Chapter-2
Class & Methods

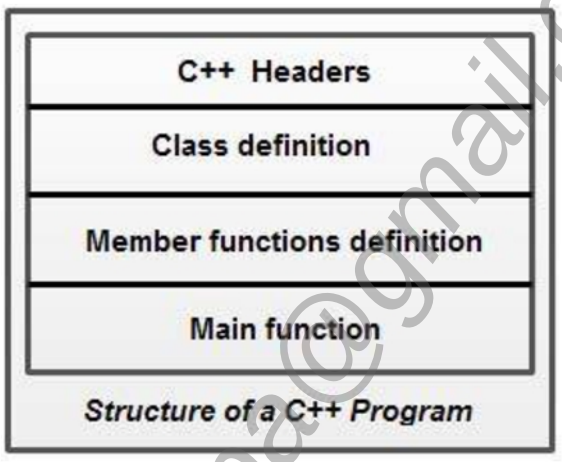
C++ Basic

Introduction to C++

- C++ is an object oriented programming language.
- It was developed by Bjarne Stroustrup at AT and T Bell Lab USA.
- It was developed on the base of C programming language and the first added part to C is concept of Class.
- Hence originally this language was named "C with class".
- Later name was changed to C++ on the basis of idea of increment operator available in C. since it is the augmented or incremented version of C, it is named C++.

Basic Program construction / C++ Structure

- C++ program structure is divided into various sections, namely, headers, class definition, member functions definitions and main function.
- A simple C++ program (without a class) includes comments, headers, main() and input/output statements as demonstrated below



1. **A simple C++ program to compute the area of rectangle in procedural approach.**

```
#include <iostream.h> //for cout and cin
#include <conio.h> //for getch()
void main()
{
        int length, breadth;
        int area; // declarations
        cout << "Enter the length of rectangle = "; // prompt user
        cin >> length; // enter length
        cout << "Enter the breadth of rectangle= "; // prompt user
        cin >> breadth; // input width
        area = length*breadth; // compute area
        cout << "Area is " << area<< endl; // output results
        getch();
} // end of main program
```

The following points should be noted in the above program:

1. Single line comment (// ……) comment i.e. Any text from the symbols // until the end of the line is ignored by the compiler. Also C comment type /*----------*/ is also a valid comment type in C++.
2. The line #include <iostream.h> must start in column one. It causes the compiler to include the text of the named file (in this case iostream.h) in the program at this point. The file iostream.h is a system supplied file which has definitions in it which are required if the program is going to use stream input or output. All programs will include this file. This statement is a preprocessor directive -- that is it gives information to the compiler but does not cause any executable code to be produced.
3. The actual program consists of the function main() which commences at the line void main( ). All programs must have a function main( ). Note that the opening brace ({) marks the beginning of the body of the function, while the

closing brace (}) indicates the end of the body of the function. The word void indicates that main() does not return a value. Running the program consists of obeying the statements in the body of the function main().

4. The body of the function main contains the actual code which is executed by the computer and is enclosed, as noted above, in braces { }.
5. Every statement which instructs the computer to do something is terminated by a semi-colon. Symbols such as main(), { } etc. are not instructions to do something and hence are not followed by a semi-colon. Preprocessor directives are instruction to the compiler itself but program statements are instruction to the computer.
6. Sequences of characters enclosed in double quotes are literal strings. Thus instructions such as

   cout << "Length = "

   send the quoted characters to the output stream cout. The special identifier endl when sent to an output stream will cause a newline to be taken on output.
7. All variables that are used in a program must be declared and given a type. In this case all the variables are of type int, i.e. whole numbers. Thus the statement

   int length, breadth;

   declares to the compiler that integer variables length and width are going to be used by the program. The compiler reserves space in memory for these variables.

2. A simple C++ program to compute the area of rectangle in Object Oriented approach.

```
#include <iostream.h>
#include <conio.h>
class Rectangle //Rectangle is Class Name
{
        private: //Access Specifier
        int length,breadth; //Data members
        public: //Access Specifier
        void setData() //member function
        {
                cout<<"Enter Length="<<endl
                cin>>length;
                cout<<"Enter breadth"<<endl;
                cin>>breadth;

        }
        void displayArea() //member function
        {
                int area;
                area= length * breadth;
                cout<<"Area of rectangle="<<area;
        }
};
void main()
{
        Rectangle rect1; // create object of Rectangle Class
        rect1.setData(); // we can invoke public members of Rectangle using object of Rectangle only.
        rect1.displayArea();
        getch();
}
```

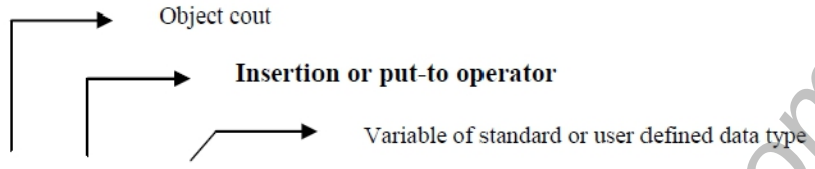

Input Output in C++

- C++ Supports rich set of functions for performing input and output operations.
- C++"s new features for handling I/O operations are called streams.
- Streams are abstractions that refer to data flow.
- Streams in C++ are:
  - Output Stream
  - Input Stream

Output Stream:

- The output stream allows us to write operations on output devices such as screen, disk etc.
- Output on the standard stream is performed using the cout object.
- C++ uses the bit-wise-left-shift operator for performing console output operation.
- The syntax for the standard output stream operation is as follows:

  cout<<variable;
- The word cout is followed by the symbol << , called the **insertion or put to** operator , and then with the items (variables, constants, expressions) that are to be output.
- Variables can be of any basic data types. The use of cout to perform an output operation is as shown :



cout << variable;

Examples:

- cout << "hello world";
- int age;
  cout<< age;
- float weight;
  cout<< weight;

- More than one item can be displayed using a single cout output stream object. Such output operations in C++ are called **cascaded output operations**. For example

  cout<<"Age is: "<<age<<"years";

  This cout object will display all the items from left to right. If value of age is 30 then this stream prints

  Age is : 30 years
- C++ does not restrict the maximum number of items to output. The complete syntax of the standard output streams operation is as follows:

  cout<<variable1<<vaariable2<<……<<variableN;
- The object cout must be associated with at least one argument.

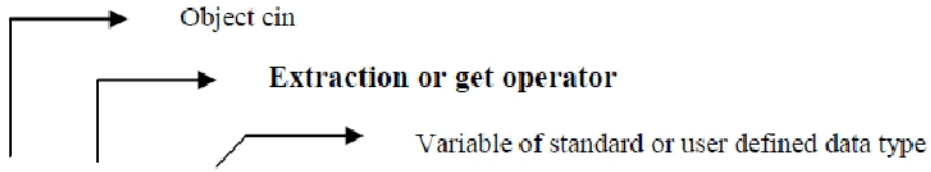
Example:

- cout<<"A"; //prints a constant character A
- cout<<10.99; //Prints constant 10.99
- cout<<" "; //prints blanks
- cout<<"\n"; //prints new line
- cout<<endl; line feed

Input Streams

- The input stream allows us to perform read operations with input devices such as keyboard, disk etc.
- Input from the standard stream is performed using the cin object.
- C++ uses the bit-wise right-shift operator for performing console input operation.
- The syntax for the standard output stream operation is as follows:

  Cin>>variable;
- The word cin is followed by the symbol >> and then with variable, into which input data is to be stored. The use of cin to perform an input operation is as shown :



cin>> variable;

Examples:

- int amount;
  cin>>amount;
- float weight;
  cin>>weight;
- char name[20];
  cin>>name;

- Input of more than one item can also be performed using the cin input stream object. Such input operation in C++ are called **cascaded input operations**.
- For example, reading the name of a person his address, age can be performed by the cin as:

  cin>> name>>address>>age;
- The cin object reads the items from left to right. The complete syntax of the standard input streams operations is as follows:

  cin>>var1>>var2>>var3>>……..>>varN;

  e.g. cin>>i>>j>>k>>l;
- The following are two important points to be noted about the stream operations.
  - Streams do not require explicit **data type specification** in I/O statement.
  - Streams do not require explicit **address operator** prior to the variable in the input statement.

- In C printf and scanf functions, format strings (%d,%s,%c etc) and address operator (&) are necessary but in cin stream format specification is not necessary and in the cout stream format specification is optional.
- **Format-free I/O** is special **features of C++** which make I/O operation comfortable.

The endl Manipulator

- The endl manipulator causes a linefeed to be inserted into the stream, so that subsequent text is displayed on the next line.
- It has same effect as sending the "\n" character but somewhat different.
- The difference is that endl flushes the output buffer, and '\n' doesn't. If we don't want the buffer flushed frequently, use '\n'. If we do, use endl.
- It is a manipulator that sends a newline to the stream and flushes the stream (puts out all pending characters that have been stored in the internal stream buffer but not yet output).

Example:

```
cout << "Perimeter is " << perimeter;
cout << endl << "Area is " << area << endl;
```

The Using Directive

- A **namespace** is designed to be used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different place within same program
- A namespace definition begins with the keyword namespace followed by the namespace name as follows

```
namespace namespace_name
{

    // code declarations
```

}.

- We use the using namespace directive that tells the compiler that the subsequent code is making use of names in the specified namespace.

Example-1: Defining class inside namespace

```
#include<iostream>
using namespace std;
namespace mynamespace
{
        class Test
        {
                public:
                void display()
                {
                        cout<<"hello world";
                }
        };
}
using namespace mynamespace;
int main()
{
        Test t;
        t.display();
}
```

In the above program, to access Test class we must first use the namespace **mynamespace as Test is** wrap up inside it. This can also be performed without making use of **using directive** by using fully qualified name with scope resolution operator(::) as below

```
#include<iostream>
namespace mynamespace
{
        class Test
        {
                public:
                void display()
                {
                        std::cout<<"hello world";// cout is object wrapped up inside std namespace
                }
        };
}
int main()
{
                mynamespace::Test t; // test is class wrapped up inside mynamespace namespace
                t.display();
}
```

Structures in C and C++

- A class is an extension of the idea of structure in C.
- It is a new way of creating and implementing **a user defined data type**.
- We know that structures provide a method for packing together data and of different types. A structure is a convenient tool for handling a group of logically related data items. Once structure has been defined, we can create variables of that type using declaration that are similar to built in data type declarations.

```
struct student
{
        char name[25];
        int age;
        float weight;
};
struct student s; // C style
or
student s; //C++ style
```

Limitation of Structure

i. The standard C does not allow the struct data type to be treated like built-in data types. For example

```
struct complex
{
        float real, imag;
};
struct complex c1,c2,c3;
```

The complex numbers c1,c2,c3 can easily be assigned values using the dot operator. But we cannot add two complex numbers or subtract one from the other. That is c3=c1 + c2; is illegal in C.

ii. Also data hiding is not permitted in C, that is structure members can be directly accessed by the structure variable by any function anywhere in their scope.

iii. C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user defined types as close as possible to the built in data types, and also provides a facility to hide the data.

iv. In c structure, it contains only variable or data as member but In C++, a class as well as structure can have both variables and functions as members. It can also declare some of its members as private so that cannot be accessed directly by the external functions.

v. The data member in c++ structure are public by default which may lead to security issue but in Class the default access specifier is private.

vi. Structure in C are only object based but Class support all features of OOP.

C++ Structure vs class

1. Structure is declared with struct keyword while class is declared with class keyword.
2. The data members in structures are public by default but in class the default access specifier is private.
3. Structure are only object based but class support all features of OOP.
4. Structure doesn't support inheritance but class supports inheritance.
5. In structure data member can't be initialized in structure definition but in class data member can be initialized in class definition.

Example:

Define a C structure named Student containing three members id, name and three variables to store different subject marks. WAP that would assign values to the individual members and display the entire detail along with total and percentage.

```
#include<stdio.h>
struct Student
{
        int id;
        char name[50];
        float math ;
        float science;
        float english;
};

int main()
{
        float tot,pct;
        struct Student s;
        printf("Enter the id");
        scanf("%d",&s.id);
        printf("\nEnter name");
        scanf("%s",s.name);
        printf("\nEnter marks in math, science and english");
        scanf("%f%f%f",&s.math,&s.science,&s.english);
        tot=s.math+s.science+s.english;
        pct=tot/3;
        printf("\nId= %d",s.id);
        printf("\nName= %s",s.name);
        printf("\nMarks in Math, sciene and english= %f%f%f ",s.math,s.science,s.english);
        printf("\nTotal obtained mark= %f",tot);
        printf("\nPercentage = %f",pct);
}
```

i. Define a C++ Structure named student containing three private data member id, name and address and two public member setData() and displayData() to input and display those details. Write a main program to test your structure.

Note: C structure can't define function but C++ structure can

```
#include<iostream>
using namespace std;
struct Student
{
        //These members are public by default
        void setData()
        {
                cout<<"Enter id,name and address";
                cin>>id>>name>>address;
        }

        void displayData()
        {
                cout<<"Id= "<<id<<endl;
                cout<<"Name= "<<name<<endl;
                cout<<"Address= "<<address<<endl;
        }

        private:
                int id;
                char name[50];
                char address[50];
```

```
};

int main()
{
        Student s; // This is error in C structure
        s.setData();
        s.displayData();

}
```

Specifying Class

- A class is user-defined data type that includes different member data and associated member functions to access on those data.
- Object-oriented programming (OOP) encapsulates data (attributes) and functions (behavior) into packages called classes.
- The data components of the class are called data members and the function components are called member functions. The data and functions of a class are intimately tied together.
- A class is like a blueprint.
- A programmer can create any number of objects of the same class.
- Classes have the property of information hiding. It allows data and functions to be hidden, if necessary, from external use.
- Since class is a collection of logically related data items and the associated functions which operate and manipulate those data. This entire collection can be called a new **user defined data-type.** So, classes are user-defined data types and behave like the built-in types of a programming language.
- Generally, a class specification has two parts.

    i. Variable declaration

    ii. Function definition

The general form of a class declarations is

```
class class-name
{
        private:
                Variable declarations;
                Function declarations;

        public:
                Variable declarations;
                Function declarations;
};
```

In the above syntax,

- The class declaration starts with a keyword "class" followed by a class-name. The class-name is an identifier.
- The body of the class is enclosed within braces and terminated by a semicolon.
- The functions and variables are collectively called class-members. The variables are, specially, called data members while the functions are called member functions.

- The two new keywords inside the above specification are – private and public. Those keywords are termed as access-specifiers, they are also termed as visibility labels. These are followed by colons(:)
- The class members that have been declared as private can be accessed only from within the class. i.e., only the member functions can have access to the private data members and private functions.
- On the other hand, public members can be accessed from anywhere outside the class (using object and dot operator).
- If no access specifier is specified, then the members are private by default.

Example of Class

```
#include<iostream>
using namespace std;
class Student
{
        int id;
        char name[20];
        char address[20];
        public:
                void setData()
                {
                        cout<<"Enter id, name and address";
                        cin>>id>>name>>address;
                }
                void displayData()
                {
                        cout<<"Id= "<<id<<endl;
                        cout<<"Name= "<<name<<endl;
                        cout<<"Address= "<<address<<endl;
                }
};
```

Access Specifiers and Data Hiding

- Access Specifiers are basically used to implement Data Hiding.
- Data Hiding in C++ Refers to restricting access to data members of a class from outside the class. This is required when we don't want other functions and classes to tamper with our class data. However, it is also important to make some member functions accessible so that the hidden data can be manipulated indirectly. This is where Access Specifiers come into the picture. It allows us to determine which class members are accessible to other classes and functions, and which class members are restricted from access.
- Access specifiers are the keyword that controls access to data member and member functions within user-defined class.
- The access restriction to the class members is specified by the labeled public, private, and protected sections within the class body.
- The keywords public, private, and protected are called access specifiers.

```
class Demo
{
  public:
  // public members go here
  protected:
  // protected members go here
```

```
  private:
  // private members go here
};
```

i. Public

A public member is accessible from anywhere outside the class but within a program.

ii. Private

A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members. By default all the members of a class are private.

iii. Protected

A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed by member in child classes and their friends.

Creating object

- An **Object** is an instance of a Class.
- When a class is defined, only the specification for the object is defined; no memory or storage is allocated.
- To use the data and access functions defined in the class, we need to create objects.
- So, once a class has been specified (or declared), we can create variables of that type (by using the class name as datatype).
- General syntax for declaration of object is

  Class_name Object_name

  Example:

  Student s1; //memory for s1 is allocated.
- The above statement creates a variable s1 of type Student. The class variables are known as objects. So, s1 is called object of class Test.
- We may also declare more than one object as follows.

  Student s1, s2, s3;
- When object is created, the necessary memory space is allocated to this object. The specification of class does not create any memory space for the objects.
- Objects can be created when a class is defined, as follows:

```
class Employee
{
        int id;
        char name[20];
        public:
        void getname();
        void putname();
}e1,e2,e3;
```

- The objects are also called **instances** of the class.
- In terms of object, a class can be defined as a collection of objects (or instances) of similar type.

Accessing Class Member

- The private data members of a class can be accessed only through the member functions of that class.
- However, the public members can be accessed from outside the class using object name and dot operator(.).
- The following is the format for calling public members of a class.

Object_name.Function_name(actual argument)

Object_name.Data_name

Example:

If e1 is the object of Class Employee then we can access the public member function getname() with the following statement

E1.getname();

Defining a member Function

- Member functions can be defined in two ways.
  i. Outside the class
  ii. Inside the class
- The code for the function body would be identical in both the cases. Irrespective of the place of definition, the function should perform the same task.

Outside the class

- In this approach, the member functions are only declared inside the class, whereas its definition is written outside the class.

General form:

```
return-type class-name::function-name(argument-list)
{
- - - - - - - - -
- - - - - - - - -- - //function body
- - - -- - -- --
}
```

- The function is, generally, defined immediately after the class-declaration. The function-name is proceeded by
  - return-type of the function
  - class-name to which the function belongs
  - symbol with double colons(::). This symbol is called the **scope resolution operator**. This operator tells the compiler that the function belongs to the class class-name.

Example:

```
void Employee::getdata()
{
- - - - - - - - -
- - - - - - - - -- - //function body
- - - -- - -- --
}
```

- In this example, the return-type is void which means the function "getdata()" doesn't return any value. The scope-resolution operator tells that the function "getdata()" is a member of the class "Employee". The argument list is also empty.
- The member functions have some special characteristics:
  i. A program may have several different classes. These classes can use same function name.
  ii. Member functions can directly access private data of that class. A non-member function cannot do so. (Exception is friend function which will be discussed later)
  iii. A member function can call another member function directly, without using the dot operator.

Inside the class

- Function body can be included in the class itself by replacing function declaration by function definition.
- If it is done, the function is treated as **an inline function**. Hence, all the restrictions that apply to inline function, will also apply here.

Example:

```
class Demo
{
        int a,b;
        public:
        void getdata()
        {
                cin>>a>>b;
        }
};
```

Here, getdata() is defined inside the class. So, it will act like an inline function.

Note:

- A function defined outside the class can also be made 'inline' simply by using the qualifier 'inline' in the header line of a function definition.

Example:

```
class Demo
{
- - - - - - - - -
- - - - - - - - - -
public:
        void getdata(); // function declaration inside the class
};

inline void Demo::getdata()
{
        //function body
}
```

Memory Allocation for objects

- The memory space for object is allocated when they are declared not when the class is specified.

- Also, while defining a class, the space is allocated for member functions only once and separate memory space is allocated for each object as shown in the below figure.
- It is because all the object belonging to a particular class uses same member functions so no separate memory space is allocated for member function but since the value of data member is different for different objects, so separate memory locations is used for each object.

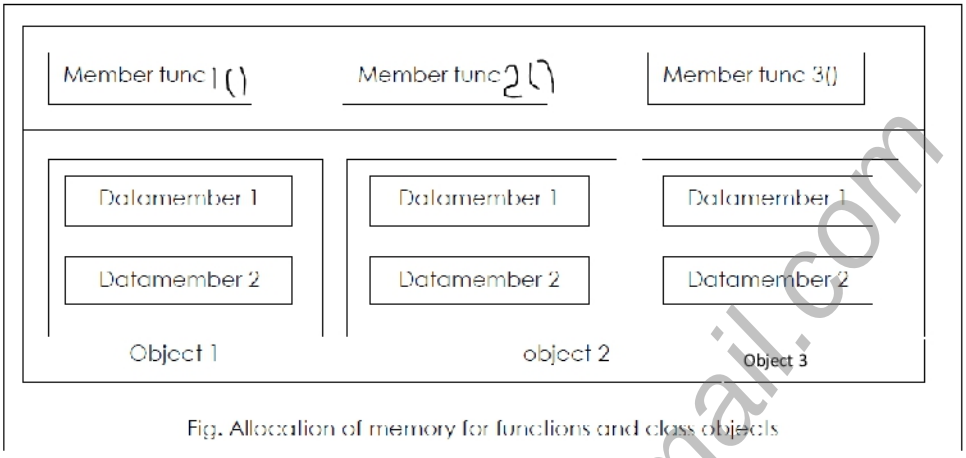
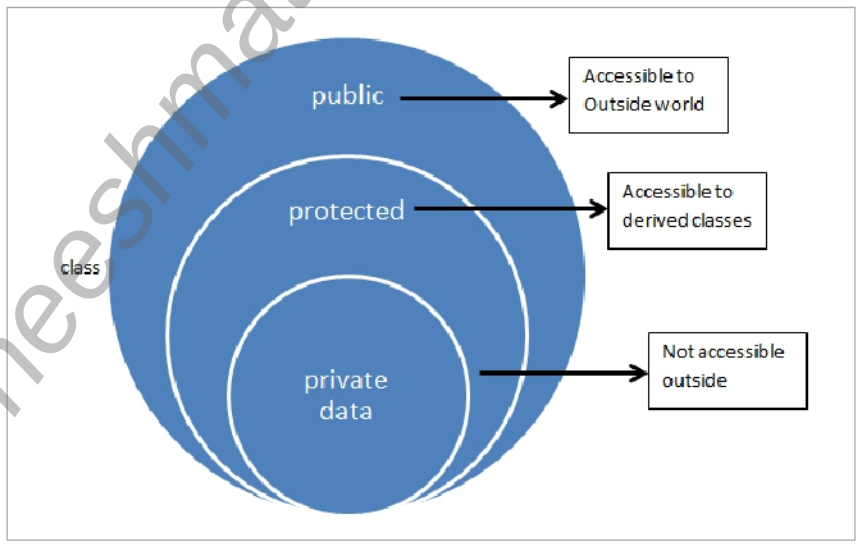


Fig. Allocation of memory for functions and class objects

Encapsulation and Data hiding

- Data encapsulation refers to the process of binding together data and functions or methods operating on this data into a single unit so that it is protected from outside interference and misuse.
- This is an important object-oriented programming concept and it leads to yet another OOP concept known as "**Data hiding**".
- Encapsulation hides data and its members whereas abstraction expose only the necessary details or interfaces to the outside world.



- A class in C++ is the one where we bundle together data members and the functions operating on these data members along with access specifiers like private, public and protected represent encapsulation.
- When we declare class members as private and methods to access class members as public we are truly implementing encapsulation.
- At the same time, we provide an abstract view of data to the outside world in the form of public methods.

Implementation of encapsulation

- Encapsulation in C++ is implemented as a class that bundles data and the functions operating on this data together. Mostly data is declared as private so that it is not accessible outside the class. The methods or functions are declared as public and can be accessed using the object of the class.
- However, we cannot directly access private members and this is called data hiding. When this is done, data is secured and can be accessed only by functions of that particular class in which the data is declared.

```
#include <iostream>
#include <string.h>
using namespace std;
class Accounts
{
        private:
                int empId;
                  double salary, basic, allowances, deductions;  // Private Data Member
                  //calculate salary of Employee
                  float calculateSalary() //Private Member Function
                  {
                          salary = basic+ allowances - deductions;
                          return salary;
                  }
         public:
                //set employee ID
                  void setEmployeeID(int eid)
                  {
                        empId=eid;
                  }
                  //input salary info of Employee
                  void inputEmployeeInfo()
                  {
                          cout<<"Enter basic Salary Amount for the employee"<<endl;
                          cin>>basic;
                          cout<<"Enter allowances Amount:"<<endl;
                          cin>>allowances;
                          cout<<"Enter Amout for any Deductions:"<<endl;
                          cin>>deductions;
                  }
                  //display Employee details
                  void displayEmployeeInfo()
                  {
                    salary = calculateSalary();
                    cout<<"Employee ID: "<<empId<<endl;
                    cout<<"Salary: "<<salary;
                  }

        };
int main()
{
         Accounts acc;
         acc.setEmployeeID(1);
         acc.inputEmployeeInfo();
         acc.displayEmployeeInfo();
}
```

```
Enter basic Salary Amount for the employee
2000
Enter allowances Amount:
500
Enter Amout for any Deductions:
300
Employee ID: 1
Salary: 2200
```

- As shown above, we have a class Accounts that bundles account data and all the functions that operate on this data into a single class Accounts. In the main function, we can create an object of this class and access functions to get the desired information.
- Now if some other classes say the employee details want to access accounts data, then it cannot do it directly. It will need to create an object of class Accounts and will be able to access only those items that are only public. This way, using encapsulation we guarantee the access control of the data and also ensure the integrity of data.

Example of C++ Program with Class

1. Create a class Rectangle with two data members (length and breadth), a function called readdata() to take detail of sides and a function called displaydata() to display area of rectangle. In main create two objects of a class Rectangle and for each object, call both of the function.

```
#include<iostream>
using namespace std;
class Rectangle
{
        float length,breadth;
        public:
                void readdata()
                {
                        cout<<"Enter lengtha and breadth of rectangle"<<endl;
                        cin>>length>>breadth;
                }
                void displaydata()
                {
                        float area;
                        area=length*breadth;
                        cout<<"Area of rectangle= "<<area<<endl;
                }
};
int main()
{
        Rectangle rect1,rect2;
        rect1.readdata();
        rect1.displaydata();
        rect2.readdata();
        rect2.displaydata();
}
```

Output:

```
Enter lengtha and breadth of rectangle
4
5
Area of rectangle= 20
Enter lengtha and breadth of rectangle
4.5
5.6
Area of rectangle= 25.2
```

2. Create a class Rectangle with two data members (length and breadth), a function called readdata() to take detail of sides and a function called displaydata() to display area of rectangle. The two functions are defined outside the class. In main create two objects of a class Rectangle and for each object, call both of the function.

```
#include<iostream>
using namespace std;
class Rectangle
{
        float length,breadth;
        public:
                void readdata();
                void displaydata();
};

void Rectangle::readdata()
{
        cout<<"Enter lengtha and breadth of rectangle"<<endl;
        cin>>length>>breadth;
}
void Rectangle::displaydata()
{
        float area;
        area=length*breadth;
        cout<<"Area of rectangle= "<<area<<endl;
}
int main()
{
        Rectangle rect1,rect2;
        rect1.readdata();
        rect1.displaydata();
        rect2.readdata();
        rect2.displaydata();
}
```

3. Create a class called Quadratic having the fields a, b, c and three function members called setData() to assign value to these fields from user and displyRoots() to display the real roots of quadratic equation. Write a main program to test your class.

```
#include<iostream>
#include<math.h>
using namespace std;
class Quadratic
{
        float a, b, c;
        public:
        void setData()
        {
```

```
                cout<<"Enter the coefficient a, b an c"<<endl;
                cin>>a>>b>>c;
        }
        void displayRoots()
        {
                double desc=pow(b,2)-4*a*c;
                double r1= (-b +sqrt(desc)) / ( 2 * a);
                double r2= (-b-sqrt(desc)) / ( 2 * a);
                if(desc<0)
                {
                        cout<<"Roots are imaginary"<<endl;
                }
                else if(desc==0)
                {
                        cout<<"Roots are equal"<<endl;
                        cout<<"Root1=Root2= "<<r1<<endl;

                }
                else
                {
                        cout<<"Roots are distinct"<<endl;
                        cout<<"Root1= "<<r1<<endl;
                        cout<<"Root2= "<<r2<<endl;
                }
        }
};

int main()
{
                Quadratic q;
                q.setData();
                q.displayRoots();
}
```

4. Define a class to represent a bank account. Include the following members

   Data Member:

   i. Name of account holder

   ii. Account Number

   iii. Account Type

   iv. Balance Amount

   Member Functions:

   i. to take detail of the fields from user

   ii. To deposit an amount

   iii. To withdraw an amount after checking balance

   iv. To display name and balance

```
#include<iostream>
using namespace std;
class Account
{
        char name[50],acctype[20];
```

```
        int accno;
        float bal=0;
        public:
                void setdata()
                {
                        cout<<"Enter name of account holder"<<endl;
                        cin>>name;
                        cout<<"Enter the type of account"<<endl;
                        cin>>acctype;
                        cout<<"Enter the account number"<<endl;
                        cin>>accno;
                }
                void deposit()
                {
                        float dep;
                        cout<<"Enter amount to be deposited"<<endl;
                        cin>>dep;
                        bal=bal+dep;
                        cout<<"Amount Deposited Successfully";
                }



                void withdraw()
                {
                        float wdraw;
                        cout<<"Enter amount to be withdraw"<<endl;
                        cin>>wdraw;

                        if(wdraw>bal)
                        {
                                cout<<"Insufficient Balance";
                        }
                        else
                        {
                                bal=bal-wdraw;
                                cout<<"Amount Withdrawn successfully";
                        }
                }
                void display()
                {
                        cout<<"Name= "<<name<<endl;
                        cout<<"Balance= "<<bal;
                }
};
int main()
{
        Account a;
        a.setdata();
        a.deposit();
        a.withdraw();
        a.display();
}
```

5. Create a class called Sum with two data members (fn and sn), a function called setdata(int , int) to assign value to fields and a function called int getsum() to calculate **sum of two numbers**. The first functions is defined inside the class and second is defined outside the class. Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Sum
{
        float fn,sn;
        public:
        void setdata(float a,float b)
        {
                fn=a;
                sn=b;
        }
        int getsum();
};
int Sum:: getsum()
{
        float area;
        area=fn+sn;
        return area;
}

int main()
{
                Sum s;
                float a,b,res;
                cout<<"Enter two numbers"<<endl;
                cin>>a>>b;
                s.setdata(a,b);
                res=s.getsum();
                cout<<"The sum of two number= "<<res;
}
```

6. Write a member function reverseit() that reverses a string passed as argument. Use for loop to swap the characters.

```
#include<iostream>
#include<string.h>
using namespace std;
class String
{
        public:
        void reverseit(char data[])
        {
                int length;
                length=strlen(data);
                char rev[100];
                int j=0;
                int i=0;
                for(int i=length-1;i>=0;i--)
                {
                        rev[j]=data[i];
                        j++;
                }
```

```
                rev[j]='\0';
                cout<<"Reversed String= "<<rev;
        }
};
int main()
{
        char str[100];
        cout<<"Enter a string";
        cin.getline(str,100);
        String s;
        s.reverseit(str);
}
```

7. Create a class Demo with one array field data of size 10 and two methods named setdata() to input 10 values from user and displaylarge() to display the largest value in the array. Also write a main program to test your class.

```
#include<iostream>
using namespace std;
class Demo
{
        int data[10];
        public:
                void setdata()
                {
                        cout<<"Enter 10 numbers"<<endl;
                        for(int i=0;i<10;i++)
                        {
                                cin>>data[i];
                        }
                }
                void displaylarge()
                {
                        int large=data[0];
                        for(int i=0;i<10;i++)
                        {
                                if(data[i]>large)
                                {
                                        large=data[i];
                                }
                        }
                        cout<<"Largest number= "<<large;
                }

};
 int main()
 {
        Demo d;
         d.setdata();
         d.displaylarge();
 }
```


Function in c++

- A function is a group of statements that together perform some specific task.
- Every C++ program has at least one function, which is main ().
- A function is also known as with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
{
   // body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

i. Return Type: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

ii. Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.

iii. Parameters: When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

iv. Function Body: The function body contains a collection of statements that define what the function does.

Example:

```
int max(int a, int b)
{
        if(a>b)
        {
                return (a);
        }
        else
        {
                return(b);
        }
}
```

Example:

1. **WAP to find the greatest among two number entered by user using the function int max()**
   i. **Using POP**

```
#include <iostream>
using namespace std;
int max();//Function Prototyping
int main()
{
        int res;

        res=max();//Function Calling
        cout<<"Largest Number="<<res;
}
int max()  //Function Defination
{
        int num1,num2, result;
        cout<<"Enter two number"<<endl;
        cin>>num1>>num2;
        if (num1 > num2)
        result = num1;
        else
        result = num2;
        return result;
```

```
}
```

ii. Using OOP

```
#include <iostream>
using namespace std;
class Compare
{
        int num1, num2;
        public:
                void setData(int n1, int n2)
                {
                        num1=n1;
                        num2=n2;
                }
                int max()
                {
                        int result;
                        if (num1 > num2)
                        result = num1;
                        else
                        result = num2;
                        return result;
                }
};
int main()
{
        int num1,num2,res;
        Compare c;
        cout<<"Enter two number"<<endl;
        cin>>num1>>num2;
        c.setData(num1,num2);
        res=c.max();
        cout<<"Largest Number="<<res;
}
```

2. **WAP to find the greatest among two number entered by user using the function int max(int,int)**
   See class Notes…

Function Overloading

- C++ allows us to specify more than one definition for a function name in the same scope, which is called function overloading.
- An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments or signature and obviously different definition (implementation).
- When you call an overloaded function or operator, the compiler determines the most appropriate definition to use, by comparing the **argument types** and **arguments number** we have used to call the function with the parameter types and number specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.

Example:
Following is the example where same function display() is being used to print different data types. Here function overloading is done based on **different types of argument.**

```
#include <iostream>
using namespace std;
class Demo
{
	public:
	void display(int i)
	{
		cout << "Printing int: " << i << endl;
	}

	void display(double f)
	{
		cout << "Printing float: " << f << endl;
	}

	void display(char* c)    // or void display (char [])
	{
		cout << "Printing character: " << c << endl;
	}
};

int main(void)
{
	Demo d;
	// Call print to print integer
	d.display(5);
	// Call print to print float
	d.display(500.263f);
	// Call print to print character
	d.display("Hello C++");
	return 0;
}
```

Output:

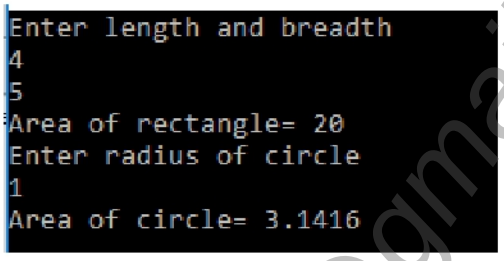
```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

WAP to find the area of circle and rectangle using the concept of function / method overloading (based on number of arguments)

```
#include<iostream>
using namespace std;
class Demo
{
	public:
	double area(double r)
	{
		return (3.1416 * r * r);
	}

	double area(double l, double b)
	{
		return (l*b);
	}
```

```
};
void main()
{
        Demo d;
        double l,b,r,aor,aoc;
        cout<<"Enter length and breadth"<<endl;
        cin>>l>>b;
        aor=d.area(l,b);
        cout<<"Area of rectangle= "<<aor<<endl;
        cout<<"Enter radius of circle"<<endl;
        cin>>r;
        aoc=d.area(r);
        cout<<"Area of circle= "<<aoc;

}
```

Output:



Inline Function

- Function call is a costly operation. During the function call it's execution our system takes overheads like: Saving the values of registers, Saving the return address, Pushing arguments in the stack, Jumping to the called function, Loading registers with new values, Returning to the calling function, and reloading the registers with previously stored values.
- For large functions this overhead is negligible but for small function taking such large overhead is not justifiable.
- To solve this problem concept of inline function is introduced in c++.
- The functions which are **expanded inline** by the compiler each time it's call is appeared instead of jumping to the called function as usual is called inline function.
- When a function is defined as inline, compiler **copies it's body where the function call is made**, instead of transferring control to that function.
- A function is made inline by using a keyword "inline" before the function definition.

Inline functions provide following advantages:

1) Function call overhead doesn't occur.

2) It also saves the overhead of push/pop variables on the stack when function is called.

3) It also saves overhead of a return call from a function.

Inline function disadvantages:

1) The added variables from the inlined function consumes additional registers memory.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Inline function may increase compile time overhead.

4) Not suitable for those system where code size is important than speed. Example, Embedded System.

5) The compiler can't perform inlining if the function is too complex i.e. function can't be expanded if it contains loops or recursion.s

Example:

Write a program to find the cube of given integer using inline function.

```
#include<iostream.h>
#include<conio.h>
inline void cube(int);
void main()
{
        int x;
        cout<<"Enter a number:"<<endl;
        cin>>x;
        cube(x);
        getch();
}

inline void cube(int a)
{
        int cube;
        cube=a*a*a;
        cout<<"Cube="<<cube<<endl;
}
```

Making a class member function Inline

- Function body can be included in the class itself by replacing function declaration by function definition.
- If it is done, the function is treated as **an inline function**. Hence, all the restrictions that apply to inline function, will also apply here.

Example:

```
class Demo
{
        int a,b;
        public:
        void getdata()
        {
                cin>>a>>b;
        }
};
```

Here, getdata() is defined inside the class. So, it will act like an inline function.

- A function defined outside the class can also be made 'inline' simply by using the qualifier 'inline' in the header line of a function definition.

Example:

```
class Demo
{
- - - - - - - - - -
- - - - - - - - - - -
public:
        void getdata(); // function declaration inside the class
};
```

```
inline void Demo::getdata()
{
        //function body
}
```

Example:

```
Class Demo
{
        public void func(); // Function declaration
};

inline void Demo::func()  //Function defination
{

}
```

WAP to display hello world by using class and inline member function

```
#include<iostream>
using namespace std;
class Demo
{
        public:
                void display();
};
 int main()
{
        Demo d;
         d.display();
}
 inline void Demo:: display()
{
        cout<<"hello world";
}
```

Default Arguments to a function

- Arguments(s) of a function is (are) the data that the function receives when called/invoked from another function.
- C++ allows a function to assign a parameter the default value in case no argument for that parameter is specified in the function call.
- When declaring a function, we can specify a default value for each of the last parameters which are called default arguments.
- This value will be used if the corresponding argument is left blank when calling to the function.
- The default value is specified similar to the variable initialization.
- The prototype for the declaration of default value of an argument looks like

```
float amount (float p, int time, float rate=0.10);
// declares a default value of 0.10 to the argument rate.
```

- The call of this function as

value = amount (4000, 5);// one argument missing for rate, it passes the value 4000 to p , 5 to time and the function looks the prototype for missing argument that is declared as default value 0.10 the function uses the default value 0.10 for the third argument.

- But the call

value = amount (4000,5,0.15);

no argument is missing, in this case function uses this value 0.15 for rate.

Note : only the trailing arguments can have default value. We must add default from right to left. E.g.

```
int add( int a, int b =9, int c= 10 ); // legal
int add(int a=8, int b, int c); // illegal
int add(int a, int b = 9, int c); //illegal
int add( int a=8, int b=9,int c=10) // legal
```

Example 1:

```
#include<iostream>
void display(int u=7);
using namespace std;

int  main ()
{
        display(5); // function call with argument
        display(); // function call with no argument
}
void display(int u)  // the default value for U is 7 if no argument is passed
{
        cout<<u<<endl;
}
```

Output:

```
5
7
```

Example2:

```
#include<iostream>
using namespace std;
void display(int U=7)  // the default value for U is 7 if no argument is passed
{
        cout<<U<<endl;
}

int  main ()
{
        display(5); // function call with argument
        display(); // function call with no argument
}
```

output:

```
5
7
```

Example 3:

WAP using class to find the simple interest where rate is 10% by default.

```
#include <iostream>
using namespace std;
class Interest
{
        float pri,time,rate;
        public:
                void setData(float p, float t, float r=10 )
                {
                        pri=p;
                        time=t;
                        rate=r;
                }
                float getInterest()
                {
                        return((pri * time * rate)/100);
                }
};
int main()
{
        float interest;
        Interest i;
        i.setData(100,1);//set p=100, t=1 and rate to default  10%
        interest=i.getInterest();
        cout<<"Interest Amount="<<interest;

}
```

Private Member Function

- As we have seen, member functions are, in general, made public. But in some cases, we may need a private function to hide them from outside world.
- Private member functions can only be called by another function that is a member of its class.
- Object of the class cannot invoke private member functions using dot operator.

Example:

```
class Demo
{
        int a,b;
        void func1(); //private member by default
        public:
        void func2();
};
int main
{
        Demo d;
        d.func1();// func1() is private member so can't be accessed by using object of class Demo
}
```

Question:

Define a class Student with the following specification

Private members:

Data member: Id, name, eng, math, science

Member function: float calculateTotal() to return the total obtained mark.

Public members:

Member functions:

getData() to take input(id, name, eng, math, science) from user showData() to display all the data member along with total on the screen.

Write a main program to test your class.

```
#include<iostream>
using namespace std;
class Student
{
        int id;
        char name[50];
        float eng,math,science;
        float calculateTotal()
        {
                return(eng+math+science);
        }
        public:
        void inputData()
        {
                cout<<"Enter id"<<endl;
                cin>>id;
                cout<<"Enter name"<<endl;
                cin>>name;
                cout<<"Enter marks in maths, science and english"<<endl;
                cin>>math>>science>>eng;
        }

        void showData()
        {
                float t;
                cout<<"ID= "<<id<<endl;
                cout<<"Name ="<<name<<endl;
                cout<<"Marks in maths, science and english ="<<math<<science<<eng<<endl;
                t=calculateTotal(); // Within the class scope, we can call members directly with their name
                cout<<"Total= "<<t;
        }

};
int main()
{
        Student s;
        s.inputData();
        s.showData();
}
```


Pointers

- A pointer is a variable that contains a memory address of data or another variable.
- Pointers are used in C++ program to access the memory and manipulate the address.
- Each pointer variable can point only to a specific type i.e. int, char, float etc.
- Pointer can have any name that is legal for other variable and it is declared in the same fashion like other variable but is always preceded by * (Asterisk) operator.

Pointer Variable Declaration and Initialization

- Pointer variable can be declared as below.

  Data_type * Variable_name

  Example:

  Declaring a pointer variable p is a pointer to int, i.e., it is pointing to an integer value in the memory address.

  ```
  Int *P;
  ```

  It means, P is a pointer variable and it can store an address of integer variable.

- Address of some variable can be assigned to a pointer variable at the time of declaration of pointer variable.

Example:

```
int n;
int *p=&n;
```

- The above statements are equivalent to following statements

```
int n;
int *p
p=&n;
```

WAP to find the address of given variable

```
#include <iostream>
using namespace std;
int main()
{
        int var1 = 3;
        int var2 = 24;
        int var3 = 17;
        cout << &var1 << endl;
        cout << &var2 << endl;
        cout << &var3 << endl;
}
```

Note: if var is a variable then &var gives its address.

Operators in Pointers

i. Address of Operator or Reference operator (&)

ii. Value at the address operator or Indirection or Deference operator (*)

- Reference operator (&) gives the address of a variable.
- To get the value stored in the memory address, we use the dereference operator (*).
- **For example**: If a number variable is stored in the memory address **0x123**, and it contains a value **5**. The **reference (&)** operator gives the value **0x123**, while the **dereference (*)** operator gives the value **5**.

C++ Program to demonstrate the concept of pointer

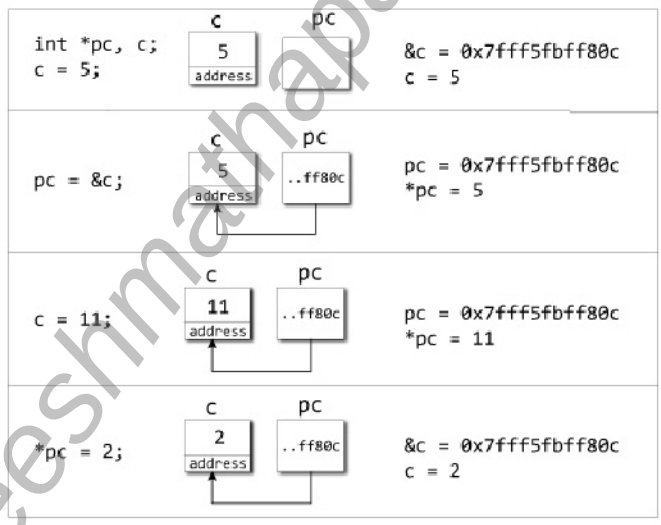
```
#include <iostream>
using namespace std;
int main()
{
        int *pc, c;

        c = 5;
        cout << "Address of c=" << &c << endl;
        cout << "Value of c=" << c << endl << endl;
```

```
    pc = &c;   // Pointer pc holds the memory address of variable c
    cout << "Address that pointer pc holds= "<< pc << endl;
    cout << "Content of the address pointer pc holds =" << *pc << endl;

    c = 11;   // The content inside memory address &c is changed from 5 to 11.
        cout << "Address that pointer pc holds= "<< pc << endl;
    cout << "Content of the address pointer pc holds =" << *pc << endl;

    *pc = 2;
    cout << "Address of c= " << &c << endl;
    cout << "Value of c=" << c << endl;

    return 0;
}
```

Output:

```
Address of c=0x6ffe34
Value of c=5

Address that pointer pc holds= 0x6ffe34
Content of the address pointer pc holds =5
Address that pointer pc holds= 0x6ffe34
Content of the address pointer pc holds =11
Address of c= 0x6ffe34
Value of c=2
```

The below figure shows the pictorial explanation.



Reference Variable

- Reference variable us a new kind of variable introduced in c++ that provides an alternate name of already existing variable. When a variable is declared as a reference, it become and alternative name for an existing variable.
- It cannot be changed to refer another variable and should be initialized at the time of declaration and cannot be NULL.
- The operator '&' is used to declare reference variable.

  Syntax:

  datatype variable_name; // variable declaration

datatype& refer_var = variable_name; // reference variable

Here,

datatype – The datatype of variable like int, char, float etc.

variable_name – This is the name of variable given by user.

refer_var – The name of reference variable.

Example:

```
#include <iostream>
using namespace std;
int main()
{
        int a = 8;
        int& b = a;
        cout << "The variable a : " << a;
        cout << "The reference variable r : " << b;
}
```

```
The variable a : 8
The reference variable r : 8
```

Pointer variable vs Reference Variable

1. A pointer is a variable which stores the address of another variable. A reference is a variable which refers to another variable.
   Example:
   ```
   int i = 3;
   int *ptr = &i;
   int &ref = i
   ```
   The first line simply defines a variable. The second defines a pointer to that variable's memory address. The third defines a reference to the first variable so that we can use variable or i interchangeably. Note int the statement int & ref, & is not an address operator, it means reference to int.

2. While using pointer variable we must make * operator to deference it but in case of reference variable no such operator is required.
   Example:
   ```
   int i = 3;
   int *ptr = &i;
   int &ref = i
   cout<<*ptr<<endl;
   cout<<ref;
   ```
   output:
   ```
   3
   3
   ```

3. Pointers can be reassigned while references cannot. In other words, a pointer can be assigned to a different address.
   Example:
   ```
   int i = 3,j=5;;
   int *ptr=&i;
   int &ref=i;
   ```

```
cout<<*ptr<<endl;
cout<<ref<<endl;
ptr=&j; //pointers can be reassigned
cout<<*ptr;
//&ref=j; this statement is wrong because reference can't be reinitialized
```

4. A reference variable must be initialized at the time of declaration but the pointer variable can be declared and initialized separately.
   Example:

```
int i = 3;
int *ptr;
ptr=&i; // it is allowed

int &ref; // not allowed
ref=i; //not allowed
```

5. Reference variable are easier, safer to use than pointer variable, however they are less flexible than pointers.
6. Pointers can point nowhere (NULL), whereas reference always refer to some object.

Pointer Object

- We can create pointers pointing to a class by using the class name as type of pointer.
- Example:

  Demo *d;

  In the above statement, d is pointer to an object of class Demo.
- We should use new operator to allocate memory for the object of Demo class as given below.

  d=new Demo();
- New operator is used to allocate memory at runtime. The new operator allocate memory and returns a pointer to start of it.
- Also we can combine above two statements into single statements as

  Demo *d = new Demo();
- Similar to structure, we have to use arrow operator(→) to access the class Member by pointer to an object.

Example:s

WAP to demonstrate the concept of pointer object.

```
#include<iostream>
using namespace std;
class Demo
{
    private:
        int n1,n2;
    public:
        void setData()
        {
            cout<<"Enter two numbers";
            cin>>n1>>n2;
        }
        void displaySum()
        {
            int sm;
            sm=n1+n2;
            cout<<"Sum of two number="<<sm;
```

```
        }
};

int main()
{
        Demo *d=new Demo();//pointer object
        d->setData();
        d->displaySum();
}
```

Function: Calling Function by value, calling function by reference, calling function by pointer

- There are basically three techniques to pass arguments to the function.
    i. Pass by value and
    ii. pass by reference
    iii. pass by pointer
- In function **call by value**, the called function creates a new set of variables and copies the value of arguments into them. The function doesn't have access to the actual variable in the calling program and can work on the copies of the values.

Example: WAP to demonstrate passing arguments to the function using pass by value.

```
#include <iostream>
using namespace std;
int max(int, int);

int main ()
{
        int a, b, ret;
        cout<<"Enter two numbers";
        cin>>a>>b;
        ret = max(a, b);
        cout << "Max value is : " << ret << endl;
}

int max(int num1, int num2)
{
        int result;
        if (num1 > num2)
          result = num1;
        else
          result = num2;
        return result;
}
```

In the above program, the value of variable a and b is passed by value to variable num1 and num2. Hence any modification done to variable num1 and num2 will not affect the calling variable a and b.

- The **call by reference method** of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

Example: WAP to demonstrate function call by reference.

```
#include <iostream>
using namespace std;
int max( int &, int &);

int main ()
{
  int a, b, ret;
  cout<<"Enter two numbers";
  cin>>a>>b;
  ret = max(a, b);
  cout << "Max value is : " << ret << endl;
}

int max(int &num1, int &num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

In the above program, during the function call the value of variable a and b are passed by reference to reference variable num1 and num2. num1 and num2 references variable a and b respectively. Hence any modification done to variable num1 and num2 will affect the calling variable a and b respectively.

- Working principle of **call by pointer** is same as that of pass by reference. In this case instead of using the reference variable, we use pointer variable in function definition and pass address of variable from function call statement.

Example: WAP to demonstrate function call by pointer

```
#include <iostream>
using namespace std;
int max( int *, int *);

int main ()
{
  int a, b, ret;
  cout<<"Enter two numbers";
  cin>>a>>b;
  ret = max(&a, &b);
  cout << "Max value is : " << ret << endl;
}

int max(int *num1, int *num2)
{
```

```
  int result;
  if (*num1 > *num2)
    result = *num1;
  else
    result = *num2;
  return result;
}
```

Object as function Argument

- Unlike other argument, we can also pass object as function argument using following three ways.

i. Pass by value:

A copy of entire object is passed to the function. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. This technique is relatively slower than other technique because it requires copying all the value of objects into the arguments of method invoked.

ii. Pass by reference:

Only the address of the object is transferred to the function. Since the actual address of object is passed, any changes made to the object inside the function will reflect in the actual object used to call the function. This method is more efficient than previous one.

iii. Pass by pointer

It is similar to pass by reference; the only difference is that in pass by reference we used reference variable in function definition but in pass by pointer we use pointer variable.

Example:

WAP to perform the addition of time in hour and minute format using the concept of object as function arguments(Pass by value)

```
#include<iostream>
using namespace std;
class Time
{
        int hour,min;
        public:
                void inputTime()
                {
                        cout<<"Enter hour"<<endl;
                        cin>>hour;
                        cout<<"Enter minute"<<endl;
                        cin>>min;
                }
```

```
        void displayTime()
        {
            cout<<"Hour= "<<hour<<"\t";
            cout<<"Minute= "<<min<<endl;
        }
        void addTime(Time a, Time b)
        {
            min=a.min +b.min;
            hour=min/60;
            min=min%60;
            hour=hour +a.hour+b.hour;
        }
};
int main()
{
    Time t1,t2,t3;
    t1.inputTime();
    t2.inputTime();
    t3.addTime(t1,t2);
    t1.displayTime();
    t2.displayTime();
    t3.displayTime();
}
```

Output:

```
Enter hour
1
Enter minute
30
Enter hour
1
Enter minute
30
Hour= 1 Minute= 30
Hour= 1 Minute= 30
Hour= 3 Minute= 0
```

WAP to perform the addition of time in hour and minute format using the concept of object as function arguments(Pass by reference)

```
#include<iostream>
using namespace std;
class Time
{
    int hour,min;
    public:
```

```
        void inputTime()
        {
                cout<<"Enter hour"<<endl;
                cin>>hour;
                cout<<"Enter minute"<<endl;
                cin>>min;
        }
        void displayTime()
        {
                cout<<"Hour= "<<hour<<"\t";
                cout<<"Minute= "<<min<<endl;
        }
        void addTime(Time &a, Time &b)
        {
                min=a.min +b.min;
                hour=min/60;
                min=min%60;
                hour=hour +a.hour+b.hour;
        }
};
int main()
{
        Time t1,t2,t3;
        t1.inputTime();
        t2.inputTime();
        t3.addTime(t1,t2);
        t1.displayTime();
        t2.displayTime();
        t3.displayTime();
}
```

WAP to perform the addition of time in hour and minute format using the concept of object as function arguments(Pass by Pointer)

```
#include<iostream>
using namespace std;
class Time
{
        int hour,min;
```

```
        public:
                void inputTime()
                {
                        cout<<"Enter hour"<<endl;
                        cin>>hour;
                        cout<<"Enter minute"<<endl;
                        cin>>min;
                }
                void displayTime()
                {
                        cout<<"Hour= "<<hour<<"\t";
                        cout<<"Minute= "<<min<<endl;
                }
                void addTime(Time *a, Time *b)
                {
                        min=a->min +b->min;
                        hour=min/60;
                        min=min%60;
                        hour=hour +a->hour+b->hour;
                }
    };
    int main()
    {
            Time t1,t2,t3;
            t1.inputTime();
            t2.inputTime();
            t3.addTime(&t1,&t2);
            t1.displayTime();
            t2.displayTime();
            t3.displayTime();
    }
}
```


Returning Objects

- A function can take object as argument as well as return object.
- A function can return object using one of the three ways as below.

i. Return by value

In this technique, a copy of the object is returned back to the function call. Similar to pass by value, it makes program to work slow while working with large objects.

ii. Return by reference

In this technique, an address of object is returned back to the function call. Here we can't return local objects by references because local object goes out of scope when exiting from the function, thereby giving garbage result. It makes program to work faster while working with large objects as compared to return by value technique.

iii. Return by pointer

This technique is similar to return by reference i.e. an address of object is returned back to the function call. The difference is that instead of returning a reference object as in return by reference, here we return pointer object.

Example:

1. **Create a class Complex with two data member (x and y) for storing real and imaginary part of a complex number and three member functions named void input (int, int) to initialize x and y, Complex sum(Complex, Complex) to return the Complex object with sum, and void show() to display the sum of two complex number. Write a main program to test your class.**

[Call by value and Return By Value]

```
#include<iostream>
using namespace std;
class Complex
{
        int x,y;
        public:
        void input(int real,int img)
        {
                x=real;
                y=img;
        }
        Complex sum (Complex c1, Complex c2)
        {
                Complex c3;
                c3.x=c1.x+c2.x;
                c3.y=c1.y+c2.y;
                return c3;
        }
        void show()
        {
                cout<<"Real Part= "<<x<<endl;
                cout<<"Imaginary part= "<<y<<endl;
        }
};
int main()
{
        Complex c1,c2,c3;
        c1.input(5,6);
        c2.input(4,5);
        c3=c3.sum(c1,c2);
        cout<<"First Complex Number"<<endl;
        c1.show();
        cout<<"Second Complex Number"<<endl;
```

```
        c2.show();
        cout<<"Result Complex Number"<<endl;
        c3.show();
}
```

```
First Complex Number
Real Part= 5
Imaginary part= 6
Second Complex Number
Real Part= 4
Imaginary part= 5
Result Complex Number
Real Part= 9
Imaginary part= 11
```

2. **Create a class Complex with two data member (x and y) for storing real and imaginary part of a complex number and three member functions named void input(int ,int) to initialize x and y, Complex &sum(Complex &, Complex &) to return the Complex object with sum, and void show() to display the sum of two complex number. Write a main program to test your class.**

[Call by reference and Return By Reference]

```
#include<iostream>
using namespace std;
class Complex
{
        private:
        int x,y;
        public:
                void input(int real, int img)
                {
                        x=real;
                        y=img;
                }
                Complex &sum(Complex &fn, Complex &sn)
                {
                        Complex c;
                        c.x=fn.x+sn.x;
                        c.y=fn.y+sn.y;
                        return(c);
                }
                void show()
                {
                        cout<<"Real:"<<x<<endl;
                        cout<<"Complex:"<<y<<endl;
                }
};


int main()
{
        Complex c1,c2,c3;
```

```
        c1.input(4,5);
        c2.input(3,6);
        c3=c3.sum(c1,c2);
        cout<<"First Complex Number"<<endl;
        c1.show();
        cout<<"Second Complex Number"<<endl;
        c2.show();
        cout<<"Result Complex Number"<<endl;
        c3.show();
    }
```

3. Create a class Complex with two data member (x and y) for storing real and imaginary part of a complex number and three member functions named void input(int ,int) to initialize x and y, Complex *sum(Complex *, Complex *) to return the Complex object with sum, and void show() to display the sum of two complex number. Write a main program to test your class.

[Call by Pointer and Return By Pointer]

```
#include<iostream>
using namespace std;
class Complex
{
    private:
    int x,y;
    public:
        void input(int real, int img)
        {
            x=real;
            y=img;
        }
        Complex *sum(Complex *fn, Complex *sn)
        {
            Complex *c=new Complex();
            c->x=fn->x+sn->x;
            c->y=fn->y+sn->y;
            return(c);
        }
        void show()
        {
            cout<<"Real:"<<x<<endl;
            cout<<"Complex:"<<y<<endl;
        }
};
```

```
int main()
{
        Complex c1,c2,*c3;
        c1.input(4,5);
        c2.input(3,6);
        c3=c3->sum(&c1,&c2);
        cout<<"First Complex Number"<<endl;
        c1.show();
        cout<<"Second Complex Number"<<endl;
        c2.show();
        cout<<"Result Complex Number"<<endl;
        c3->show();
}
```

Static Data Members

- A data member of class that is qualified as static is called as static data members. A static member variable has following properties.
  - i. It is declared with keyword static.
  - ii. It is initialized to zero when the first object of its class is created. No other initialization is permitted.
  - iii. Only one copy of that member is created for the entire class and is shared by all objects of that class.
  - iv. It is visible only within the class, but its lifetime is the entire program.
- Basically, static variables are used to maintain value **common to all the objects.**

  Syntax:

  static data_type varliable_name;
- For example, static member can be used as a counter that records the occurrences of all the objects.

Example:

```
#include<iostream>
using namespace std;
class Demo
{
        static  int count;
        public:
                void check()
                {
                        count++;
                }
                void display()
                {
                        cout<<"Value ofCount="<<count<<endl;
                }
};

int Demo::count;
```

```
int main()
 {
                Demo a,b,c,d; // count will be initialzed to zero and each object will be using the same copy
of count because count is static
                a.check();
                b.check();
                c.check();
                d.check();
                a.display();
                b.display();
                c.display();
                d.display();

 }
```

Output:

```
Value ofCount=4
Value ofCount=4
Value ofCount=4
Value ofCount=4
```

Note:

- **The type and scope of each static member variable must be defined outside the class definition as below:**

  int Demp:: count;

  Since the static member are associated with the class itself rather than within any class object, they are also known as **class variable.** Class variable are stored separately rather than as part of object.

- what is the output if the static member count is define as below

  int Demo::count=7;

  Output:

```
Value ofCount=11
Value ofCount=11
Value ofCount=11
Value ofCount=11
```

- If the static qualifier is removed, what would be the output?
  i. If static is removed first of all we have to initialize the count by ourself.

     i.e. int count = 0;

  ii. Then the statement int Demo:: count must be removed because count is no more static.

  iii. Then we will see the following output.

```
Value ofCount=1
Value ofCount=1
Value ofCount=1
Value ofCount=1
```

Static Member function

- A member function that is declared static has the following property.
  i. A static function can have access to only other static data members and static member functions declared in the same class.

  ii. A static member function can be called using the class name instead of its object as follows.

Class_name:: function_name

iii. Static member functions doesn't have access to the 'this' pointer of the class.

iv. They can't be declared as virtual.

Example:

WAP to find the area of rectangle using the concept of static members.

```
#include<iostream>
using namespace std;
class Demo
{
        static int l,b;
        public:
                static void setData()
                {
                        cout<<"Enter length and breadth"<<endl;
                        cin>>l>>b;
                }
                static void displayArea()
                {
                        int area;
                        area=l*b;
                        cout<<"Area of rectangle"<<area;
                }
};

int Demo::l;
int Demo::b;

 int main()
 {
          Demo::setData();
         Demo::displayArea();
}
```

Note:

- We can also create object to access the static members

```
Demo x,y;
x.setData();
x.displayArea();
y.setData();
y.displayArea();
```

- Will we get desired output if we use following sequence of code? Demo x,y;

```
x.setData();
y.setData();
x.displayArea();
y.displayArea();
```

No, because, since l and b are static so it is common to both object x and y, so object y will overwrite the value set by object x.

Array of Objects

- The array of variables that are of type class are called arrays of objects.
- Consider the following class definition.

```
Class Person
{
        int roll;
        char name[50],
        public:
        void getdata();
        void showdata();
};
```

- The identifier Person is a user-defined type and can be used to create variables of Person type relating to different categories called as objects.
- Example:

  Person doctor[3]; //array of doctors consisting of three objects named doctor[0], doctor[1] and doctor[2] of type Person class.

  Person engineer[2];//array of Engineers consisting of two objects named engineer[0], engineer[1] of type Person class.

- We use the following syntax to access the member function by array of objects.

  Doctor[i].getdata(); where i=0, 1, or 2.

Example:

WAP to create a class Student with two data members id and name and two member function setdata() to input detail from user and displaydata() to display the detail of student. Create five objects of a class Student and call both the function using the concept of object arrays.

```
#include<iostream>
using namespace std;
        class Student
        {
                int id;
                char name[50];
                public:
                        void setdata()
                        {
                                cout<<"Enter id and name of student"<<endl;
                                cin>>id>>name;
                        }
                        void displaydata()
                        {
                                cout<<"ID= "<<id<<endl;
                                cout<<"Name= "<<name<<endl;
                        }
        };


        int main()
```

```
{
        Student s[5];
        for(int i=0;i<5;i++)
        {
                s[i].setdata();
        }
        for(int i=0;i<5;i++)
        {
                s[i].displaydata();
        }
}
```

This pointer

- It is simply a pointer that points to an object for which this function was called.

i. This can be used inside any method to refer to the current object.

```
class Demo
{
    int length,breadth;
    public:
            Demo(int a, int b)
            {
                    this->length=a;
                    this->breadth=b;
            }
}
```

However, it is not mandatory to use this pointer explicitly. We can simply use assignment statements like below.

```
class Demo
{
    int length,breadth;
    public:
            Demo(int a, int b)
            {
                    length=a;
                    breadth=b;
            }
}
```

ii. It can be used to distinguish between class instance variable and formal instance variable when there name are same.

```
class Demo
{
    int length,breadth;
    public:
            Demo(int length, int breadth)
            {
                    this->length=length;
                    this->breadth=breadth;
            }
};
```

iii. It can be used to return the object that invoked the function.

Create a class Demo with one data member n for storing a number, constructor for initializing the field, and a method Demo max(Demo) that return Demo object with maximum value between two Demo object. In main create two object of class Demo, set the value and find the maximum one.

```
#include<iostream>
using namespace std;
class Demo
{
    int val;
    public:
        Demo()
        {
            val=0;
        }

        Demo(int n)
        {
            val=n;
        }

        Demo max(Demo p)
        {
            if(p.val >val)
            return p;
            else
            return *this; // return the object that invoked it i.e. d1
        }

        void show()
        {
            cout<<val;
        }
};

int main()
{
    Demo d1(7), d2(8),d;
    d=d1.max(d2);
    cout<<"Max= ";
     d.show();
}
```

```
Max= 8
```


Friend Function

- Basically, a nonmember function can't have access to the private members of a class.
- A function is said to be a friend function of a class if it can access the members (including private members) of the class even if it is not the member function of this class.
- In other words, a friend function is a non-member function of any class that has access to the private members of the class after it is declared as friend in class definition.

- A friend function is declared as follows:

```
class A
{
        - - - - - - - - - - -- - - - - -
        - - - - - - - - - - -- - - - - -
        public:
        ------------------
        ------------------
        friend void func(void); //friend function declaration
};
```

  - The declaration is preceded by the keyword “friend”. Its definition is written somewhere outside the class. The definition doesn’t use keyword friend or scope resolution operator. The function that are declared with keyword friend are called friend function

Characteristics of a friend function:

- A friend function can be either global or a member of some other class.
- Since, friend function is not a part of the class, it can be declared anywhere in the public, private and protected section of the class.
- It cannot be called by using the object of the class since it is not in the scope of the class.
- It is called like a normal function without the help of any object.
- Unlike member functions, it cannot access member names directly. So, it has to use an object name and dot operator with each member name (like A.x)
- It, generally, takes objects as arguments.

Merits

- Able to access members without need of inheriting the class.
- Allows a more obvious syntax for calling a function rather than member function. For example num.square(); is less obvious than square(num);
- Can be declared either in the public or the private part of a class.
- Can be used to increase versatility of overloaded operator.
- **A friend function can act as bridge between two classes. Suppose we want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. In this situation, there is nothing like a friend function which can accomplish our task. Example below will shows how friend functions can act as a bridge between two classes**

Demerits

- **Friend function have access to private members of a class from outside the class which violates the law of data hiding.**
- Breach of data integrity
- Conceptually messy
- Lead to spaghetti-code situation if numerous friends muddy the boundary between classes.
- Maximum size of memory will be occupied by objects according to size of friend members.
- Cannot do any run time polymorphism in its members
- Friend function has a serious drawback because friend function breaches the wall of OOPs because the basic idea behind oops is that we can’t access private members outside the class but using friend function it makes this possible. So it is against oops paradigm.

WAP to demonstrate friend function.

```
#include<iostream>
using namespace std;
class Demo
{
        private:
                int length,breadth;
        public:
                void setData(int l, int b)
                {
                        length=l;
                        breadth=b;
                }
        friend void area(Demo);
};
void area(Demo d)
{
        int a;
        a=d.length * d.breadth;
        cout<<"Area of Rectangle="<<a;
}
int main()
{
        Demo d;
        d.setData(5,6);
        area(d);
}
```

```
Area of Rectangle=30
--------------------------------
```

So in the above program we see that the function area() is friend to class Demo. **The program clearly demonstrates how a nonmember friend function of class is accessing the two private members i.e. length and breadth of class.**

WAP to find the sum of two number fn and sn using the concept of friend function. fn and sn are private members of different class.

```
#include<iostream>
using namespace std;
class Second;//forward declaration
class First
{
        int fn;
        public:
                void setdata(int n)
                {
                        fn=n;
                }
                friend void sum(First,Second);
};
```

```
class Second
{
	int sn;
	public:
		void setdata(int n)
		{
			sn=n;
		}
		friend void sum(First,Second);
};

void sum(First f, Second s)
{
	int tot;
	tot=f.fn+s.sn;
	cout<<"Sum="<<tot;
}

int main()
{
	First f;
	Second s;
	f.setdata(5);
	s.setdata(6);
	sum(f,s);
}
```

Create classes called class1 and class2 with each of having one private member. Add member function to set a value (say setvalue) on each class. Add one more function max () that is friendly to both classes. max() function should compare two private member of two classes and show maximum among them. Create one-one object of each class then set a value on them. Display the maximum number among them.

```
#include<iostream>
using namespace std;
class Class2;// forward declaration
class Class1
{
	int a;
	public:
		void setdata()
		{
			cout<<"Enter a number"<<endl;
			cin>>a;
		}
		friend void max(Class1,Class2);
};
class Class2
{
	int a;
	public:
		void setdata()
		{
```

```
                cout<<"Enter a number"<<endl;
                cin>>a;
        }
        friend void max(Class1,Class2);
};

void max(Class1 c1, Class2 c2)
{
    if(c1.a>c2.a)
    {
        cout<<"Larger= "<<c1.a;
    }
    else
    {
        cout<<"Larger= "<<c2.a;
    }
}

int main()
{
    Class1 x;
    Class2 y;
    x.setdata();
    y.setdata();
    max(x,y);
}
```

Friend Class

- A class can also be declared to be the friend of some other class.
- A friend class can access private and protected members of other class in which it is declared as friend i.e. When we create a friend class then all the private and protected members of the friend class also become the friend of the other class.
- This requires the condition that the friend becoming class must be first declared or defined (forward declaration).

```
#include<iostream>
using namespace std;
class A
{
    friend class B; // forward declaration i.e. B is friend of Class A now
    private:
    int n;
    public:
        void setData(int x)
        {
            n=x;
        }
};

class B
{
    public:
        void display( A a)
        {
```

```
            cout<<"Value of N="<<a.n; // since Class B is friend to Class A, hence we can access
                                        private member of A using A's object from class B.
        }
};

int main()
{
    A x;
    B y;
    x.setData(55);
    y.display(x);
}
```

```
Value of N=55
```

Object State and behaviour

- In OOP, program are viewed as collection of object. Each object has two main properties: State and Behavior
- State of an object are the set of data fields with their current value. The state represents the cumulative results of an object behavior.
- Behavior of an object is how an object act and reacts in terms of state change and message passing.
- In OOP, an object state is defined by data member of the object while an object behavior is defined by the member function .
- State is determined by values of its attributes i.e. what the object have, while behavior determines action of an object i.e. what the object do.
- An object Lamp may have attribute Lamp_Status whose state can be LAMP : on and off and behavior are the Lamp turn on and turn off methods that change the lamp state.
- For object Bank_Account
  State:id, name, balance etc..
  Behaviour:deposit, withdraw etc.

Assignment 3

1. How does a c++ structure differ from a c++ class?
2. "Friend function breaches the encapsulation". Justify. Also mention the uses of friend function.
3. **Create a class distance that has separate integer data member (feet and inches). Provide two member functions. First to initialize these data members and another function to add two distance objects passed as argument to this function and return new distance objects.**
   In main create three objects of class distance. Pass distance value to first of two objects and display the result from the third object.
4. WAP to add two object each having private data member feet and inch using the concept of
   i. Call by value
   ii. Call by reference
   iii. Call by pointer

   (Hint: 1 feet = 12 inch)

End of Chapter-2