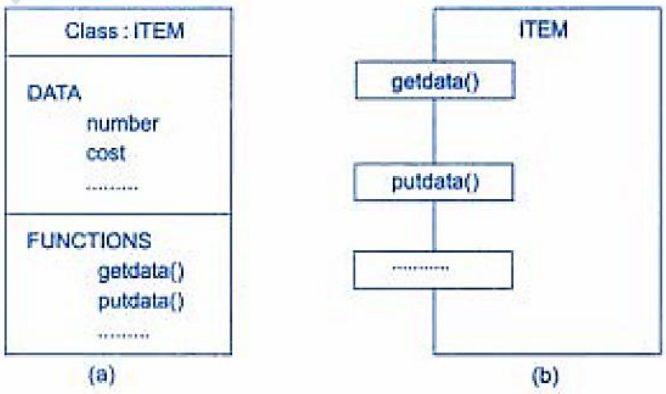
A Simple Class Example

A typical class declaration would look like:

```
class item
{
     int number;//variables declaration
     float cost;// private by default
     public:
     void getdata(int a, float b);
     void putdata(void);
}; // ends with semicolon
```

The data members are private by default while both the functions are public by declaration. The function `getdata()` can be used to assign values to the member variables `number` and `cost`, and `putdata()` for displaying their values. These functions provide the only access to the data members from outside the class.

Figure shows two different notations used by the OOP analysts to represent a class.



Representation of class

Creating Objects

```
item x; // memory for x is created
item p,q,r;
```

creates a variable x of type item. In C++, the class variables are known as ***objects.*** Therefore, x is called an object of type item.

Note that class specification, like a structure, provides only a template and does not create any memory space for the objects.

```
class item
{
     ......
     ......
     ......

} p,q,r;
```

Accessing Class Members

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The main() cannot contain statements that access number and cost directly.

The following is the format for calling a member function:

```
object-name.function-name (actual-arguments);
```

For example, the function call statement

```
x.getdata(100,75.5);
```

is valid and assigns the value 100 to number and 75.5 to cost of the object x by implementing the getdata() function.

Similarly, the statement

```
x.putdata();
```

would display the values of data members.

```
getdata(100,75.5); // error
x.number = 100; // error
     class xyz
     {
          int x;
          int y;
     public:
          int z;
```

```
}
xyz p;
p.x =0;//error
p.z =10;// ok z is public
```

Defining Member Functions

Member functions can be defined in two places:

- **Outside the class definition.**
- **Inside the class definition.**

Outside the Class Definition

An important. difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which class the function belongs to. The general form of a member function definition is:

```
return_type class_name :: function_name (argument declaration)
{
    function body;
}
```

The membership label class-name:: tells the compiler that the function *function-name* belongs to the class *class_name.*

```
void item :: getdata ( int a, float b)
{
    number = a;
    cost =b;
}
void item :: putdata (void)
{
    cout << "Number=" << number << "\n";
    cout << "Cost=" << cost << "\n";
}
```

The member functions have some special characteristics that are often used in the program development These characteristics are:

- Several different classes can use the same function name. The 'membership label' will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so. (However, an exception to this rule is a *friend* function discussed later.)

• A member function can call another member function directly, without using the dot operator.

Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class.

```
class item
{
     int number;
     float cost;
public:
     void getdata (int a, float b);
     void putdata (void)              // definition inside the
class
     {
          cout << "Number=" << number << "\n";
          cout << "Cost=" << cost << "\n";
     }

};
```

When a function is defined inside a class, it is treated as an ***inline function.*** Therefore, all the restrictions and limitations that apply to an inline function are also applicable here. Normally, only small functions are defined inside the class definition.

A C++ Program with Class

```
#include <iostream>
using namespace std;

class item
{
     int number;
     float cost;
public:
     void getdata ( int a, float b);
     void putdata (void)      // definition inside the class
     {
          cout << "Number=" << number << "\n";
          cout << "Cost=" << cost << "\n";
     }

};
```

```
// ............. Member Function Definition .................
void item :: getdata ( int a, float b)
{
     number = a;
     cost = b;
}
// ............. Main program ..............................
int main()
{

     item x;// creats object x
     cout << "\nobject x " << "\n";
     x.getdata(100, 299.95);
     x.putdata();
     item y;// creats another object y
     cout << "\nobject y " << "\n";
     y.getdata(200, 120.25);
     y.putdata();
     return 0;
}
```

Result:

object x
Number=100
Cost=299.95
object y
Number=200
Cost=120.25

Making an Outside Function Inline

We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of function definition, Example:

```
inline void item :: getdata ( int a, float b)
{
     number = a;
     cost = b;
}
```

Nesting of Member Functions

We just discussed that a member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member

function can be called by using its name inside another member function of the same class. This is known as ***nesting of member functions.***

```
#include <iostream>
#include <conio.h>
#include <string>
using namespace std;
class binary
{
     string s;
public:
     void read (void)
     {
          cout << "Enter a binary number: ";
          cin >> s;
     }
void chk_bin (void)
{
     for(int i=0; i<s.length( );i++ )
     {
          if ( s.at(i) != '0' && s.at(i) != '1')
          {
               cout < " \nincorrect binary number format ... the
program will quit";
               getch ();
               exit (0);
          }
     }
}
void ones(void)
{
     chk_bin(); //calling member function
     for(int i=0;i<s.length();i++)
     {
          if(s.at(i)=='0')
               s.at(i)='1';
          else
               s.at(i)='0';
     }
}
void displayones()
{
     ones(); //calling member function
```

```
    cout<<"\nThe ones complement of the above binary number is:
"<<s;
}
};
int main ( )
{
    binary b;
    b.read() ;
    b.displayones();
    getch();
    return 0;
}
```

Private Member Functions

Tasks such as deleting an account in a customer file, or providing increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is n member of its class. Even an object cannot invoke a private function using the dot operator. Consider a class as defined below:

```
class sample
{
int m;
void read(void); // private member function
public:
void update(void);
void write (void);
};
```

If **s1** is an object of **sample** then

```
s1.read(); // won't work; objects cannot access
           //private members
```

However, the function read() can be called by the function update() to update the value of **m.**

```
void sample::update(void)
{
    read(); // simple call; no object used
}
```

Arrays within a Class

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size= 10; // provides value for array size
class array
{
     int a [size]: // 'a' ts tnt type array
     public:
     void setval(void):
     void display(void);
};
```

The array variable a[ ] declared as a private member of the class array can be used in the member functions, like any other array variable. We can perform any operations on it.

For instance, in the above class definition, the member function **setval()** sets the values of elements of the array **a[ ]**, and **display()** function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

Let us consider a shopping list of items for which we place an order with a dealer every month. The list includes details such as the code number and price of each item. We would like to perform operations such as adding an item to the list, deleting an item from the list and printing the total value of the order. Following program shows how these operations are implemented using a class with arrays as data members.

```
#include <iostream>
using namespace std;
const int m=50;
class items
{
     int itemcode[m];
     float itemprice[m];
     int count;
     public:
     void cnt(void) {count = 0;} // initializes count to 0
     void getitem(void);
     void displaysum(void);
     void remove(void);
     void displayitems(void);
};
//=======================================
```

```
void items::getitem(void) // assign values to data members
                          //of item
{
     cout <<"enter item code : ";
     cin >>itemcode[count];
     cout<<"enter item cost:";
     cin >>itemprice[count];
     count++;
}
void items::displaysum(void) //display total value of all
                             //items
{
     float sum=0;
     for(int i=0; i<count; i++)
          sum=sum+itemprice[i];
     cout<<"\ntotal value :" <<sum<<"\n";
}
void items::remove(void) // delete a specified item
{
     int a;
     cout<<"\n enter item code";
     cin >>a;
     for(int i=0; i<count; i++)
          if (itemcode[i]==a)
               itemprice[i]=0;
}
void items:: displayitems(void) //displaying items
{
     cout<<"\n code    price\n";
     for(int i=0; i<count; i++)
     {
          cout<<"\n"<<itemcode[i]
              <<"       "<<itemprice[i];
     }
     cout<<"\n ";
}

//=======================================
int main()
{
items order;
order.cnt();
int x;
```

```
do
     {
          cout << "\nyou can do the following;"
               << " enter appropriate number \n";
          cout <<"\n1 : add an item";
          cout <<"\n2 : display total value";
          cout <<"\n3 : delete an item";
          cout <<"\n4 : display all items";
          cout <<"\n5 : quit";
          cout <<"\nwhat is your option? ";
          cin>>x;
          switch (x)
          {
               case 1:order.getitem(); break;
               case 2:order.displaysum(); break;
               case 3:order.remove(); break;
               case 4:order.displayitems(); break;
               case 5: break;
               default: cout<<"\n error in input; try again\n";
          }

     } while (x!=5);
     return 0;
}
```

The sample output of above code is

you can do the following; enter appropriate number
1 : add an item
2 : display total value
3 : delete an item
4 : display all items
5 : quit
what is your option? 1
enter item code : 111
enter item cost:100

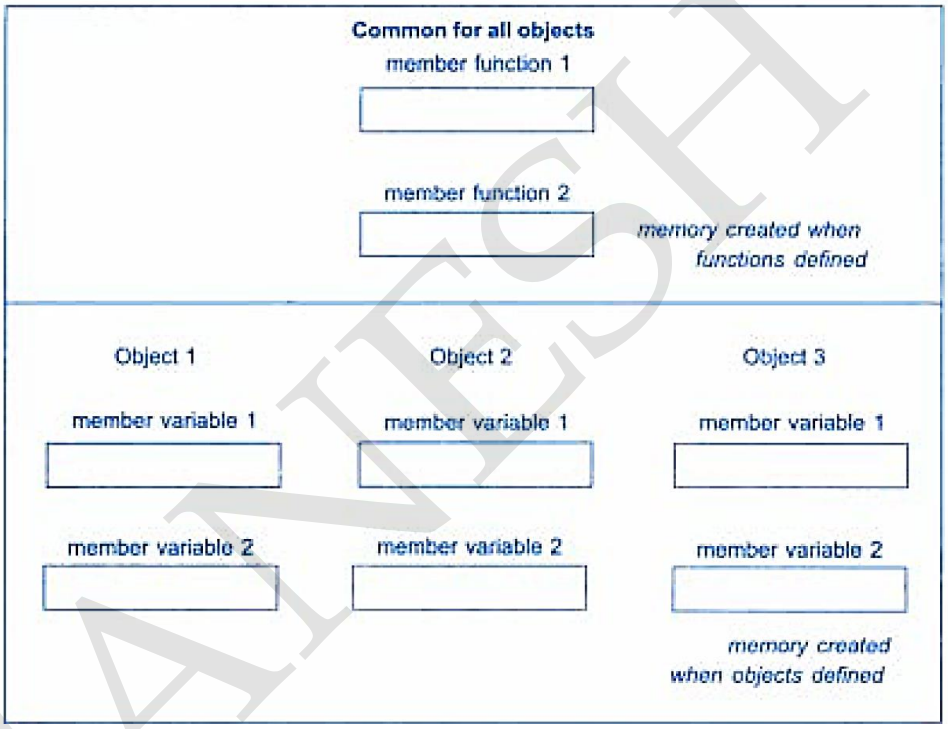
you can do the following; enter appropriate number
1 : add an item
2 : display total value
3 : delete an item
4 : display all items
5 : quit
what is your option? 1

```
enter item code : 222
enter item cost:200

you can do the following; enter appropriate number
1 : add an item
2 : display total value
3 : delete an item
4 : display all items
5 : quit
what is your option? 1
enter item code : 333
enter item cost:300

you can do the following; enter appropriate number
1 : add an item
2 : display total value
3 : delete an item
4 : display all items
5 : quit
what is your option? 2
total value :600

you can do the following; enter appropriate number
1 : add an item
2 : display total value
3 : delete an item
4 : display all items
5 : quit
what is your option? 3
 enter item code222

you can do the following; enter appropriate number

1 : add an item
2 : display total value
3 : delete an item
4 : display all items
5 : quit
what is your option? 4
 code        price
111          100
222          0
333          300
```

you can do the following; enter appropriate number
1 : add an item
2 : display total value
3 : delete an item
4 : display all items
5 : quit
what is your option? 5

The program uses two arrays, namely itemcode[ ] to hold the code number of items and itemprice[ ] to hold the prices. A third data member count is used to keep a record of items in the list. The program uses a total of four functions to implement the operations to be performed on the list.

Memory Allocation for Objects



We have stated that the memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true.

Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created.

Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects. This is shown in above Fig.

Static Data Members

A data member of a class can be qualified as static. The properties of a **static** member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are:

• It is initialized to zero when the first object of its class is created. No other initialization is permitted.

• Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

• It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects.

```
class items
{
     static int num;
     ..............
     ..............
public:
     ..............
     ..............
};
int items:: num; // definition of static data members
```

Note that the type and scope of each static member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object.

Since they are associated with the class itself rather than with any class object, they are also known as ***class variables.***

Example:

```
#include <iostream>
using namespace std;
class item
{
     static int count;
     int number;
     public:
     void getdata(int a)
```

```
        {
        number=a;
        count++;
        }
        void getcount(void)
        {
             cout <<"Count: ";
             cout << count << "\n";
        }
};
int item::count;
int main()
{
        item a, b, c; //count is initialized to zero
        a.getcount ();
        b.getcount();
        c.getcount ();
        a.getdata(100); // get data into object a
        b.getdata (200);// get data into object b
        c.getdata(300);// get //display countdata into object c
        cout << "After reading data"<< "\n";
        a.getcount ();//display count
        b.getcount();
        c.getcount ();
        return 0;
}
```

Output:

```
Count: 0
Count: 0
Count: 0
After reading data
Count: 3
Count: 3
Count: 3
```

Static Member Functions

Like **static** member variable, we can also have static member functions. A member function that is declared static has the following properties:

• A static function can have access to only other static members (functions or variables) declared in the same class.

• A static member function can be called using the class name (instead of its objects) as follows:

```
class_name :: function_name();
```

Example:

```
class items
{
     static int num;
     ..............
     ..............
public:
     ..............
     ..............
     static void showcount()
     {
          cout<< num;
     }
};

int items:: num; // definition of static data members

int main()
{
     ..............
     ..............
     items :: showcount();
     ..............
     ..............
}
```

Example :

```
#include <iostream>
using namespace std;
class static_type
{
```

```
    static int i; //static data member
public:
    static void init(int x) { i=x; } //static member function
    void show() { cout<<i; }
};
int static_type::i; // define static i
int main()
{
    static_type::init(100);//initialise static data before object
                                creation
    static_type x;
    x.show( ); //displays 100
    return 0;
}
```

Example 2:

```
#include <iostream>
using namespace std;
class test
{
    int code;
    static int count; // static member variable
    public:
    void setcode (void)
    {
    code = ++count;
    }
    void showcode(void)
    {
    cout <<"object number: "<<code << "\n";
    }
    static void showcount(void) // static member function
    {
       cout << "count: "<<count<< "\n";
    }
};
int test :: count;
int main()
{
    test t1,t2;
    t1.setcode();
    t2.setcode();
```

```
    test::showcount();
    test t3;
    t3.setcode();
    test::showcount();
    t1.showcode();
    t2.showcode();
    t3.showcode();
    return 0;
}
```

Arrays of Objects

We know that an array can be of any data type including **struct.** Similarly, we can also have arrays of variables that are of the type **class.** Such variables are called ***arrays of objects.*** Consider the following class definition:

```
class employee
{
    char name[10];
    float age;
    public:
    void getdata(void);
    void putdata(void);
};
```

The identifier employee is a user-defined data type and can be used to create objects that relate to different. categories of the employees. Example:

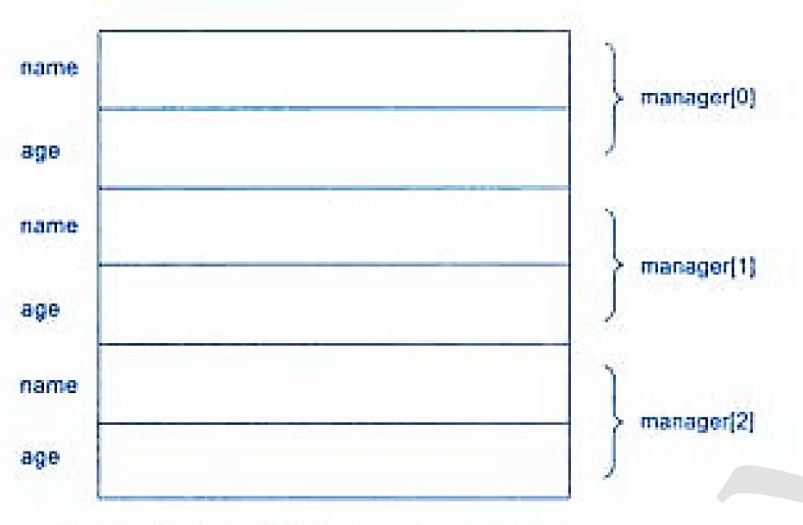
```
employee manager[3] ; //array of manager
employee foreman[15] ; //array of foremen
employee worker[75] ; //array of worker
```

The array manager contains three objects (managers). namely, manager[0], manager[1] and manager[2], of type employee class. Similarly, the foreman array contains 15 objects (foremen) and the worker array contains 75 objects(workers).

Since an array of objects behaves like any other array, we can use the usual array accessing methods to access individual elements, and then the dot member operator to access the member functions. For example, the statement

```
manager[i].putdata( );
```

An array of objects is stored inside the memory in the same way as a multi-dimensional array. The array manager is represented in following Fig. 5.5. Note that only the space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.




```
#include <iostream>
using namespace std;

class employee
{
     char name[10];
     float age;
     public:
     void getdata(void);
     void putdata(void);
};
void employee::getdata(void)
{
     cout <<"\nEnter name: ";
     cin >> name;
     cout <<"\nEnter age: ";
     cin >> age;
}
void employee::putdata(void)
{
     cout <<"\nName: "<<name;
     cout <<"\nAge: "<<age;
}

const int size=3;

int main()
{
     employee manager[size];
     for(int i=0; i<size; i++)
```

```
        {
            cout <<"\nDetails of Manager"<<i+1<<"\n";
            manager[i].getdata();
        }
        cout<<"\n";
        for(int i=0; i<size; i++)
        {
            cout <<"\nManager"<<i+1<<"\n";
            manager[i].putdata();
        }
        return 0;
    }
```

Objects as Function Arguments

Like any other data type an object may be used as a function argument. This can be done in two ways:

- A copy of the entire object is passed to the function. (pass by vale)
- Only the address of the object is transferred to the function. (pass by reference)

```
#include <iostream>
using namespace std;
class time
{
    int hours;
    int minutes;
public:
    void gettime(int h, int m)
    {
        hours=h;
        minutes=m;
    }
    void puttime(void)
    {
        cout<<hours<<" hours and ";
        cout<<minutes<<" minutes"<<"\n";
    }
    void sum(time, time); // declaration with object as arguments
};
void time::sum(time t1, time t2)
{
    minutes = t1.minutes +t2.minutes;
```

```
    hours= minutes/60;
    minutes = minutes%60;
    hours= hours + t1.hours+ t2.hours;
}
int main()
{
    time t1,t2,t3;
    t1.gettime(2,45); // get t1
    t2.gettime(3,30); // get t2
    t3.sum(t1,t2); //t3=t1+t2
    cout<<"t1 = "; t1.puttime(); // display t1
    cout<<"t2 = "; t2.puttime(); // display t2
    cout<<"t3 = "; t3.puttime(); // display t3
    return 0;
}
```

Friendly Functions

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

```
#include<iostream>
using namespace std;

class base
{
    int val1,val2;
public:
    void get()
    {
        cin>>val1>>val2;
    }
    friend float mean(base ob);
};

float mean(base ob)
{
    return float(ob.val1 + ob.val2)/2;
}
```

```
int main()
{
     base obj;
     obj.get();
     cout<<"\n Mean value is : "<<mean(obj);
     return 0;
}
```

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name (e.g. A.x).
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.
- Member functions of one class can be friend function of another class.

```
class X
{
     int fun();
};
class Y
{
     friend int X::fun();
};
```

- We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a friend class. This can be specified follows:

```
class Z
{
     friend class X;
};
```

- Consider following example

```
class Y;// forward declaration
class X
{
```

```
        friend int fun(X,Y);
};
class Y
{
        friend int fun(X,Y);
};

int fun(X x, Y y)
{
        ........;
}
```

• As pointed out earlier, a friend function can be called by reference. In this case, local copies of the objects are not made. Instead, a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.

This method can be used to alter the values of the private members of a class. Remember, altering the values of private members is against the basic principles of **data hiding.** It should be used only when absolutely necessary.

// write a C++ prog using friend function to exchange the private values of the two classes.

```
#include <iostream>

using namespace std;

class class_2;

class class_1
{
    int value1;
  public:
    void indata(int a) {value1 = a;}
    void display(void) {cout << value1 << "\n";}
    friend void exchange(class_1 &, class_2 &);
};

class class_2
{
    int value2;
  public:
    void indata(int a) {value2 = a;}
    void display(void) {cout << value2 << "\n";}
    friend void exchange(class_1 &, class_2 &);
};
```

```
void exchange(class_1 & x, class_2 & y)
{
     int temp = x.value1;
     x.value1 = y.value2;
     y.value2 = temp;
}

int main()
{
     class_1 C1;
     class_2 C2;

     C1.indata(100);
     C2.indata(200);

     cout << "Values before exchange" << "\n";
     C1.display();
     C2.display();
     exchange(C1, C2); // swapping

     cout << "Values after exchange " << "\n";
     C1.display();
     C2.display();

     return 0;
}
```

Returning Objects

A function cannot only receive objects as arguments but also can return them.

```
X fun(X x, X y)
{
    X z;
    z= x+y;
    return z;
}
```

const Member Functions

If a member function does not alter any data in the class. then we may declare it as a **const** member function as follows;

```
void mul(int, int) const;
double get_balance() const;
```

The qualifier **const** is appended to the function prototypes (in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.

Pointers to Objects

```
#include <iostream>
using namespace std;
class cl {
     int i;
public:
     int get_i() { return i; }
};
int main()
{
     cl ob, *p;
     p = &ob; // get address of ob
     cout << p->get_i(); // use -> to call get_i()
     return 0;
}
```

Pointers to Members

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a '"fully qualified" class member name.

A class member pointer can be declared using the operator **::*** with the class name. For example, given the class

```
class A
{
    private:
        int m;
    public:
        void show();
};
```

We can define a pointer to the member m as follows:

```
int A ::* p = &A :: m;
```

The p pointer created thus acts like a class member in that it must be invoked with a class object. In the statement above, the phrase **A::*** means " pointer-to--member of A class". The phrase **&A :: m** means the "address of the m member of A class".

```
int *p = &m; // wont work
```

This is because m is not simply an int type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer **p** can now be used to access the member **m** inside member functions (or friend functions). Let us assume that **a** is an object of **A** declared in a member function.

We can access **m** using the pointer **p** as follows:

```
cout << a.*p; // display
cout << a.m; //same as obave
```

Now, look at the following code:

```
ap = &a; // ap is pointer to object a
cout << ap -> *p; // display m
cout << ap -> m; // some as above
```

The *dereferencing operator* **->*** is used to access a member when we use pointers to both the object and the member. The *dereferencing operator* **.*** is used when the object itself is used with the member pointer. Note that ***p** is used like a member name.

We can also design pointers to member functions which, then, can be invoked using the dereferencing operators in the main as shown below :

```
(object-name .* pointer-to-member function) (10);
(pointer-to-object ->* pointer-to-member function) (10);
```

The precedence of ( ) is higher than that of **.*** and **->***, so the parentheses are necessary. Example:

```
#include <iostream>
using namespace std;
class M
{
    int x;
    int y;
public:
    void set_xy (int a, int b)
    {
        x = a;
        y = b;
    }
    friend int sum(M m);
};

int sum(M m)
{
```

```
        int M ::* px = &M :: x;
        int M ::* py = &M :: y;
        M *pm = &m;
        int s= m.*px + pm->*py;
        return s;
}
int main()
{
        M n;
        void (M ::* pf)(int,int) = &M :: set_xy;
        (n .* pf)(10, 20);
        cout << "SUM = " << sum(n) << "\n";
        M *op = &n;
        (op->*pf)(10,40);
        cout << "SUM = "<< sum(n) << "\n";
return 0;
}
```

Local Classes

Classes can be defined and used inside **a** function or a block. Such classes are called local classes. Examples:

```
void test( int a) //function
{
        .......
        .......
        class student // local class
        {
                ......
                .......
        };
        .......
        student s1(a); // create object
        .......
}
```

Local classes can use global variables (declanid above the function) and static variables declared, inside the function but cannot use automatic local variables. The global variables should be used with the soope operator (::).

There are some restrictions in constructing local classes. They cannot have static data members and member functions must be defined inside the local classes. Enclosing function cannot access the private members of a local class. However. we can achieve this by declaring the enclosing function as a friend.

```
38    return 0;
39}
```

output

```
Enter consumer name & unit consumed :sattar 200
Name              Charge

sattar            210

Press o for exit / press 1 to input again :1

Enter consumer name & unit consumed :santo 300

Nmae          Charge

santo         290

Press o for exit / press 1 to input again : 0
```

Chapter 4

Review Questions

4.1: **State whether the following statements are TRUE or FALSE.**
(a) A function argument is a value returned by the function to the calling program.
(b) When arguments are passed by value, the function works with the original arguments in the calling program.
(c) When a function returns a value, the entire function call can be assigned to a variable.
(d) A function can return a value by reference.
(e) When an argument is passed by reference, a temporary variable is created in the calling program to hold the argument value.
(f) It is not necessary to specify the variable name in the function prototype.

Ans:

(a) FALSE
(b) FALSE
(c) TRUE
(d) TRUE
(e) FALSE
(f) TRUE

4.2: **What are the advantages of function prototypes in C++?**

Ans: Function prototyping is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values.

4.3: **Describe the different styles of writing prototypes.**

Ans:
General form of function prototyping :
return_type function_name (argument_list)

Example :
int do_something (void);
float area (float a, float b);
float area (float, float);

4.4: **Find errors, if any, in the following function prototypes.**
(a) float average(x,y);
(b) int mul(int a,b);
(c) int display(….);
(d) void Vect(int? &V, int & size);
(e) void print(float data[], size = 201);

Ans:

No.	Error	Correction
(a)	Undefined symbol x, y	float average (float x, floaty)
(b)	Undefined symbol b	int mul (int a, int b);
(c)	No error	
(d)	invalid character in variable name	void vect (int &v, int &size);
(e)	Undefined symbol ‘s’	void print (float data [ ], int size = 20);

4.5: **What is the main advantage of passing arguments by reference?**

Ans: When we pass arguments by reference, the formal arguments in the called function become aliases to the ‘actual’ arguments in the calling function.

4.6: **When will you make a function inline? Why?**

Ans: When a function contains a small number of statements, then it is declared as inline function.

By declaring a function inline the execution time can be minimized.

4.7: **How does an inline function differ from a preprocessor macro?**

Ans: The macos are not really functions and therefore, the usual error checking does not occur during compilation. But using inline-function this problem can be solved.

4.8: **When do we need to use default arguments in a function?**

Ans: When some constant values are used in a user defined function, then it is needed to assign a default value to the parameter.
Example :

```
1Float area (float r, float PI = 3.1416)
2    {
3          return PI*r*r;
4    }
```

4.9: **What is the significance of an empty parenthesis in a function declaration?**

Ans: An empty parentheses implies arguments is void type.

4.10: **What do you meant by overloading of a function? When do we use this concept?**

Ans: Overloading of a function means the use of the same thing for different purposes.
When we need to design a family of functions-with one function name but with different argument lists, then we use this concept.

4.11: **Comment on the following function definitions:**
(a)

```
1int *f( )
2{
3int m = 1;
4.....
5.....
6return(&m);
7}
```

(b)

```
1double f( )
2{
3.....
4.....
5return(1);
6}
```

(c)

```
1int & f()
2{
3int n - 10;
4.....
5.....
6return(n);
7}
```

Ans:

No.	Comment
(a)	This function returns address of m after execution this function.
(b)	This function returns 1 after execution.
(c)	returns address of n

Debugging Exercises

4.1: **Identify the error in the following program.**

```
#include <iostream.h>
int fun()
{
    return 1;
}
float fun()
{
    return 10.23;
}
void main()
{
    cout <<(int)fun() << ' ';
    cout << (float)fun() << ' ';
}
```

Solution: Here two function are same except return type. Function overloading can be used using different argument type but not return type.

Correction : This error can be solved as follows :

```
#include<iostream.h>

int fun()
{
   return 1;
}
float fun1()
{
   return 10.23;
}

void main()
{
    cout<<fun()<<"  ";
    cout<<fun1()<<"  ";
}
```

4.2: **Identify the error in the following program.**

```
#include <iostream.h>
void display(const Int constl=5)
{
  const int const2=5;
  int arrayl[constl];
  int array2[const2];
  for(int 1=0; i<5; 1++)
 {
    arrayl[i] = i;
    array2[i] = i*10;
    cout <<arrayl[i]<< ' ' << array2[i] << ' ' ;
 }
}
void main()
{
  display(5);
}
```

Solution:

```
#include<iostream.h>
```

```
void display()
{
    const int const1=5;
    const int const2=5;
    int array1[const1];
    int array2[const2];

     for(int i=0;i<5;i++)
     {
        array1[i]=i;
        array2[i]=i*10;
        cout<<array1[i]<<"  "<<array2[i]<<"  ";
     }
}

void main()
{
   display();
}
```

4.3: **Identify the error in the following program.**

```
#include <iostream.h>
int gValue=10;
void extra()
{
    cout << gValue << ' ';
}
void main()
{
   extra();
   {
   int gValue = 20;
   cout << gValue << ' ';
   cout << : gValue << ' ';
   }
}
```

Solution:
Here cout << : gvalue << " "; replace with cout <<::gvalue<< " ";

```
#include <iostream.h>
int gValue=10;
void extra()
{
```

```
    cout << gValue << ‘ ‘;
}
void main()
{
   extra();
   {
   int gValue = 20;
   cout << gValue << ‘ ‘;
   cout <<::gvalue<< " ";
   }
}
```

4.4: **Find errors, if any, in the following function definition for displaying a matrix: void display(int A[][], int m, int n)**

```
{
 for(1=0; i<m; i++)
 for(j=o; j<n; j++)
  cout<<" "<<A[i][j];
  cout<<"\n";
}
```

Solution:
First dimension of an array may be variable but others must be constant.

Here int A [ ] [ ] replace by the following code:
int A [ ] [10];
int A[10] [10];
int A[ ] [size];
int A [size] [size];

Where const int size = 100;
any other numerical value can be assigned to size.

Programming Exercises

4.1: **Write a function to read a matrix of size m*n from the keyboard.**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3
4  void matrix(int m,int n)
```

```
5  {
6     float **p;
7     p=new float*[m];
8     for(int i=0;i<m;i++)
9     {
10       p[i]=new float[n];
11    }
12    cout<<" Enter "<<m<<"by"<<n<<" matrix elements one by one "<<endl;
13    for(i=0;i<m;i++)
14    {
15       for(int j=0;j<n;j++)
16       {
17          float value;
18          cin>>value;
19          p[i][j]=value;
20       }
21    }
22    cout<<" The given matrix is :"<<endl;
23    for(i=0;i<m;i++)
24    {
25       for(int j=0;j<n;j++)
26       {
27          cout<<p[i][j]<<"   ";
28       }
29       cout<<"\n";
30    }
31}
32
33int main()
34{
35    int r,c;
36    cout<<" Enter size of matrix : ";
37    cin>>r>>c;
38    matrix(r,c);
39    return 0;
40}
```

output

Enter size of matrix : 3 4

Enter 3 by 4 matrix elements one by one

1 2 3 4

2 3 4 5

3 4 5 6

The given matrix is :

```
1   2   3   4

2   3   4   5

3   4   5   6
```

4.2: **Write a program to read a matrix of size m*n from the keyboard and display the same on the screen using function.**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3
4  void matrix(int m,int n)
5  {
6     float **p;s
7     p=new float*[m];
8     for(int i=0;i<m;i++)
9     {
10       p[i]=new float[n];
11    }
12    cout<<" Enter "<<m<<" by "<<n<<" matrix elements one by one "<<endl;
13    for(i=0;i<m;i++)
14    {
15       for(int j=0;j<n;j++)
16       {
17          float value;
18          cin>>value;
19          p[i][j]=value;
20       }
21    }
22    cout<<" The given matrix is :"<<endl;
23    for(i=0;i<m;i++)
24    {
25       for(int j=0;j<n;j++)
26       {
27          cout<<p[i][j]<<"   ";
28       }
29       cout<<"\n";
30    }
31}
32
33int main()
34{
35    int r,c;
36    cout<<" Enter size of matrix : ";
```

```
37    cin>>r>>c;
38    matrix(r,c);
39    return 0;
40}
```

output

Enter size of matrix : 4 4

Enter 4 by 4 matrix elements one by one

1 2 3 4 7

2 3 4 5 8

3 4 5 6 9

The given matrix is :

1 2 3 4 7

2 3 4 5 8

3 4 5 6 9

4.3: **Rewrite the program of Exercise 4.2 to make the row parameter of the matrix as a default argument.**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3
4  void matrix(int n,int m=3)
5  {
6     float **p;
7     p=new float*[m];
8     for(int i=0;i<m;i++)
9     {
10       p[i]=new float[n];
11    }
12    cout<<" Enter "<<m<<" by "<<n<<" matrix elements one by one "<<endl;
13    for(i=0;i<m;i++)
14    {
15       for(int j=0;j<n;j++)
16       {
```

```
17          float value;
18          cin>>value;
19          p[i][j]=value;
20      }
21   }
22   cout<<" The given matrix is :"<<endl;
23   for(i=0;i<m;i++)
24   {
25      for(int j=0;j<n;j++)
26      {
27         cout<<p[i][j]<<"   ";
28      }
29      cout<<"\n";
30   }
31}
32
33int main()
34{
35   int c;
36   cout<<" Enter column of matrix : ";
37   cin>>c;
38   matrix(c);
39   return 0;
40}
```

output

Enter column of matrix : 3

Enter 3 by 3 matrix elements one by one

1 2 3

2 3 4

3 4 5

The given matrix is :

1 2 3

2 3 4

3 4 5

4.4: **The effect of a default argument can be alternatively achieved by overloading. Discuss with examples.**

Solution:

```
#include<iostream.h>
#include<iomanip.h>

void matrix(int m,int n)
{
   float **p;
   p=new float*[m];
   for(int i=0;i<m;i++)
   {
      p[i]=new float[n];
   }
   cout<<" Enter "<<m<<"by"<<n<<" matrix elements one by one "<<endl;
   for(i=0;i<m;i++)
   {
      for(int j=0;j<n;j++)
      {
         float value;
         cin>>value;
         p[i][j]=value;
      }
   }
   cout<<" The given matrix is :"<<endl;
   for(i=0;i<m;i++)
   {
      for(int j=0;j<n;j++)
      {
         cout<<p[i][j]<<"    ";
      }
      cout<<"\n";
   }
}
void matrix(int m,long int n=3)
{
   float **p;
   p=new float*[m];

       for(int i=0;i<m;i++)
   {
      p[i]=new float[n];
   }
   cout<<" Enter "<<m<<" by "<<n<<" matrix elements one by one "<<endl;
   for(i=0;i<m;i++)
   {
      for(int j=0;j<n;j++)
      {
         float value;
         cin>>value;
         p[i][j]=value;
```

```
49      }
50   }
51   cout<<" The given matrix is :"<<endl;
52   for(i=0;i<m;i++)
53   {
54      for(int j=0;j<n;j++)
55      {
56         cout<<p[i][j]<<"   ";
57      }
58      cout<<"\n";
59   }
60}
61
62int main()
63{
64   int r;
65   cout<<" Enter row of matrix : ";
66   cin>>r;
67   matrix(r);
68   return 0;
69}
```

output

Enter column of matrix : 2

Enter 2 by 3 matrix elements one by one

1 0 1

0 2 1

The given matrix is :

1 0 1

0 2 1

4.5: **Write a macro that obtains the largest of the three numbers.**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3
4  float  large(float a,float b,float c)
5  {
```

```
6     float largest;
7     if(a>b)
8     {
9        if(a>c)
10          largest=a;
11       else
12          largest=c;
13    }
14    else
15    {
16       if(b>c)
17          largest=b;
18       else
19          largest=c;
20    }
21    return largest;
22}
23
24int main()
25{
26    float x,y,z;
27    cout<<" Enter three values : ";
28    cin>>x>>y>>z;
29    float largest=large(x,y,z);
30    cout<<" large = "<<largest<<endl;
31    return 0;
32}
```

output

Enter three values : 4 5 8

large = 8

4.6: **Redo Exercise 4.16 using inline function. Test the function using a main function.**

Solution:

Blank

4.7: **Write a function power() to raise a number m to power n. The function takes a double value for m and int value for n and returns the result correctly. Use a default value of 2 for n to make the function to calculate the squares when this argument is omitted. Write a main that gets the values of m and n from the user to test the function.**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  #include<math.h>
4
5  long double power(double m,int n)
6  {
7     long double mn=pow(m,n);
8     return mn;
9  }
10long double power(double m,long int n=2)
11{
12    long double mn=pow(m,n);
13    return mn;
14}
15int main()
16{
17    long double mn;
18    double m;
19    int n;
20
21    cout<<" Enter the value of m & n"<<endl;
22    cin>>m>>n;
23    mn=power(m,n);
24    cout<<" m to power n : "<<mn<<endl;
25    mn=power(m);
26    cout<<" m to power n : "<<mn<<endl;
27    return 0;
28}
```

output

Enter the value of m & n

12 6

m to power n : 2985984

m to power n: 144

4.6: **Redo Exercise 4.16 using inline function. Test the function using a main function.**

Solution:

Blank

4.7: **Write a function power() to raise a number m to power n. The function takes a double value for m and int value for n and returns the result correctly. Use a default value of 2 for**

n to make the function to calculate the squares when this argument is omitted. Write a main that gets the values of m and n from the user to test the function.

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  #include<math.h>
4
5  long double power(double m,int n)
6  {
7     long double mn=pow(m,n);
8     return mn;
9  }
10long double power(double m,long int n=2)
11{
12    long double mn=pow(m,n);
13    return mn;
14}
15int main()
16{
17    long double mn;
18    double m;
19    int n;
20
21    cout<<" Enter the value of m & n"<<endl;
22    cin>>m>>n;
23    mn=power(m,n);
24    cout<<" m to power n : "<<mn<<endl;
25    mn=power(m);
26    cout<<" m to power n : "<<mn<<endl;
27    return 0;
28}
```

output

Enter the value of m & n

12 6

m to power n : 2985984

m to power n: 144

4.8: **Write a function that performs the same operation as that of Exercise 4.18 but takes an int value for m. Both the functions should have the same name. Write a main that calls both the functions. Use the concept of function overloading.**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  #include<math.h>
4
5  long double power(int m,int n)
6  {
7      long double mn= (long double)pow(m,n);
8      return mn;
9  }
10long double power(int m,long int n=2)
11{
12    long double mn=(long double)pow(m,n);
13    return mn;
14}
15int main()
16{
17    long double mn;
18    int m;
19    int n;
20
21    cout<<" Enter the value of m & n"<<endl;
22    cin>>m>>n;
23    mn=power(m,n);
24    cout<<" m to power n : "<<mn<<endl;
25    mn=power(m);
26    cout<<" m to power n : "<<mn<<endl;
27    return 0;
28}
```

output

Enter the value of m & n

15 16

m to power n : 6.568408e+18

m to power n: 225

Chapter 5

Review Questions

5.1: **How do structures in C and C++ differ?**

Ans:
C structure member functions are not permitted but in C++ member functions are permitted.

5.2: **What is a class? How does it accomplish data hiding?**

Ans:
A class is a way to bind the data and its associated functions together. In class we can declare a data as private for which the functions accomplish data−outside the class can not access the data and thus if hiding.

5.3: **How does a C++ structure differ from a C++ class?**

Ans:
Initially (in C) a structure was used to bundle different of data types together to perform a particular functionality C++ extended the structure to contain functions also. The difference is that all declarations inside a structure are default public.

5.4: **What are objects? How are they created?**

Ans:
Object is a member of class. Let us consider a simple example. int a; here a is a variable of int type. Again consider class fruit.
{
}
here fruit is the class-name. We can create an object as follows:
fruit mango;
here mango is a object.

5.5: **How is a member function of a class defined?**

Ans:
member function of a class can be defined in two places:
* Outside the class definition.
* Inside the class definition.

Inside the class definition : same as other normal function.

Outside the class definition : general form:
return-type class-name : function-name (argument list)

```
{
function body
}
```

5.6: **Can we use the same function name for a member function of a class and an outside function in the same program file? If yes, how are they distinguished? If no, give reasons.**

Ans:
Yes, We can distinguish them during calling to main ( ) function. The following example illustrates this:

```
1  #include<iostream.h>
2  void f()
3  {
4  cout<<"Outside Of class \n";
5  }
6
7  class santo
8  {
9  public:
10void f()
11{
12            cout<<"Inside of class \n";
13}
14};
15
16void main()
17{
18f();  // outside f() is calling.
19santo robin;
20robin.f();  // Inside f() is calling.
21}
```

5.7: **Describe the mechanism of accessing data members and member functions in the following cases:**
(a) Inside the main program.
(b) Inside a member function of the same class.
(c) Inside a member function of another class.

Ans:
(a) Using object and dot membership operator.
(b) Just like accessing a local variable of a function.
(c) Using object and dot membership operator.

The following example explains how to access data members and member functions inside a member function of another class.

```
1  #include<iostream.h>
2
3  class a
4  {
5       public:
6             int x;
7        void display()
8         {
9            cout<<"This is class a \n";
10           x=111;
11        }
12};
13
14class b
15{
16     public:
17      void display()
18      {
19         a  s;
20         cout<<" Now member function 'display()' of class a is calling from class b \n";
21         s.display();
22         cout<<" x = "<<s.x<<"\n";
23      }
24};
25
26void main()
27{
28         b  billal; // billal is a object of class b.
29          billal.display();
30}
```

5.8: **When do we declare a member of a class static?**

Ans:
When we need a new context of a variable the n we declare this variable as static.

5.9: **What is a friend function? What are the merits and demerits of using friend functions?**

Ans:
A function that acts as a bridge among different classes, then it is called friend function.

Merits :
We can access the other class members in our class if we use friend keyword. We can access the members without inheriting the class.

demerits :
Maximum size of the memory will occupied by objects according to the size of friend members.

5.10: **State whether the following statements are TRUE or FALSE.**
(a) Data items in a class must always be private.
(b) A function designed as private is accessible only to member functions of that class.
(c) A function designed as public can be accessed like any other ordinary functions.
(d) Member functions defined inside a class specifier become inline functions by default.
(e) Classes can bring together all aspects of an entity in one place.
(f) Class members are public by default.
(g) Friend junctions have access to only public members of a class.
(h) An entire class can be made a friend of another class.
(i) Functions cannot return class objects.
(j) Data members can be initialized inside class specifier.

Ans:
(a) FALSE
(b) TRUE
(c) FALSE

*A function designed as public can be accessed like any other ordinary functions from the member function of same class.
(d) TRUE
(e) TRUE
(f) FALSE
(g) FALSE
(h) TRUE
(i) FALSE
(j) FALSE

Debugging Exercises

5.1: **Identify the error in the following program**

```
1  #include <iosream.h>
2  struct Room
3  {
4     int width;
5     int length;
6     void setValue(int w, int l)
7     {
8          width = w;
9          length = l;
```

```
10    }
11};
12void main()
13{
14Room objRoom;
15objRoom.setValue(12, 1,4);
16}
```

Solution:
Void setvalue (in w, int l) function must be public.

5.2: **Identify the error in the following program**

```
1  #include <iosream.h>
2  class Room
3  {
4     int width, int length;
5     void setValue(int w, int h)
6     {
7          width = w;
8          length = h;
9      }
10};
11void main()
12{
13Room objRoom;
14objRoom.width=12;
15}
```

Solution:
Void setvalue (int w, int l) function must be public.

5.3: **Identify the error in the following program**

```
1  #include <iosream.h>
2  class Item
3  {
4     private:
5           static int count;
6     public:
7          Item()
8          {
9          count++;
10}
```

```
11int getCount()
12{
13    return count;
14}
15int* getCountAddress()
16{
17    return count;
18   }
19};
20int Item::count = 0;
21void main()
22{
23    Item objlteml;
24    Item objltem2;
25
26    cout << objlteml.getCount() << ' ';
27    cout << objltem2.getCount() << ' ';
28
29    cout << objlteml.getCountAddress() << ' ';
30    cout << objltem2.getCountAddress() << ' ';
31}
```

Solution:

```
1
2int* getCountAddress ( )
3     {
4        return &count;
5    }
```

Note: All other code remain unchanged.

5.4: **Identify the error in the following program**

```
1  #include <iosream.h>
2  class staticfunction
3  {
4     static int count;
5  public:
6     static void setCounto()
7      {
8        count++;
9      }
10    void displayCount()
11   {
12      cout << count;
13   }
```

```
14};
15int staticFunction::count = 10;
16void main()
17{
18    staticFunction obj1;
19    obj1setcount(5);
20    staticFunction::setCount();
21    obj1.displayCount();
22}
```

Solution:
setCount ( ) is a void argument type, so here obj1.setCount (5); replace with obj1.setcount( );

5.5: **Identify the error in the following program**

```
1  #include <iosream.h>
2  class Length
3  {
4      int feet;
5      float inches;
6  public:
7        Length()
8     {
9         feet = 5;
10        inches = 6.0;
11    }
12    Length(int f, float in)
13    {
14        feet = f;
15        inches=in;
16    }
17    Length addLength(Length 1)
18    {
19
20       1.inches this->inches;
21       1.feet += this->feet;
22       if(1.inches>12)
23    {
24
25       1.inches-=12;
26       1.feet++;
27    }
28    return 1;
29 }
30 int getFeet()
31 {
32      return feet;
33  }
```

```
34  float getInches()
35  {
36     return inches;
37  }
38};
39void main()
40{
41
42   Length objLength1;
43   Length objLenngth1(5, 6.5);
44   objLength1 = objLength1.addLength(objLength2);
45   cout << objLenth1.getFeet() << ' ';
46   cout << objLength1.getInches() << ' ';
47}
```

Solution:
Just write the main function like this:

```
1  #include<iostream.h>
2
3  void main()
4  {
5      Length objLength1;
6      Length objLength2(5,6.5);
7      objLength1=objLength1.addLenghth(objLenghth2);
8
9      cout<<objLength1.getFeet()<<"  ";
10     cout<<objLength1.getInches()<<"  ";
11}
```

5.6: **Identify the error in the following program**

```
1  #include <iosream.h>
2  class Room
3  void Area()
4  {
5     int width, height;
6     class Room
7     {
8         int width, height;
9         public:
10        void setvalue(int w, int h)
11        {
12            width = w;
13            height = h;
14        }
```

```
15      void displayvalues()
16      {
17         cout << (float)width << ' ' << (float)height;
18      }
19  };
20  Room objRoom1;
21  objRoom1.setValue(12, 8);
22  objRoom1.displayvalues();
23}
24
25void main()
26{
27Area();
28Room objRoom2;
29}
```

Solution:
Undefined structure Room in main ( ) function.
Correction : Change the main ( ) Function as follow:

```
1void main()
2{
3 Area();
4}
```

Programming Exercises

5.1: **Define a class to represent a bank account. Include the following members:**

Data members:

1. Name of the depositor.
2. Account number.
3. Type of account.
4. Balance amount in the account.

Member functions:

1. To assign initial values.
2. To deposit an amount.
3. To withdraw an amount after checking the balance.
4. To display the name and balance.

Write a main program to test the program.

Solution:

```
#include<iostream.h>
#include<iomanip.h>
class bank
{
    char name[40];
    int ac_no;
    char ac_type[20];
    double balance;
public:
   int assign(void);
   void deposite(float b);
   void withdraw(float c);
   void display(void);
};

int bank::assign(void)
{
   float initial;
   cout<<" You have to pay   500 TK to open your account \n"
   <<" You have to store at least 500 TK to keep your account active\n"
   <<"Would you want to open a account????\n"
   <<" If Yes press 1 \n"
   <<" If No press 0 : ";
   int test;
   cin>>test;
   if(test==1)
   {
      initial=500;
      balance=initial;
     cout<<" Enter name ,account number & account type to creat account : \n";
       cin>>name>>ac_no>>ac_type;
   }
   else
   ;// do nothing

   return  test;

}
void bank::deposite(float b)
{
   balance+=b;
}
void bank::withdraw(float c)
{
   balance-=c;
   if(balance<500)
   {
      cout<<" Sorry your balance is not sufficient to withdraw "<<c<<"TK\n"
```

```
49          <<" You have to store at least 500 TK to keep your account active\n";
50            balance+=c;
51   }
52}
53void bank::display(void)
54{
55   cout<<setw(12)<<"Name"<<setw(20)<<"Account type"<<setw(12)<<"Balance"<<endl;
56   cout<<setw(12)<<name<<setw(17)<<ac_type<<setw(14)<<balance<<endl;
57}
58
59int main()
60{
61   bank account;
62
63   int  t;
64   t=account.assign();
65   if(t==1)
66   {
67      cout<<" Would you want to deposite: ?"<<endl
68      <<"If NO press 0(zero)"<<endl
69      <<"If YES enter deposite ammount :"<<endl;
70      float dp;
71      cin>>dp;
72      account.deposite(dp);
73      cout<<" Would you want to withdraw : ?"<<endl
74      <<"If NO press 0(zero)"<<endl
75      <<"If YES enter withdrawal ammount :"<<endl;
76      float wd;
77      cin>>wd;
78      account.withdraw(wd);
79      cout<<" see details :"<<endl<<endl;
80      account.display();
81   }
82         else if(t==0)
83   cout<<" Thank you ,see again\n";
84   return 0;
85}
```

output

```
You have to pay 500 TK to open your account
You have to store at least 500 TK to keep your account active
Would you want to open a account????
If Yes press 1
If No press 0 : 0
Thank you ,see again
```

5.2: **Write a class to represent a vector (a series of float values). Include member functions to perform the following tasks:**

(a) To create the vector.

(b) To modify the value of a given element.

(c) To multiply by a scalar value.

(d) To display the vector in the form (10, 20, 30 …)

Write a program to test your class.

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  class vector
4  {
5      float *p;
6      int size;
7  public:
8      void creat_vector(int a);
9      void set_element(int i,float value);
10     void modify(void);
11     void multiply(float b);
12     void display(void);
13};
14
15void vector::creat_vector(int a)
16{
17    size=a;
18    p=new float[size];
19}
20void vector::set_element(int i,float value)
21{
22    p[i]=value;
23}
24void vector :: multiply(float b)
25{
26    for(int i=0;i<size;i++)
27        p[i]=b*p[i];
28}
29void vector:: display(void)
30{
31      cout<<"p["<<size<<"] = ( ";
32      for(int i=0;i<size;i++)
33      {
34          if(i==size-1)
```

```
35            cout<<p[i];
36          else
37          cout<<p[i]<<" , ";
38
39       }
40       cout<<")"<<endl;
41}
42
43void vector::modify(void)
44{
45    int i;
46    cout<<" to edit a given element enter position of the element : ";
47    cin>>i;
48    i--;
49    cout<<" Now enter new value of "<<i+1<<"th  element : ";
50    float v;
51    cin>>v;
52    p[i]=v;
53    cout<<" Now new contents : "<<endl;
54    display();
55
56    cout<<" to delete an element enter position of the element :";
57    cin>>i;
58    i--;
59
60    for(int j=i;j<size;j++)
61    {
62       p[j]=p[j+1];
63    }
64    size--;
65    cout<<" New contents : "<<endl;
66    display();
67}
68
69int main()
70{
71    vector santo;
72    int s;
73    cout<<" enter size of vector : ";
74    cin>>s;
75    santo.creat_vector(s);
76    cout<<" enter "<<s<<" elements  one by one :"<<endl;
77    for(int i=0;i<s;i++)
78    {
79       float v;
80       cin>>v;
81       santo.set_element(i,v);
82    }
83    cout<<" Now contents  :"<<endl;
84    santo.display();
85    cout<<" to multiply this vector by a scalar quantity enter this scalar quantity : ";
```

```
86    float m;
87    cin>>m;
88    santo.multiply(m);
89    cout<<" Now contents : "<<endl;
90    santo.display();
91    santo.modify();
92    return 0;
93}
```

output

```
enter size of vector : 5

enter 5 elements one by one :

11    22    33    44    55

Now contents p[5] = ( 11 , 22 , 33 , 44 , 55)

to multiply this vector by a scalar quantity enter this scalar quantity : 2

Now contents :

p[5] = ( 22 , 44 , 66 , 88 , 110)

to edit a given element enter position of the element : 3

Now enter new value of 3th element : 100

Now new contents :

p[5] = ( 22 , 44 , 100 , 88 , 110)

to delete an element enter position of the element :2

New contents :

p[4] = ( 22 , 100 , 88 , 110)
```

5.3: **Modify the class and the program of Exercise 5.1 for handling 10 customers.**

Solution:

```
1    #include<iostream.h>
2    #include<iomanip.h>
3    #define size 10
4    char *serial[size]={" FIRST "," SECOND "," THIRD "," 4th "," 5th "," 6th "," 7th "," 8th ","
```

```
5   9th ","10th"};
6
7   class bank
8   {
9      char name[40];
10     int ac_no;
11     char ac_type[20];
12     double balance;
13  public:
14     int assign(void);
15     void deposit(float b);
16     void withdraw(float c);
17     void displayon(void);
18          void displayoff(void);
19  };
20
21  int bank::assign(void)
22  {
23     float initial;
24     cout<<" You have to pay   500 TK to open your account \n"
25     <<" You have to store at least 500 TK to keep your account active\n"
26     <<"Would you want to open a account????\n"
27     <<" If Yes press 1 \n"
28     <<" If No press 0 : ";
29     int test;
30     cin>>test;
31     if(test==1)
32     {
33        initial=500;
34        balance=initial;
35  cout<<" Enter name ,account number & account type to create account : \n";
36          cin>>name>>ac_no>>ac_type;
37     }
38     else
39     ;// do nothing
40
41     return  test;
42
43  }
44  void bank::deposit(float b)
45  {
46     balance+=b;
47  }
48  void bank::withdraw(float c)
49  {
50     balance-=c;
51     if(balance<500)
52     {
53     cout<<" Sorry your balance is not sufficient to withdraw "<<c<<"TK\n"
54     <<" You have to store at least 500 TK to keep your account active\n";
55               balance+=c;
```

```
56     }
57  }
58  void bank::displayon(void)
59  {
60     cout<<setw(12)<<name<<setw(17)<<ac_type<<setw(14)<<balance<<endl;
61  }
62  void bank::displayoff(void)
63  {       cout<<" Account has not created"<<endl;  }
64  int main()
65  {
66     bank account[size];
67          int  t[10];
68     for(int i=0;i<size;i++)
69     {
70                  cout<<" Enter information for "<<serial[i]<<"customer : "<<endl;
71        t[i]=account[i].assign();
72                  if(t[i]==1)
73                  {
74            cout<<" Would you want to deposit: ?"<<endl
75            <<"If NO press 0(zero)"<<endl
76            <<"If YES enter deposit amount :"<<endl;
77            float dp;
78            cin>>dp;
79            account[i].deposit(dp);
80            cout<<" Would you want to with draw : ?"<<endl
81            <<"If NO press 0(zero)"<<endl
82            <<"If YES enter withdrawal amount :"<<endl;
83            float wd;
84            cin>>wd;
85            account[i].withdraw(wd);
86            cout<<endl<<endl;
87                  }
88                  else if(t[i]==0)
89                  cout<<"Thank  you , see again  \n";
90
91     }
92
93      cout<<" see details :"<<endl<<endl;
94      cout<<setw(12)<<"Name"<<setw(20)<<"Account type"
95                 <<setw(12)<<"Balance"<<endl;
96
97     for(i=0;i<size;i++)
98     {
99                    if(t[i]==1)
100          account[i].displayon();
101                  else  if(t[i]==0)
102                    account[i].displayoff();
103    }
104    return 0;
    }
```

Note: Here we will show output only for **Three** customers. But when you run this program you can see output for 10 customer.

output

```
Enter information for FIRST customer :
You have to pay 500 TR to open your account
You have to store at least 500 TR to keep your account active Would you want to open a account????
If Yes press 1
If No press 0 : 0
Thank you , see again
Enter information for SECOND customer :
You have to pay 500 TR to open your account
You have to store at least 500 TR to keep your account active Would you want to open a account????
If Yes press 1
If No press 0 : 1
Enter name ,account number & account type to create account :
Robin 11123 saving
Would you want to deposit: ?
If HO press 0(zero)
If YES enter deposit amount :
0
Would you want to with draw : ?
If HO press 0(zero)
If YES enter withdrawal amount :
0
Enter information for 3rd customer :
You have to pay 500 TK to open your account
You have to store at least 500 TK to keep your account active Would you want to open a account????
If Yes press 1
If No press 0 : 1
Enter name ,account number & account type to create account :
Billal 11123 fixed
Would you want to deposit: ?
If HO press 0(zero)
If YES enter deposit amount :
1000000
Would you want to with draw : ?
If HO press 0(zero)
If YES enter withdrawal amount :
100000

see details :

Name        Account type        Balance

Account has not created
```

Robin	saving	500
Billal	fixed	900500

5.4: **Modify the class and the program of Exercise 5.12 such that the program would be able to add two vectors and display the resultant vector. (Note that we can pass objects as function arguments)**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  #define size 8
4  class vector
5  {
6      float *p;
7
8  public:
9      void creat_vector(void);
10     void set_element(int i,float value);
11     friend void add(vector v1,vector v2);
12
13};
14void vector::creat_vector(void)
15{
16    p=new float[size];
17}
18void vector::set_element(int i,float value)
19{
20    p[i]=value;
21}
22void add(vector v1,vector v2)
23{
24
25    float *sum;
26    cout<<"sum["<<size<<"] = (";
27    sum= new float[size];
28
29    for(int i=0;i<size;i++)
30    {
31        sum[i]=v1.p[i]+v2.p[i];
32        if(i==size-1)
33            cout<<sum[i];
34        else
35            cout<<sum[i]<<" , ";
36    }
37    cout<<")"<<endl;
```

```
38
39}
40
41int main()
42{
43    vector x1,x2,x3;
44    x1.creat_vector();
45    x2.creat_vector();
46    x3.creat_vector();
47    cout<<" Enter "<<size<<" elements of FIRST vector : ";
48    for(int i=0;i<size;i++)
49    {
50        float v;
51        cin>>v;
52        x1.set_element(i,v);
53    }
54
55    cout<<" Enter "<<size<<" elements of SECOND vector : ";
56    for(i=0;i<size;i++)
57    {
58            float v;
59        cin>>v;
60        x2.set_element(i,v);
61    }
62    add(x1,x2);
63
64    return 0;
65}
```

output

Enter 8 elements of FIRST vector : 4 7 8 2 4 3 2 9

Enter 8 elements of SECOND vector : 1 2 3 4 5 6 7 8

sum[8] = (5 , 9 , 11 , 6 , 9 , 9 , 9 , 17)

5.5: **Create two classes DM and DB which store the value of distances. DM stores distances in meters and centimeters and DB in feet and inches. Write a program that can read values for the class objects and add one object of DM with another object of DB.**
Use a friend function to carry out the addition operation. The object that stores the results may be a DM object or DB object, depending on the units in which the results are required. The display should be in the format of feet and inches or meters and centimeters depending on the object on display.

Solution:

```
1  #include<iostream.h>
2  #define factor 0.3048
3  class DB;
4  class DM
5  {
6                   float d;
7          public:
8          void store(float x){d=x;}
9          friend void sum(DM,DB);
10         void show();
11};
12class DB
13{
14        float d1;
15        public:
16        void store(float y){d1=y;}
17        friend void sum(DM,DB);
18        void show();
19};
20
21void DM::show()
22{
23
24      cout<<"\n Distance = "<<d<<" meter or "<<d*100<<" centimeter\n";
25}
26
27void DB::show()
28{
29
30      cout<<"\n Distance = "<<d1<<" feet or "<<d1*12<<" inches \n";
31}
32void sum(DM m,DB b)
33{
34
35         float sum;
36
37         sum=m.d+b.d1*factor;
38         float f;
39         f=sum/factor;
40         DM m1;
41         DB b1;
42
43         m1.store(sum);
44         b1.store(f);
45
46       cout<<" press 1 to display result in meter\n"
47          <<" press 2 to display result in feet \n"
48          <<" What is your option ? : ";
49          int test;
50          cin>>test;
51
```

```
52          if(test==1)
53          m1.show();
54          else if(test==2)
55          b1.show();
56
57
58}
59
60
61int main()
62{
63     DM dm;
64     DB db;
65     dm.store(10.5);
66     db.store(12.3);
67    sum(dm,db);
68     return 0;
69}
```

output

```
Press 1 to display result in meter
Press 2 to display result in feet
What is your option ? 1
Distance = 14.24904 meter or 1424.903931 centimeter
```

Chapter 6

Review Questions

6.1: **What is a constructor? Is it mandatory to use constructors in a class?**

Ans:A constructor is a 'special' member function whose task is to initialize the object of its class. It is not mandatory to use constructor in a class.

6.2: **How do we invoke a constructor function?**

Ans:Constructor function are invoked automatically when the objects are created.

6.3: **List some of the special properties of the constructor functions.**

Ans:Special properties of the constructor functions: