The scope Resolution operator:

*) It is possible and usually necessary for the library programmer to define the member function outside their respective classes.

*) The scope resolution operator makes it possible.

```
class Distance
{
  int ifeet;
  float finches;
public:
  void setfeet(int);
  int getfeet();
  void setInches(float);
  float getinches();
};

void Distance :: setfeet()
{
  ifeet = x;
}

void Distance :: setInches()
{
  finches = y;
}

int Distance :: getfeet()
{
  return ifeet;
}

float Distance :: getinches()
{
  return finches;
}
```

*) we can observe that the member functions have been only prototyped within the class, they have been defined outside using scope resolution operator.

Creating Libraries using the scope resolution operator.

*) As in C language, creating a new datatype in C++ using classes is also a three-step process that is executed by the library programmer.

Step1: place the class definition in a header file

```
/* Beginning of Distance.h */
class Distance
{
  int ifeet;
  float finches;
  public:
        void setfeet(int);      // prototype only
        void setinches(int);
        int getfeet();
        float getinches();
};
```

Step 2: place the definitions of the member functions in a C++ source file. A file that contains definitions of the member functions of a class is known as implementation file of that class.

```
/* Distlip.Cpp*/
#include <Distance.h>
Void Distance :: setfeet (int x)
{
  ifeet = x;
}
Void Distance :: setinches (float y)
{
  finches = y;
}
int Distance :: getfeet ()
{
  Return ifeet;
}
float Distance :: getinches()
{
  Return finches;
}
```


Step3: provide the header file and the library, to the other programmer.

Using classes in Application programs.

The steps followed by programmers for using this new datatype are:

Step 1: include the header file provided by the library programmer.

```
#include <Distance.h>
Void main()
{

}
```

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

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


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

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

Void main()
{
  Distance d1,d2;
 d1.setfeet(2);
 d1.setinches(2.2);
 d2.Setfeet(3);
 d2.setinches(3.3);
 cout<<d1.getfeet()<<d1.getinches();
 cout<<d2.getfeet()<<d2.getinches();
}
```

Step 4: compile the source code to get object code.

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

→ The 'this' pointer:

Note: 'this' pointer is a constant pointer that holds the memory address of the current object.

For a class X, the type of this pointer is 'X* const'

*) Every object in C++ has access to its own address through an important pointer called this pointer.

*) The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

*) The facility to create and call member functions of class objects is provided by the compiler.

*) The compiler does this by using a unique pointer known as the 'this' pointer.

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

*) After the compiler has made sure that no attempt has been made to access the private member of an object by non-member function, it converts the C++ code into a ordinary 'C' language code as follows.

1) It converts the class into a structure with only data member as follows.

Before:

```
class Distance
{
    int ifeet;
    float finches;
    void setfeet(int);
    void setinches(float);
    int getfeet(); // prototype only
    float getinches();
};
```

After

```
struct Distance
{
  int ifeet;
  float finches;
};
```

2) It puts a declaration of 'this' pointer as a leading formal argument in the prototype of all member functions:

Before: void setFeet(int);
AFTER: void setFeet(Distance * const, int);

Before: void setInches(float);
AFTER: void setInches(Distance * const, int);

Before: int getfeet();
After: int getfeet(Distance * const);

Before: float getinches();
After: float getinches(Distance * const);

3) * It puts the definition of the 'this pointer' as a leading formal argument in the definitions of all member functions as follows.

* It also modifies all the statements to access object members by accessing them through the 'this' pointer using pointer-to-member access operator (→)

fore:
```
Void Distance :: setfeet (int x)
{
 ifeet = x;
}
```

```
Void Distance :: setfeet (Distance * const this, int x)
{
  this → ifeet = x;
}
```

fore: int Distance :: getfeet ()
```
    { return ifeet;
    }
```

```
: int Distance :: getfeet (Distance * const this)
  { return this → ifeet;
  }
```

4) It passes the address of invoking object as a leading parameter to each call to the member function as follows:

Before d1.setfeet(1);
After setfeet(&d1, 1);

Before d1.setInches(1.1);
After setInches(&d1, 1.1);

Before cout << d1.getfeet();
After. cout << getfeet(&d1);

[Data Abstraction
Explicit Address Manipulation]

→ The Arrow Operator

*) Member functions can be called with respect through a pointer pointing at the object.

*) The arrow operator (→) does this. An Illustrative example follows.

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

Void main()
{
  Distance d1;    //object
  Distance *ptr;  //pointer
  ptr = &d1;      //pointer initialized
  ptr -> setfeet(1); // Calling member functions through
  ptr -> setInches(1.1);      pointers.
  Cout << ptr -> getfeet() << ptr -> getInches();
}
```

o/p : 1 1.1

A) It is interesting to note that Just like the dot (.) operator, the definition of the arrow (→) operator has also been extended in C++.

*) It takes not only data members on its right as in C, but also member functions.

Calling one member function from Another

*) One member function can be called from another. An illustrative example follows.

```
Class A
{
  int x;
public: Void setx(int);
        Void setxindirect(int);
};
Void A :: setx(int p)
{
  x = p;
}
Void A :: setxindirect(int q)
{
  setx(q);
}
```

```
Void main()
{
  A A1
  A1.setxindirect(1);
}
```

*) It is relatively simple to explain the above program.

1)*) The call to the A::setxindirect() changes from

A1.setxindirect(1);

to

setxindirect(&A1, 1);

2)*) the definition changes

```
Void A :: setxindirect(int q)
{ setx(q);
}
```

to

```
Void setxindirect(A * const this, int q)
{ this -> setx(q);
}
```

↓ pointing to the current instance

which inturn changes to

```
Void setxindirect(A * Const this, int q)
{
  Setx(this, q);
}
```

→ Member functions and member data.

Various kinds of member functions and member data that classes in C++ have.

1) Overloaded member functions.

*) Member functions can be overloaded just like non-member functions example.


```
#include <iostream.h>
class A
{
  public: void show();
          void show(int);
};

void A :: show()
{
  cout << "Hello";
}

void A :: show(int x)
{
  int i;
  for(i=0; i<x; i++)
    cout << "Hello";
}

void main()
{
  A A1;
  A1.show();
  A1.show(3);
}
```

o/p

Hello
Hello
Hello
Hello

Listing: overloaded member function.

*) Function overloading enables us to have two functions of the same name and same signature in two different classes. The following class definitions illustrate the point.

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

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

Listing: facility of overloading functions permits member functions of two different classes to have the same name.

*) A function of the same name 'show()' is defined in both the classes - 'A and B'. The signature also appears to be the same.

*) But our knowledge of the 'this' pointer, we know that the signature are actually different.

The function prototypes in the respective classes are actually as follows:

```
void show(A * const);
void show(B * const);
```

Default values for formal Argument of member function

*) Default value can be assigned (specified) for formal arguments of member functions also.

```
#include <iostream.h>
class A
{
public: void show(int =1);
};
void A::show(int p)
{
  for(int i=0; i<p; i++)
    cout<<"Hello";
}
```

```
void main()
{
  A A1;
  A1.show(); // default value
  A1.show(3); // default value overridden
}
```

o/p
Hello
Hello
Hello
Hello

→ Inline member functions.

*) C++ inline function is ~~powerful~~ powerful concept that is commonly used with classes. If the function is inline.

*) The compiler places a copy of the code of that

of that function at each point where the function is called at compile time.

*) Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

```
Class A
{
  public: void show();
};
inline void A::show()
{
  definition
}
```


→ constant member function.

*) A const or a constant member function can only read or retrieve the data members of the calling object without modifying them.

*) If such a member function attempts to modify any

any data member of the calling object, a compile-time error is generated.

*) The syntax for defining a const member function is

```
Return_type function_name (parameter_list) const
{
  // body of member function
}
```

```
int x, y, sum;
Void Add (int x, int y) const;
{ sum = x + y;
  return sum;
}
```

/* Beginning of distance.h */

```
class Distance
{
  int feet;
  float inches;
public:
  void setfeet(int);
  void setinches(float);
  int getfeet() const;
  float getinches() const;
};
```

```
#include "Distance.h"
void Distance :: setfeet(int x)
{ feet = x;
}
Void Distance :: setinches(int y)
{ inches = y;
}
int getfeet
Void Distance :: getfeet () const
{
  feet++; // ERROR;
  return feet;
}
```

```
float Distance :: getInches () const
{
  Inches = 0.0;   // Error
  return inches;
}
```

→ Mutable Data members.

*) A mutable data member is never constant. It can be modified inside constant function also.

*) prefixing the declaration of a data member with the keyword mutable makes it mutable.


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

```
void def()
 {
   x++;   // OK: can modify
   y++;   // OK: can modify a mutable data.
 }
};
```

→ Friends

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

*) A friend function is prototyped within the definition of the class of which it is intended to be a friend.

*) The prototype is prefixed with the keyword friend. Since it is non-member function, it is defined without using the scope resolution operator.

*) Moreover it is not called with respect to an object.

```
/* Beginning of friend.cpp */

Class A
{
   int x;
   public: friend void abc(A &); // prototype of
                                    friend function
};

Void abc(A& obj) // definition of friend function
{
   obj.x++;        // accessing private members
}

Void main()
{
   A A1;
   abc(A1);
}
```


→ A few points about the friend function.

*) friend keyword should appear in the prototype only

*) Since it is non-member function of the class of which it is friend, prototype can appear either in private or public section of the class

*) A friend function takes one extra parameter as compared to member function. This is because it cannot be called with respect to object. Instead, the object itself appears as an explicit parameter in the function calls.

*) We need not and should not use the scope resolution operator while defining the friend function.

→ Friend classes:

*) A class can be a friend of another class.

*) Member functions of a friend class can access private data members of objects of the class of which it is a friend.

*) If class B is to be made a friend of class A, then the statement i.e friend class B; should be written within the definition of class A.

for example: class A
{
friend class B; // declaring B as a friend of A.
/* rest of class A
};

*) It does not matter whether the statement declaring class B as a friend is mentioned within the private or the public section of class A.

*) member function of class B can access the private data members of objects of class A.

```
Class B; // forward declaration

Class A
{
  int x;
  public: void setx(int=0);
        // int getx();
          friend class B;
};

Class B
{
  A *ptr;
  public: void map(A * const);
          void test_friend(int);
};

Void B::map(A *const p)
{
  ptr = p;
}

Void B::test_friend(int i)
{
  ptr->x = i; // Accessing private data member
}
```

*) we can see, member functions of class B are able to access private data member of objects of class A.

→ Friend Member function

*) How can we make some specific member functions of one class friendly to another class.

⇒ friend member function

```
Class B;
Class A
{
 private: int numA;
 public : A()
          {
            numA=12;
          }
    friend int add(A, B);
};

int add(A objectA, B objectB)
{
  return (objectA.numA + objectB.numB);
}

int main()
{
  A objectA;
  B objectB;
  cout << "sum:" << add(objectA, objectB);
}
```

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

class B
{
  private: int numB;
  public : B()
           {
             numB = 3;
           }
  friend int add(A, B);
};
```

o/p sum = 15

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

Class withfriend
{
 private : i;
 public : Friend Void fun();
};
```

```
Void fun()
{
 withfriend o1;
  o1.i =10;
 cout<<o1.i;
}
int main()
{
 fun();
}
```

o/p 10

⇒ Class TWO;

```
Class ONE
{
 public: Void fun(TWO); // Member func declaration
};
```

```
int main()
{
 ONE one;
 TWO o;
  one.fun(o);
}
```

o/p 10//

```
class TWO
{
 private: int value;
 public:
    friend Void ONE::Fun(TWO); // friend func declaration
};

void ONE::fun(TWO o)
  {
    o.value =10;
    cout<<o.value;
  }
```

The modified definition of the class A is

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

*) In order to compile this code succesfully, the compiler should first see the definition of the class B. otherwise, it does not know the test_friend() is a member function of the class B.

) A pointer of type A is a private data member of class B. So, the compiler should also know that there is a class A before it compiles the definition of class B.

```
class A;
class B
{
  A *ptr;
  public: void map();
  void test_friend(int=0);
};
```

```
class A
{ int x;
  public:
    friend void B::
      test_friend(int=0);
};
```

→ Friends as Bridges

*) Friend functions can be used as bridges between two classes

*) suppose there are two unrelated classes whose private data members need a simultaneous update through a common function.

*) This function should be declared as a friend to both the classes.


```
class B;

class A
{
  friend void ab(A&, B&);
};

class B
{
  friend void ab(A&, B&);
};
```

→ Static Members

*) Static Member Data

*) static data member hold global data that is common to all objects of the class. (common data accessed by all objects)

*) when we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

*) All static data is initialized to zero when the first object is created,

~~For Example~~: *) we prefix the declaration of a variable within the class definition with the keyword static to make it a static data member of the class.

```
class Account
{
    static float interest_rate;
};
```

*) A statement declaring a static data member inside a class will obviously not cause any memory to get allocated for it.

*) Memory for a static data member will not get allocated when objects of the class are declared. This is because a static data member is not a member of any object.

*) We must not forget to write the statement to define (allocate memory for) a static member variable. Explicitly defining a static data member outside the class is necessary.

```
float Account :: interest_rate;
```

*) The above statement initializes 'interest_rate' to zero. If some other initial value is desired, rewrite the statement as `float Account :: interest_rate = 4.5;`

```
Class A
{
  int x;
  char y;
  float z;
  static float s;
};
float A :: s = 1.1;
void main()
{
  cout << sizeof(A);
}
```

o/p 7

Object & function, namespace
Object & Array Nested class
Object inside class

→ Objects & functions:

*) objects can appear as local variable inside function. They can also be passed by value or by reference to a function.

*) Finally, they can be returned by value or by reference from function.

Example:

```
Class Distance
{
  int feet;
  float inches;

public: void setfeet(int);
        void setinches(float);
        int getfeet();
        float getinches();
};      Distance add(Distance);
```

```
Distance Distance::add(Distance dd)
{
  Distance temp;
  temp.feet = feet + dd.feet;
  temp.inches = inches + dd.inches;
  return temp;
}

void main()
{
  Distance d1, d2, d3;
  d1.setfeet(1);
  d1.setinches(1.1);
  d2.setfeet(2);
  d2.setinches(2.2);
  d3 = d1.add(d2);
  cout << d3.getfeet() << d3.getinches();
}
```

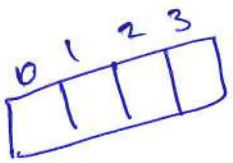
→ Objects and Arrays

We can create arrays of objects. The following program shows how.

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

Void main()
{
  int a, b;
  Distance a[SIZE];
  int
  for(i=0; i<SIZE; i++)
  {
  Cout<<"Enter the feet";
  Cin>>a;
    a[i].Setfeet(a);
Cout<<"Enter the inches";
  Cin>>b;
    a[i].Setinches(b);
}
  for(int i=0; i<SIZE; i++)
  {
  Cout<<a[i].getfeet()<<
        a[i].getinches();
  }
}
```

→ Array inside objects

```
#define SIZE 4
Class A
{
  int a[SIZE];
public: void setelement(int p, int elem);
        int getelement(int p);
};

Void A::setelement(int p, int elem)
{
  if (p>=SIZE)
    return;
  a[p] = elem;
}

int A::getelement(int p)
{
  if (p>=SIZE)
    return -1;
  return (a[p]);
}
```




```
Void main()
{
  int p, elem;
  A a1, a2;
cout << "Element & position";
cin >> ele >> p;
a1.setelement(p, elem);
cout << "Enter the pos";
cin >> p;
int res = a1.getelement(p);
if (res == -1)
{ cout << "out of bound";
else cout << res;
}
```

→ Namespaces

*) Namespaces enable the C++ programmer to prevent pollution of the global namespace that leads to name clashes.

*) global namespace refers to the entire source code. It also includes all the directly & indirectly included header files.

*) By default, the name of each class is visible in the entire source code, that is, global namespace. This can lead to problem.

*) Suppose a class with the same name is defined in two header files:

```
/* A1.h */
class A
{
};
/* A2.h */
class A
{
};
```

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

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

*) How can this problem be overcome? How can we ensure that an application is able to use both definitions of class 'A' simultaneously?

*) Enclosing the two definitions of the class in separate namespaces overcomes this problem.

```
/*A1.h*/
namespace A1
{
  class A
  {
  };
}
```

```
/*A2.h*/
namespace A2
{
  class A
  {
  };
}
```



```
#include "A1.h"
#include "A2.h"
Void main()
{
  A1::A obj1;
  A2::A obj2;
}
```

→ Nested Class :

*) A class can be defined inside another class. Such a class is known as a nested class.

*) The class that contains the nested class is known as the enclosing class.

*) Nested class can be defined inside prive, prokected or public portions of enclosing class

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


*) The size of objects of an enclosing class is not affected by the prelence of nested classes.

```
#include <iostream.h>
 Class A
 {
   int x;
   public: class B
           { int y;
           };
 };
```

```
Void main()
{
 cout << sizeof (int);
 cout << size of (A);
}
o/p 4
    4
```

*) How are the member functions of a nested class defined?

Member functions of a nested class can be defined outside the definition of the enclosing class.

*) This is done by prefixing the function name with the name of the enclosing class followed by scope resolution operator. This, is in turn, is followed by name of the nested class followed by scope resolution operator.

```
/* nestclass.h */
class A
{
  public: class B
          {
            public: void BTest();
          };
};
```

```
#include "nestclass.h"
Void A::B::BTest()
{
 /* definition */
}
```

*) A nested class may be only prototyped within its enclosed class & defined later.

```
Class A
{
  Class B; // prototype only
};

Class A :: B
{
  /* definition of Class B */
}
```

*) Objects of the nested class are defined as A::B B1;

→ Namespace

```
namespace My
{
  class Vector { };
  class car { };
}
namespace your
{
  Class Vector { };
  class Bus { };
}
using my::Vector; // using your::Vector;
```

①
```
void main()
{
  Vector v;
}
```

②
```
using my::car;
void main()
{
  car c;
}
```

③
```
using namespace my;
void main()
{
  car c;
  Vector v;
}
```

```
void main()
{
  my::car c;
```

Nested Class

```
Class person
{
  public: string name;
     Class Address
     {
       public:
            string country;
            string stname;
            int phno;
     };
  Address addr;
  void Addrsplace()
  {
    cout<<name <<endl <<addr.country <<endl <<
          addr.stname <<endl <<addr.phno<<endl ;
  }
};

int main()
{
  person anil;   // Person :: Addrss ad;
  anil.name = "anil";
  anil.addr.country = "india";
  anil.addr.stname = "MGRoad";
  anil.addr.pho = 69;
  anil.addrsplace();
}
```

→ Namespace

```
namespace My
{
  class Vector {};
  class car{};
};
namespace your
{
  Class Vector{};
  class Bus{};
};
using my::Vector; //using your::Vector;
void main()
{
  Vector v;
};
```

② using my::car;

```
void main()
{
  car c;
};
```

③ using namespace my;

```
void main()
{
  car c;
  Vector v;
};
```

④

```
void main()
{
  my::car c;
};
```

⟹ Constructors

*) The Contructor gets called automatically for each object that has just got created.

*) It appears as member function of each class.

*) It has the same name as that of the class.

*) It may or may not take parameters.

*) It does not return anything (not even void)

*) The compiler embeds a call to the constructor for each object when it is created. Suppose a class A has been declared as follows:

```
/* Beginning of A.h */
class A
{
  int x;
public: void setx(const int=0);
        int getx();
};
```

*) Consider the statement that declares an object of a class A in the following listing.

```
#include "A.h"
Void main()
{
  A A1;   //object declared ... constructor called
}
```

) The statement in the function 'main()' is transformed into the following statements.

```
A A1; //memory allocated for the object (four bytes)
A1.AC(); //Constructor called implicitly by compiler
```


=) The Zero-argument constructor:

*) we can and should define our own constructor if
e need arises. If we do so,
) the Compiler does not define the constructor. However
it still embeds implicit calls to the constructor as before.
) Constructor is a non-static member function, It is
called for an object. It, therefore takes the 'this'
pointer as a leading formal argument.

*) The addrs of the invoking object is passed as a leading parameter to the constructor call.

```
/* A.h */
*) class A
{
  int x;
  public: A(); // constructor.
          void setx(int);
          int getx();
};
```

```
#include "A.h"
#include <iostream.h>
A::A()
{
  cout<<"Constructor of class A called";
}
```

```
#include "A.h"
void main()
{
  AA1;
  cout<<"End of prg";
}
```

o/p Constructor of class A called.
End of prg

⇒ parameterized constructor :-

*) constructors take arguments and can, therefore be overloaded.

*) suppose, for the class 'Distance', the library programmer decides that while creating an object, the application programmer should be able to pass some initial values for data members containing the object. he can create a parameterized constructor.

```
// Distance.h
#include
class Distance
{
  public: Distance();
          Distance(int, float);
        // parameterized constructor
};
```

```
// Distance.cpp
#include "Distance.h"
Distance :: Distance()
{
  feet = 0;
  inches = 0.0;
}
Distance :: Distance(int p, float q)
{
  feet = p;
  inches = q;
}
```

```
#include <iostream.h>
#include "Distance.h"
Void main()
{
  Distance d1(1, 1.1); //parameterized constructor
  cout << d1.getfee() << d1.getInches();
}
```

o/p 1 1.1

⇒ Copy Constructor

*) The Copy Constructor is a special type of parameterized constructor.

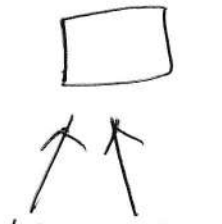
*) As its name implies, it copies one object to another. It is called when an object is created and equated to an existing object at the same time.

*) The copy constructor member-wise copies the object passed as a parameter to it into the object for which it is called.

```
A A1;
A A2 = A1;
```

*) A A1; // copy constructor will be called
A A2 = A1;

i.e A A2(A1)



*) void abc(A A2) // copy constructor
{
}
A A1;
abc(A1);

*) A abc() // copy constructor
{
A A1;
return A1;
}

⇒ Destructor

*) The destructor gets called for each object that is about to go out of scope.

*) It appears as a member function of each class whether we define it or not.

*) It has same name as that of class but prefixed with a tilde sign. It does not takes parameters.

*) It does not return anything (not even void)

*) The prototype of a destructor is

```
~<classname>();
```

*) The need for a function that guarantees deinitialization of member data of a class and free up the resources acquired by the object.

```
/* Beginning of A.h */

Class A
{
  int x;
  public: A();
          void setx(int);
          int getx();
          ~A();
};
```

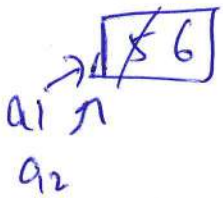
```
/* A.cpp */
#include "A.h"
#include <iostream.h>

A::A()
{
  Cout << "constructor is called";
}
A::~A()
{
 Cout << "Destructor is called";
}
```

```
#include "A.h"
#include <iostream.h>
Void main()
{
  A A1;
 cout << "End of prg";
}
```

o/p Constructor is called.
End of prg.
Destructor is called.

For example;

```
class A
{
  int a;
};
Void main()
{
  A a1, a2;
  a1.a = 5
  a2.a = 6;
}
```




```
class A
{
  Static int a;
};
int A::a;
Void main()
{
  A a1, a2;
  a1.a = 5;
  a1.a = 6;
}
```


→ Static member function:

*) How do we create a member function that need not be called with respect to an existing object?.

*) This function's sole purpose is to access and modify Static data members of the class. Static member functions fulfill the above criteria.

*) Prefixing the function prototype with the keyword static specifies static member function. However, the keyword static should not reappear in the definition of the function.

```
Class Account
{
  static float interest_rate;
  public: static void set_interest_rate(float/int);
};
float Account::interest_rate = 4.5;

void Account::set_interest_rate(float p)
{
  interest_rate = p;
}
```

Now the 'Account::set_interest_rate()' function can be called directly `Account::set_interest_rate(5);`

Note: Static member function do not take the 'this' pointer as formal argument. Therefore accessing non-static data members through static member function results in compile time error

⇒ The Need for structures.

*) There are cases where the value of one variable depends upon that of another variable

*) Take the example of date. A date can be programmatically represented in C by three different integer variables taken together.

```
int d, m, y;
```

Here d, m and y are three variable are not grouped together in the code they actually belong to the same group.

*) The value of one variable may influence the value of other two.

*) Suppose d=1; m=1; y=2002; // 1st January, 2002
Now, if we write next_day (&d, &m, &y)
'd' will become 2, 'm' will remain 1, and 'y' will remain 2002.
But if
d=28, m=2, y=1999; and we call the function as
next_day (&d, &m, &y);
'd' will become 1, 'm' will become 3, and 'y' will remain 1999.

*) A change in the value of one may change the value of the other two. But there is no language construct that actually places them in the same group.

Thus, members of the wrong group may accidentally sent to the function.

```
d1 = 28, m1=2; y1=1999;
d2 =19, m2=3, y2=1999;
next_day (&d1, &m1, &y1);
next_day (&d1, &m2, &y2);
```

There is nothing in the language itself that prevents the wrong set of variables from being sent to the function.

*) Let us try arrays to solve the problem. Suppose the next_day() function accepts an array as a parameter. Its prototype will be: void next_day(int *);

*) Let us declare date as an array of three integers

```
int date[3];
date[0] = 28;
date[1] = 2;
date[2] = 1999; // 28th feb 1999
```

Let us call the function as follows

```
next_day(date);
```

*) The values of date[0], date[1], and date[2] will be correctly set to 1, 3 and 1999 respectively.

*) Although this method seems to work, it certainly appears unconvincing After all any integer array can be passed to the function. even if it does not represent a date.

*) There is no datatype of date itself. moreover, this solution of array will not work if the variables are not of the same type.

*) Solution: Create a datatype called date itself using structures

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

*) Now the 'next_day()' function will accept the address of a variable of the structure date as parameter.

```
void next_day (struct date *);
```

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

*) Now, d1.d, d1.m, d1.y will correctly set the value, since function takes the address of entire structure variable

*) there is no chance of variable of diffuent groups being sent to the function.

*) Structure is a programming construct in c that allows us to put togethar variables that should be togethar.

*) The new data types (declaring variables of this data type)

```
struct date d1;
```

They call the associated functions by passing these variables or address to them.

```
d1.d=31;
d1.m=12;
d1.y=2003;
next_day(&d1);
printf("The next day is %d/%d/%d", d1.d, d1.m, d1.y);
```

⇒ Creating a New Data type using structures:

Creation of a new data type using structures is a three-step procedure that is executed by the library programmer.

Step1 :- put the structure definition and the prototypes of the associated functions in a header file.

```
/* Beginning of date.h */
/* This file contains structure definition & prototypes */

struct date
{
  int d,m,y;
};

void next_day (struct date *);    // get next date
void get_sys_date (struct date *); // get current system date

/* End of date.h */
```

Listing: Header file containing definition of a structure variable and prototype of its associated functions.

Step2: put the definition of the associated functions in a source code and create a library.

```
/* Beginning of date.c */
/* This file contains the definitions of the associated functions*/
#include "date.h"

void next_day (struct date * p)
{
 // Calculate the date, represented by *p & set it to *p.
}

void get_sys_date (struct date *p)
{
 // determine the current system date and set it to *p
}

 /* End of date.c */
```

Listing: Defining the associated function of a structure.

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

⇒ Using structures in Application programs

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

Step1: Include the header file provided by the library programmer in the source code.

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

Step2: Declare variables of the new date type in the source code.

```
#include "date.h"
void main()
{
  struct date d;
}
```


Step3: Embed calls to the associated functions by passing these variable in the source code.

```
#include "date.h"
void main()
{
  struct date d;
  d.d = 28;
  d.m = 2;
  d.y = 1999;
  next_day(&d);
}
```

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

Step 5: Link the object file with the library to get the executable file.

⟹ procedure-oriented programming system

*) The procedure-oriented programming pattern characterizes a program as a set of linear steps to be carried out, such reading, processing and printing.

*) To achieve these goals, procedures (functions) are used. Since the primary focus is on procedures this approach is called procedure oriented programming.

*) The program are written with one or more procedures (functions). And each procedure does a specific task.

*) Data is passed openly (freely) from one procedure to another, There will be a global data that can be accessed by all procedure.

*) In pop, the main task is taken up first and is divided into smaller logically independent components which in turn accomplish the specified sub tasks.

*) Thus, it is a top-down in designing a prg. figure 1.1 shows a structure of a prg in pop.

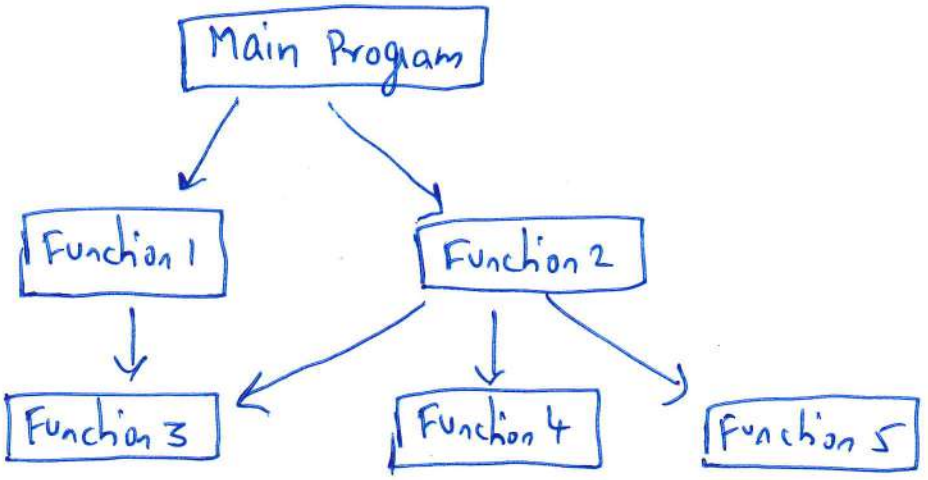


Fig: Block structure of a program in Pop

Drawbacks:

*) The complexity increases as the programs grow larger & complex.

*) It does not model real world objects as close to a user's perspective as possible.

⇒ object-oriented programming (OOP)

*) In order to overcome the drawbacks of the procedure-oriented programming, the second programming paradigm called object oriented programming was conceived. It is popularly called OOPs.

*) In OOPs, the primary focus is on data rather than procedure. The data cannot be moved freely around the system however it is more closely associated with the functions.

*) The functions cannot exist without data, and data need functions to give it shapes.

*) In this paradigm, the programs are divided into a number of entities called objects. data the data & functions are designed to characterize these objects.

*) Data is hidden and cannot be shared by outside functions. The objects that are created may interact with each other via functions.

*) In OOPs, the objects are created first and the data and functions around them are developed later, so OOPs follows bottom-up design.

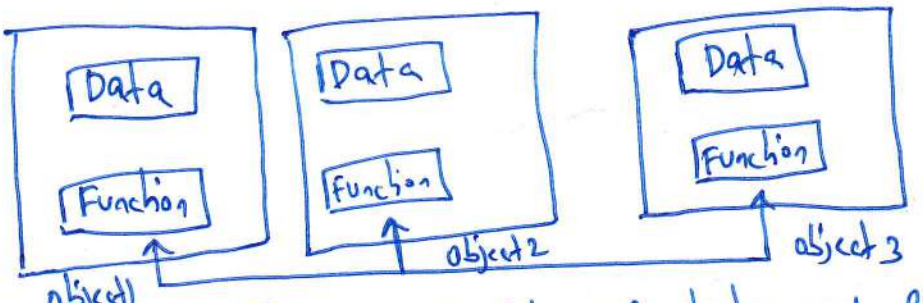


Fig: organization of data and function

Procedure-oriented Programming	Object-oriented Programming
*) Focus is on the procedure rather than the data.	*) Focus is on the data rather than the procedure
*) Data is not secure as it freely moves from one procedure to another	*) Data is secured.
*) Employs top-down programming design	*) Employs bottom-up programming design
*) It does not model the real world's objects	*) It does model the real world object.
*) programs are decomposed into functions.	*) programs are decomposed into objects.

OOPS CONCEPTS

⇒ OBJECTS & CLASSES

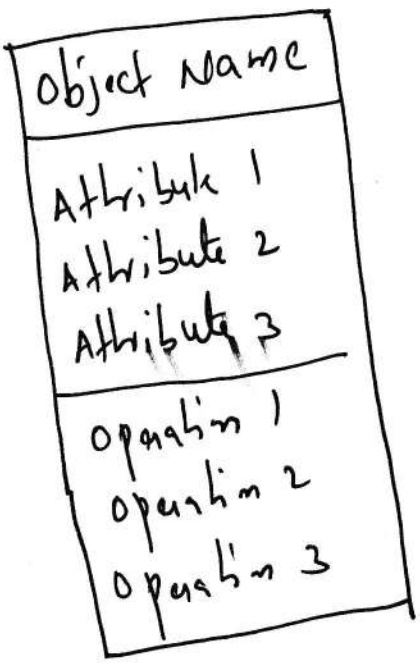
*) object is an entity; which has got some name, structure (attributes) and behaviours (operations).

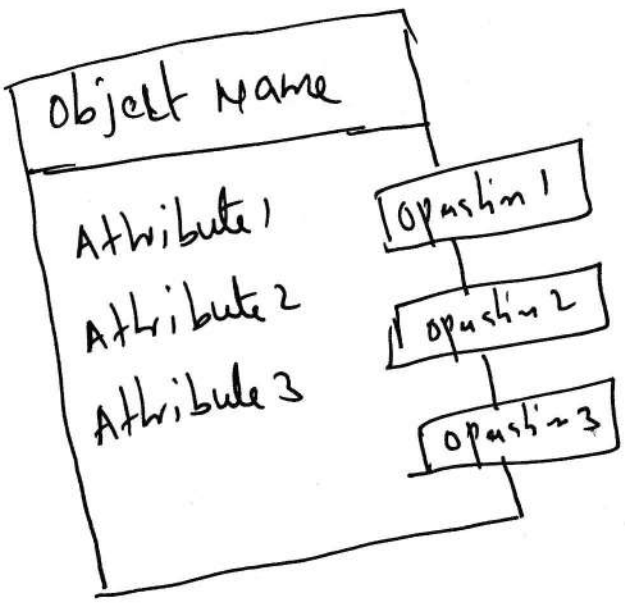
These entities are represented as an object in the program.

*) An object can be a person, place, thing with which a computer must deal.

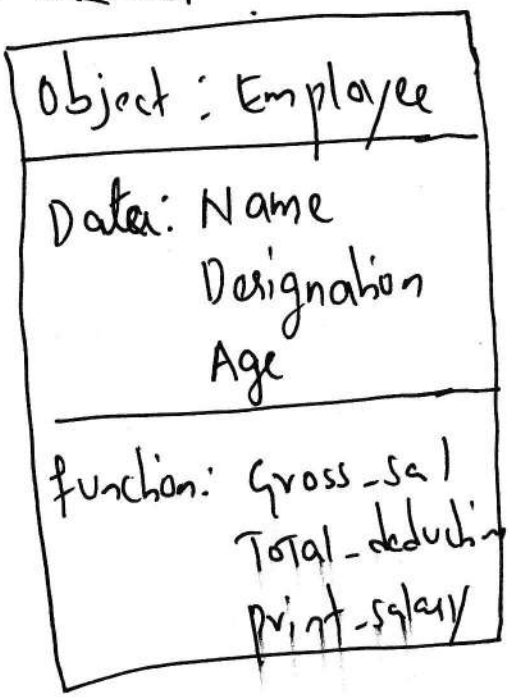
*) object may correspond to real world (such as students, employees, bank accounts, inventory etc) computer hardware (such as printer, scanner, monitor etc)

*) object can be represented as





→ ~~Classes~~



→ classes :-

*) A class is a collection of objects having common features. we can say that the class is a description of identical objects.

*) A class is a user defined datatype which encloses datamembers as well as the functions that manipulates these data.

*) once a class is created it is possible for us to create any number of objects for that class.

*) Each object is then said to be an instance of its class.

→ Abstraction:

*) It is an essential element of OOP.

*) It can be defined as the separation of unnecessary details, so to reduce the complexities of understanding requirements. That is, the essential features are represented without including the background details.

*) Data abstraction is actually another manifestation of Data Encapsulation.

→ Encapsulation:

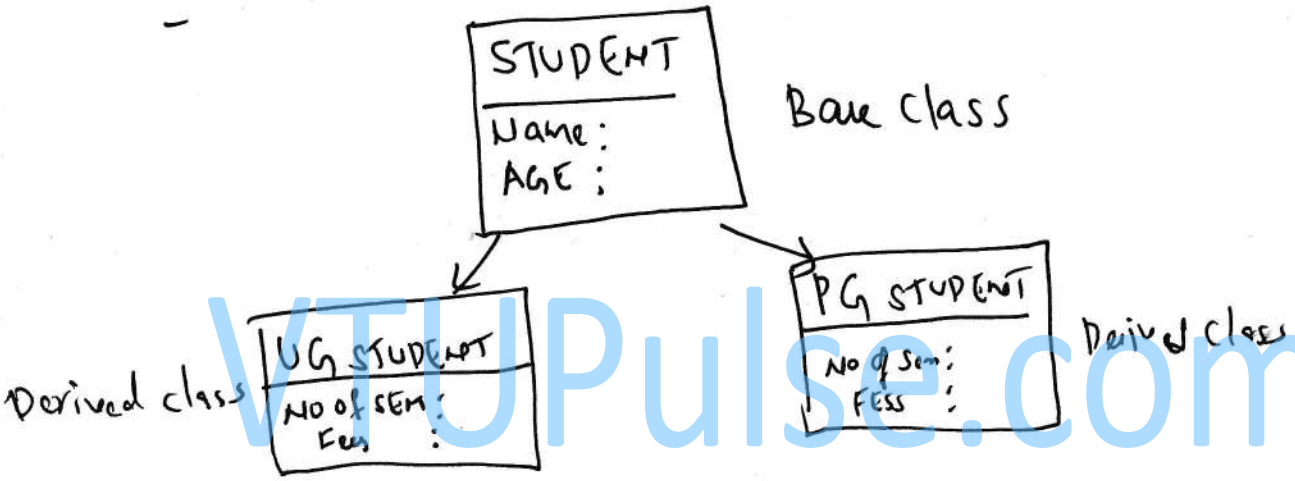
*) It is the mechanism that puts in capsule the data & function together.

*) It is the result of hiding the implementation details of an object from its user.

*) It prevents unauthorized access to some piece of information (data) or functionality. The object hides its data from other objects & allows the data to be accessed by only those functions which are packaged in the class of that object.

*) INHERITANCE:-

- is the most powerful feature of OOPs.
- In C++, it is implemented by creating the new classes from the existing class, Here, we can take the form of the existing class and then add code to it, without modifying the existing class.

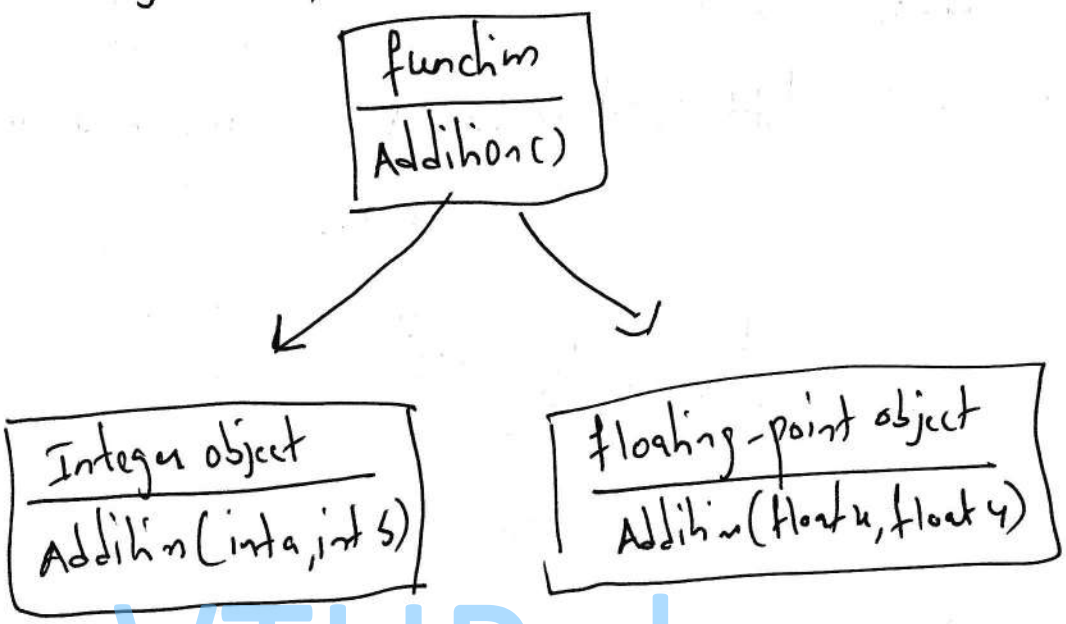


Inheritance of class student.

```
class student    // base class
{
    private:  char name[10];
              int  AGE;
    public:   void getdata();
              void putdata();
};

class UGSTUDENT : public student    // derived class
{
    private:  int NO_of_sem;
              int fee;
```

⇒ polymorphism:

*) The meaning of polymorphism is "having many forms". A single entity can take more then one form.



⇒ ~~Difference~~ Comparison of C++ with C

*) C++ is an extension of C language. &

*) C++ compiler can compile programs written in C language, however the reverse is not possible.

*) Decision-making constructs, looping constructs, structures, functions etc are written in exactly the same way in C++ as they are in C language.

*) C++ provides a number of additional keywords and language constructs that enable it to implement the object oriented programming paradigm.

Example:
```
class Date
{
    private: int d,m,y;
    public : Date();
             void get_sys_date();
             void next_date();
};
```

* Listing: Redefining Date structure in C++.

*) The following differences can be noticed b/w Datastructure in C and C++.

- The keyword class has been used instead of struct.
- Two new keywords - private & public - appear in the code.
- Apart from data members, the class constructor also has member functions.
- A function that has the same name as the class itself is also present in the class. This is the class constructor will be discussed later.
- Console input/output in C++
- Some non-object-oriented features provided exclusively in C++ (reference variables, function overloading, inline function)

Differences between C and C++:

C	C++
*) It is a procedure-oriented language	*) It is object oriented language (Emphasis on data)
*) It does not contain scope resolution operator	*) It contains scope resolution operator (::)
*) All variables must be declared before the first executable statement	*) The variables are declared at any point, where they are required.
*) The keywords new & delete are not available (memory management)	*) Keywords new & delete are available.
*) It does not provide default arguments.	*) It provides default arguments.

⇒ Console Input/output in C++

Console output:

*) The output function in C language, such as 'printf()', however, there are some more ways of

of outputting to the console in C++.

```
#include <iostream.h>
void main( )
{
  int x;
  x=10;
  cout << x;
}
```

o/p 10

*) 'cout' is actually an object of the class 'ostream_withassign' (we can think of it as a variable of the structure 'ostream_withassign').

*) It stands as an alias for the console output device, that is, the monitor

*) The << symbol, originally the left shift operator, has its definition extended in C++, it operates as the 'insertion' operator.

*) It is a binary operator, It takes two operands. The operand on its left must be some object of the 'ostream' class. The operand on its right must be a value of some fundamental datatype.

*) The value on the right side of the 'insertion operator' is 'inserted' into the stream headed towards the device

associated with the object on the left.

* The file 'iostream.h' needs to be included in the source code to ensure succesful compilation because the object 'cout' and the 'insertion' operator have been declared in that file.

* Another object endl allows us to insert a newline into the output stream.

```
#include<iostream.h>
Void main()
{
  int x,y;
  x=10;
  y=20;
 Cout<<x;
 Cout<<endl; //Insert newline.
 Cout<<y;
}
```

o/p 10
20

Listing: Inserting a newline by endl

```
#include<iostream.h>
void main()
{
  int x;
  float y;
  x=10;
  y=2.2;
  cout<<x<<endl<<y;
}
```

o/p
10
2.2

Listing: Cascading the 'insertion' operator

*) we can pass constants instead of variables as operands to the 'insertion' operator.

```
#include<iostream.h>
Void main()
{
  Cout << 10<< endl<<"Hello\n"<<3.4;
}
```

o/p 10
Hello
3.4

Listing: outputting constants using the 'insertion' operator.

Console Input:

*) The input function in C language such as scanf(), can be included in C++ program, However, we do have some more ways of inputting from the console C++.


```
#include<iostream.h>
void main()
{
  int x;
  Cout<<"Enter a number";
  Cin>>x; // Console i/p
  cout <<" you entered "<<x;
}
```

o/p
Enter a number 10

*) cin is actually an object of the class 'istream_withassign' It stands as an alias for console Input device, that is the keyboard.

*) The >> symbol, right shift operator, has had its definition extended in C++.

*) It operates as the extraction operator. It is a binary operator and takes two operands.

*) The operand on its left must be some object of 'istream_withassign' class. The operand on its right must be a variable of some fundamental data type.

*) The value for the variable on the right side of the 'extraction' operator is extracted from the stream originating from the device associated with the object on left


```
#include<iostream.h>
void main()
{
  int x,y;
  cout<<"Enter two numbers";
  cin>>x>>y;
  cout<<"you entered "<<x<<" and "<<y;
}
```

o/p Enter two numbers
10 ↵
20 ↵
you entered 10 and 20

⇒ Variables in C++

*) Variables in C++ can be declared anywhere inside a function and not necessarily at its very beginning for example:

```
#include<iostream.h>
void main()
{
  int x;
  x=10;
  cout<<"value of x:"<<x<<endl;
  int *ptr; // declaring a variable
  iptr = &x;
  cout<<"Addres of x="<<iptr<<endl;
}
```

o/p value of x = 10
Addres of x = 0x21874

⇒ Reference Variables in C++

*) How are assignment operations such as 'x=y' executed during run time?

*) The OS maintains the addresses of each variable as it allocates memory for them during runtime.

*) In order to access the value of a variable, the OS first finds the address of the variable and then transfers

control to the byte whose addres matches that of the variable.

*) Suppose the following statement is executed x=y; (integer)

The steps followed are:

1) The OS first finds the address of 'y'.
2) The OS transfers control to the byte whose address matches this address.
3) The OS reads the value from the blocks of four bytes that starts with this byte.
4) The OS pushes the read value into a temporary stack.
5) The OS finds the address of x
6) The OS transfers control to the byte whose address matches this address.
7) The OS copies the value from the stack where it had put it earlier, into the blocks of four bytes that starts with the byte whose address it has found above (address of x)

*) Syntax for declaring a reference variable is as follows:

<data_type> &<ref_var_name> = <existing_var_name>;

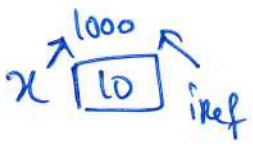
int & iRef = x;

*) iRef is a reference to x, This means that although 'iRef' and 'x' have separate entries in O.S, their addresses are actually the same.


```
#include<iostream.h>
Void main()
{
  int x;
  x=10;                          O/p  10
  Cout<<x<<endl;                      20
  int &Ref=x; // Ref is a refernce to x  21
  Ref=20;       // same as x=10;
  cout<<x<<endl;
  x++;
  Cout << Ref<<endl;
}
```

Listing: Reference variable.

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



y=10;

o/p
```
10
10
10
11
```

Listing: Reading the value of a reference variable

```
#include<iostream.h>
int increment(int &);
void main()
{
  int x;
  x=10;
  increment(x);
  cout<<x<<endl;
}
int increment(int &r)
{
  r++;
  return r;
}
```

o/p
```
11
```

Listing: passing by reference.

Listing: Returning by reference.

```
#include<iostream.h>
int &larger(const int &, const int &);
void main()
{
  int x,y;
  x=10, y=20;
  int &r=larger(x,y);
  r=-1;
  cout<<x<<endl<<y<<endl;
}
int &larger(const int &a, const int &b)
{
  if(a>b)
    return a;
  else
    return b;
}
```

o/p
```
10
-1
```

Listing : Returning the reference of a local variable.

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

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

*) The problem with the above program is that when the 'abc()' function terminates, 'x' will go out of scope.

Consequently, the statement

abc()=-1;

in the main() function will write '-1' in an unallocated block of memory, this can lead to run-time error.

⇒ Function prototyping:

*) function prototyping is necessary in C++. A prototype describes the function's interface to the compiler.

*) It tells the compiler the return type of the function as well as the number, type and sequence of its formal arguments.

The general Syntax of function prototype is as follows:

return_type function_name (argument_list);

for example:

int add (int, int);

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

Listing: Function prototyping

```
#include<iostream.h>
int add(int, int);
Void main()
{
  int x,y,z;
  Cout<<"Enter the value of x&y";
  Cin>>x>>y;
  z=add(x,y);
  Cout<<z<<endl;
}
```

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

O/p

```
Enter the value of x & y
  10 20
  30
```

Why is prototyping important? By making prototyping necessary, the compiler ensures the following.

- The return value of a function is handled correctly.
- Correct number & type of arguments are passed to a function.

⇒ Function Overloading:

*) C++ allows two or more functions to have the same name. for this, however, they must have different signatures.

*) Signature of a function means the number, type and sequence of formal arguments of the function.


```
#include<iostream.h>
int add(int, int);
int add(int, int, int);

Void main()
{
  int x,y;
  x = add(10,20);
  y = add(30,40,50);
  cout<<x<<endl<<y<<endl;
}
```

```
int add(int a, int b)
{
 return(a+5);
}

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

o/p

30

120

*) The Compiler decides which function is to be called based upon the number, type and sequence of parameters that are passed to the function call.

*) when Compiler encounters the first function call, x = add(10,20); it decides that the function that takes two integers as formal arguments is to be executed.

*) Accordingly, the linker then searches for the definition of the add() function

y = add(30,40,50);

is also handled by the compiler and the linker.

⇒ Default values for Formal Arguments of functions

*) It is possible to specify default values for some or all of the formal arguments of a function.

*) If no value is passed for an argument when the function is called, the default value specified for it is passed.

*) If parameters are ~~pro en~~ passed in the normal fashion for such an argument, the default value is ignored.

```
#include<iostream.h>
int add(int, int, int c=0); // third argument has default value.
void main()
{
  int x,y;
  x=add(10,20,30);  // default value ignored
  y=add(40,50);     // default value taken for the third parameter
  cout<<x<<endl<<y<<endl;
}
int add(int a, int b, int c)
{
 return (a+b+c);
}
```

o/p 60
90

*) In the above listing, a default value - zero - has been specified for the third argument of the add() function. In the absence of a value being passed to it, the compiler assigns the default value.

⇒ Inline functions:

*) Inline functions are used to increase the speed of execution of the executable file. C++ inserts calls to the normal functions and the inline functions in different ways in an ~~execution. else~~ executable.

```
Void main()
{
  double x;
  x = cube(2);
  x = cube(4);
  x = cube(10);
}

double cube(double n)
{
  Return n*n*n;
}
```


*) The C++ inline function provides a solution to this problem. An ~~iinl~~ inline function is a function whose compiled code is 'inline' with the rest of the program. That is, the Compiler replaces the function call with the Corresponding function code.

*) With inline code, the program does not have to Jump to another location to execute the code and then Jump back. Inline functions, thus, run a little faster than Regular functions.

*) If an inline function is called repeatedly, then multiple copies of the function definition appear in the code. Thus, the executable program itself becomes so large that it occupies a lot of space in the computer memory during run time. Consequently program runs slow instead of running fast.

*) For specifying an inline function, you must:

- prefix the definition of the function with the inline keyword
- define the function before all functions that call it, that is, define it in the header file itself.