Date / /

CONSTRUCTORS & DESTRUCTORS

Scanned by CamScanner

Constructor :- it is a member function of a class. it has the same name as that of the class' name & its primary job is to initialize the object to legal initial value of class.

↳ It automatically called when an object is created of that class.

```
class Student
  {
    private:
      int rollno;
      float marks;
    public:
      :
      student()
      {
        rollno.=0;
        marks=0.0;
      }
```

Constructor//

↳ constructor is a special initialization function that is called automatically wherever an instance of the class is declared. The name of the constructor is same as that of class & it return no value.

Need:-

* Constructors are used to initialize variables and to allocate memory.
* Similar to Normal functions, constructors can be overloaded.
* when an object is created and initialized at the same time, a copy constructor gets called.
* Destructor destroys the objects when they are no longer required.

Scanned by CamScanner

constructor & destructors have no return type, not even void.

Q. Why we need of initialize of constructor.
because it does not show any garbage value.

```
class student
{private:
    int rollno;
    float marks;
    public:
    void init()
    {rollno=0;
     marks=0.0;
    }
    int main()
    {student senior;
        senior.init();
        :
        :
    }
```

Scanned by CamScanner

A constructor for a class is needed so that the compiler automatically initialization an object as soon as it is created. A class constructor, if defined, is called whenever a prog. creates an objects of that class.

Declaration & Definition:

A constructor is defined like other member functions of a class. It can be defined either inside the class defination or outside the class defination.

```
class ABC
{ int i,j,k
  public:
   ABC();
   };
ABC::ABC() // Define
{
i=0; j=0; k=0;
}
```

// Def^n of Constr.

Do pgms of multiple constructor & Copy Constructor (Implementation)

Scanned by CamScanner

with private or protected constructor, you cannot create an object of the same class in a non-member function. however, this is allowed in the member function & friend functions.

Default constructors:

A constructor that accepts no parameter is called the default constructor.

```
class A { int i;
  public
   void get val(void);
   :
   :
  };
A O1 // uses default constructor };
O1.get val();
```

```
class Em
{ private:
    int a;
  public:
    em() // default
    { a=0; }
```

The default constructor provided by the compiler does not do anything specific. It initializes the data members by any dummy value.

Scanned by CamScanner

↳ Declaring a constructor with arugments hides the default constructor.
↳ A constructor with default arugments is equivalent to a default constructor.

Parameterized Constructors:

A constructor may be take arug. such constructor case called Parameterized

```
class ABC
{ int i, j, k
  public:
  ABC ()
  {
ABC (int a)
{ i=a; j=0; k=0;
}
ABC (int a, int b)
{ i=a; j=b; k=0;
}
ABC (int a, int b, int c)
{ i=a; j=b; k=c }
```

Parameterized constructor can take values as arguments into the constructor function when the objects are created.

Scanned by CamScanner

```
ABC ob(2);
ABC ob(2,3);
ABC ob(2,3,4);
```

With a parameterized constructor, the initial value must be passed at the time of object creation.

```
#include <iostream.h>
#include <conio.h>
class subject {
  int days;
  int subject no;
  public:
  subject (int d=123, int sn=101);
  void printsub(void)
  {
  cout << "subject NO:" << subject no;
  cout << "\n" << Days : << days "\n";
  }
  };
```

Scanned by CamScanner

```
~~subject :: Sut~~
Student :: Subject (int d, int sn)
  { cout << " constructing subject \n";
      days = d;
      subject no = sn;
  }
  class Student { int roll no;
                  float marks;
                  public:
                  Student()
                 {
                 cout << " constructing Student \n";
                     roll no = 0;
                     marks = 0.0;
                 }
                 void get val (void)
                 { cout << " enter roll no. & marks";
                   cin >> roll no >> marks;
                   cout << " \n";
                 }
```

Scanned by CamScanner

```
void print (void)
{
cout << "Roll no" << rollno;
cout << "\n marks : " << marks;
}
};
class Admission
{ subject sub;
  student stud;
  float fee;
  public:
  Admission ()
  { cout << " constructing Admission \n";
    fees = 0.0;
  }
  void print (void)
  { stud.print ();
    sub.printsub();
    cout << "fees :" << fees;
  }
};
```

Multiple Constructors with in a class

```
class integer
  { int m, n;
  Public:
    integer () — ①
    { m=0;
      n=0; }

    integer (int a, int b) — ②
    { m=a;
      n=b; }
    integer (integer &i) — ③ → receives one integer obj as an arg.
    {
      m = i.m;
      n = i.n;
    }
  };
```

* when more than one constructor function is defined in a class ⇒ Constructor is overloaded.

Scanned by CamScanner

```
int main()
{ classes();
  Admission adm;
  cout << "Back in Main"
  return(0);
}
```

Copy constructor : A copy constructor is that constructor which is used to initialize one object with the values from another object of same class during declaration]

Destructors: A destructor takes no arguments whose name is the same as the class name but is preceed by the '(~)'

```
class simple { int i,j;
                public:
                simple (int a, int b)
                { i=a; j=b; }
                ~simple()
                :
                :
```

~ → tilde

~simple() ⇒ destructor

Scanned by CamScanner

- A destructor takes no arg. & no return types can be specified for. It is called automatically by the compiler when an object is destroyed.

- A destructor clean up the storage that is no longer accessible.
- A constructor initialize an object & a destructor deinitialize an object when it is no longer needed.

Need for Destructor:

During construction of an object by the constructor, resources may be allocated for use. For ex, a constructor may have opened a file & a memory area may be allocated to it.

These allocated resources must be deallocated before the object is destroyed. Now whose responsibility is it.

Scanned by CamScanner

Declaration & Defention:

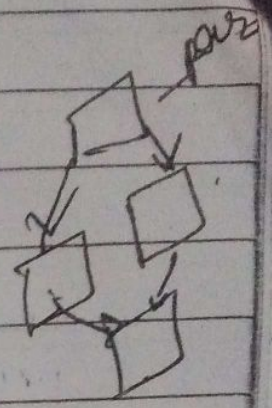
```
Class ABC
{ public:
ABC()
{ i=2;
j=3;
}
~ABC                    // destructor
{ cout<< " Destoring ;
}
};

ABC :: ~ABC()           // Destructor defined
{
cout<< " ";
}
```

↳ Destructor should be defined under the public section of a class, so that its object can be destoyed in function.

Scanned by CamScanner

Inheritance :

inheritance :- Inheritance is the capability of one class to inherit properties from another class.



Need of Inheritance :

→ capability to express the inheritance relationship.

→ idea of reusability

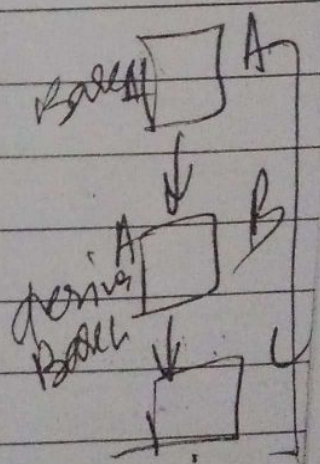
→ one learn the transitive nature of inheritance.

↳ Single inheritance :

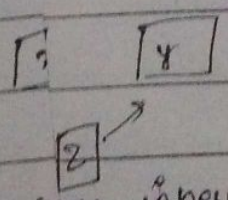
x ← Base class

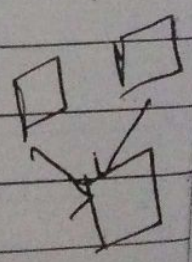
↓

Y ← sub class.



↳ Multiple inheritance



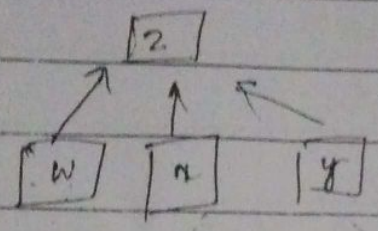


When a class inherit from multiple base classes it is known as multiple inheritance.

Scanned by CamScanner

Hierarchical Inheritance :

When many subclasses inherit from a single base classes, it is known as Hierarchical inheritance



multilevel

the transitive nature of inheritance is reflected by this form of inheritance. When a sub class inherit from a class that itself inherit from another classes.

```
class Base
{ int i, j;
  void show()
  {
  cout << "private \n" << i+j;
  }
  protected:
   void disp()
   {
```

```
cout<< "protected \n"<<i+j;
public:
base()
{ i=2; j=5;
}
void print()
{ show();
  Disp();
cout<< "public "<<i+j;
}
};
void main()
{
Base ob;
ob.disp();
ob.show();
ob.print();
  class Base1
  { public:
    void print()
    }
```

Scanned by CamScanner

```
#include <iostream.h>
# include <conio.h>
class student
{
int days;
int subjectno;
public
subject (int d=123 , in sn =101);
void printsub(xnd)
{
cout << "Subject No:" << subjectno;
cout << "\n" << "Days:" << days << "\n";
}
};
subject :: Subject(int d ;int sn)
{ cout << "continuening subject \n";
days = d;
subjectno=sn;
}
```

Scanned by CamScanner

vishant.aggarwal@gmail.com
vishant.aggarwal@yahoo.com
rajpal.arjun@yahoo.in

```
class Student
{ int rollno;
  float marks;
public:
  Student()
  { cout<<"constructing students\n";
    rollno=0;
    marks=0.0;
  }
  void getval(void)
  { cout<<"enter rollno. & marks";
    cin>>rollno>>marks;
    cout<<"\n";
  }
  void print(void)
  { cout<<"rollno:"<<rollno;
    cout<<"\n"<<"marks"<<marks;
  } };
```

```
class Admission
{ subject sub;
  Student stud;
  float fees;
Public:
  Admission()
  { cout<<"constructing Admission";
    fees=0.0;
  }
  void print(void)
  { stud.print()
    sub.printsub();
    cout<<"Fees:"<<fees<<"\n";
  } };
int main()
{ classes();
  Admission Adm;
  cout<<"\n Back in main \n";
  return 0; }
```

CLASSES
&
OBJECTS

Scanned by CamScanner

classes and objects:

Classes: A class is way to bind the data describing an entity & its associated functions together.

Two parts of class specif^n :- (a) class declaration
(b) class fun def^{n}s.

syntax:

```
class class_name
{
  private:                    //can be accessed
      [variable declarations;]  only within the
      [function declarations;]  class.
  protected:
      [variable declarations;]
      [function declarations;]
  public:
      [variable declarations;]
      [function declarations;]
};
```

E.g.,

```
class item
{ int number;          //var declaration
  float cost;
public:
  void getdata (int a, float b);   //fun declaration.
  void putdata(void);
};
```

Scanned by CamScanner

Declaration of Classes:

1. Data Members :- are the data-type properties that describe the characteristics of a class. There may be zero or more data members of any type in a class.

2. Member function: are the set of opr that may be applied to objects of that class. There may be zero or more member functions for a class. They are reffered to as the class interface.

3. Prog. Access levels: that control access to members from within the prog. These levels are private, protected or public.

4. class Tagname: that serves as a type specifier for class using which objects of this class type can be created.

class definition: which describes the components members both data members & function members of the class.

Scanned by CamScanner

b. class method definitions which describe how certain class member func. are implemented.

Need for classes:

classes are needed to represent real world entities that not only have data type properties but also associated opr (their behaviours).

Ex:

```
class salesman {
        int salesman_no;
        char salesman_name;
        int product_no;
        float target;
    public:
        float add_mon_sale (float sales);
        float calc_comm (void);
        void prn_com (void);
    };
```

↳ if no label is specified then by default, members are private.

The class methods definition:

1. outside the class definition.
2. Inside the class definition.

↳ If member function is very small that it can defined inside its self such function definition are considered by default to be line. The inline property is only hint to the compiler to insert the code for the body of the function at place where it called there by saving head of function call.

```
class ABC
{ private:
    int a;
  public:
    void readVar()
    { cin >> a;
    }
    void dispvar()
    { cout << a;
    }
};
```

Scanned by CamScanner

Outside the class declaration:

If a member function is large than it should be declare outside the prototype of such function, however must be provided in the class defination.

```
class x
{ int a;
  public:
   void readvar();
   void dispvar();
};
void x::dispvar()
{ cout<<a;
}
void x::readvar()
{ cout<<a;
}
```

Syntax:-

⇒ return type class name :: funname (arg declaration)
{ function body
}

:: → scope resolution operator :- tells the compiler that the function funame belongs to the class classname.

Scanned by CamScanner

? Simple situation where in we are interested to write / object that convert a temp. given C° to F° & vice-versa.

```
class tempconvert
{
    private:
        float C, F;
    public:
        float CtoF(float);
        float FtoC(float);
    };
float temp.convert::C toF(float a);
    { F=1.8*a+32;
      return F;
    }
float temp convert::F to C(float b);
    { C= (b-32)/1.8;
      return C;
    }
```

Scanned by CamScanner

Creating Object

An object is an instance of a class & it can be created of its class type by using the class name.

Syntax :-

Class name object name;

Eg class ABC
{ ___
};
public: int D;
ABC P;
P.D = 10;
P.C = 10 = error
because C is private.

? Write a prog. convert Temp. F to C & Vice-versa

```
#include <iostream.h>
# include <conio.h>
class tempconvert
{ float C,F;
  public:
  float CtoF(float);
  float FtoC(float);
};
float tempconvert :: CtoF(float a)
{
  F=1.8 * a+32;
  return F;
}
```

* ⇒ format for accessing class members :- Create Objects

Object name . function name (actual args);

```
float tempconvert::ftoC (float b)
{
    C = (b - 32)/1.8;
    return C;
}
void main()
{
temp convert ctemp;
float val, res;
clrscr();
cout<<"\n enter the temp. in centigrade";
cin>>val;
res = ctemp.CtoF (val);
cout<<" the temp. in fahenheit";
cout<<"\n Enter the temp. in fahenheit";
cin>>val;
res = ctemp.FtoC (val);
cout<<"\n the temp in centigrade"<<res;
getch();
}
```

Scanned by CamScanner

Array with in class

A class can have array as its member variable

Q. Prog an accept & print marks of 5 subject having array inside it.

```
# include <iostream.h>
# include <stdio.h>
# include <conio.h>
class student
{
char name[21];
float subject[5];
public:
  void readdata();
  void dispdata();
};
void student :: readdata()
{
cout<<"\n enter the name:";
gets(name);
for(int i=0; i<=5; i++)
```

Scanned by CamScanner

```
{
cout<<"\n enter the marks of "<<i+1<<"sub";
cin>>subjects[i];
}
}
void student :: dispdata()
{
cout<<"\n Name:"<<name;
for (int i=0; i<5; i++)
{
cout<<" marks of subject "<<i+1<<"="<<subject[i];
}
}
void main()
{
student s1;
clrscr();
s1.readdata();
s1.~~read~~disp();
getch();
}
```

Scanned by CamScanner

Public members :

are the members that can be used by any function.

```
class x
{ public :
  int a;
  int sqr(int a)
  {
  return a*a;
  }
  } o1;
void main ()
{ int b;
o1.a=10
b=o1.sqr(15);
}
```

Private Members :

Are the class members that hidden from the outside world.

```
class x
{ private :
```

A private member function can only be called by another function i.e. in a member of its class.

*Even an Object cannot invoke a private fun using (dot.) operator.

```
class sample
{ int m;
  void read(void); //private
Public:
  void update(void);
  void write(void);
}; s1.
```

s1.read ( ); → won't work

```
void sample::update (void)
{ read ();
}
```

//simple call, no object is used.

```
int a;
int sqr(int a)
{return a*a;
}
public:
int b;
int twice (int i)
{
return 2*i;
}
int tsq (int i)
{ int p= sqr(i); // private function being invoked by a member func.
int q= twice(p);
return q;
}
};
X O1;
int main()
{ O1.b=5; // Public member.
O1.a=2; // Wrong a is private member.
O1. twice(10); // ok twice() is a public member fen.
```

o/. sqr(10); // Wrong sqr() Private members function
// & hence can be accessed only by
other members functions of the same class

o/. tsq(10); // The function tsq() automatically
invokes sqr() &
// twice() functions. sqr() being a
private function can.
// only be invoked by other members fun. i.e
// twice() & tsq();

→ When a members function is called by another
members func. of the same class, it is known as
nesting of members functions as in tsq().

Private members provide data hiding by preventing
you from accessing the data directly.

Pointers to Members:

```
class A
{Private: int m;
Public:
void show();
};
int A::*ip = &A::m;
```

⇒ ip is a pointer that acts like a class member i.e it must be invoked with a class object.

⇒ A::* → "Pointer-to-member of A class".

⇒ &A::m → "address of the m member of A class"

Scanned by CamScanner

Protected Members:

↳ are the members that can be used only by member func. & friends of the class in which it is declared.

↳ It is similar to private members that they cannot be accessed by non-member functions.

↳

The scope Rules & classes

Global class

A class is said to be global class if its definition occurs outside the bodies of all functions in a prog.

↳ Which means that objects of this class type can be declared from anywhere in the prog.

```
# include <iostream.h>
class X        → Global class type X
{
:
};
X O1;          → Global object O1 of type X
```

Scanned by CamScanner

```
int main ()
  {
    X o2;   → local object o2 of type x.
  }
void fun(void)
  {
    X o3;   → local object o3 of type x.
    :
  }
```

B. local class :

A class is said to be local class if its definition occurs inside a function body, which means that objects of this class type can be declared only within the function that defines this class type.

```
# include <iostream.h>
:
int main()
{
  class y   → local class type y
  {
    :
  };
  y ob1;   → local object ob1 of type y
```

Scanned by CamScanner

Global Object:

An object is said to be global object if it declared outside all the function bodies & it means that this object is globally available to all func. in prog. i.e this object can be used anywhere in the prog.

```
#include <iostream.h>
class X                    // Global class type
{ public:
    int a;
    void fc(void);
};
X ob1;
int main()
{ ob1.a=10; // valid ob1 is globally available
  ob1.fc ();; //valid
  :
}
void func1(void)
{ ob1.a=20;
  ob1.fc()
  :
}
```

Local Object :

An object is said to be a local object if it is declared within a func^n, which means that this object is locally available to the function that declares it & it can't be used outside the function declaring it.

```
#include <iostream.h>
class X                    // Global class
{public:
    int a;
    void f(void);
};
int main()
{ class Y                  // Local class
  {public:
    int i;
    void afun(void);
  }; X ob1;                // Local object ob1 of global class type X
     Y ob2;                // Local object ob2 of local class Y
  ob1.a=5;
  ob1.f();
```

Scanned by CamScanner

```
    Ob2.i=15;
    Ob2.afunc();
    :
}

void func1(void)
    X Ob3;          // local object Ob3
    Y Ob4;          // invalid. Y class type is not available to main.
    Ob3.a=25;       // Valid
    Ob3.fc();       // Valid
    Ob1.a=10;       // invalid. Ob1. is not available to func().
    Ob2.afunc();    // invalid. Ob2 also not available to func().
```

13 Friend function

A friend func. is a non-member function that granted access to a class's private & protected member a friend class is a class whose member func. can access another class private & protected members.

```
class sample
{ private:
    :
```

* Using friend: private members can be accessed from outside the class means ie a non member function can access the private data of a class

* function def^n 'friend' do not use keyword 'friend'

Scanned by CamScanner

```
public :
:
friend void check(void);
};
void check(void)
```

* A friend function can be called by reference. In this case, local copies of the object are not made. Instead a pointer to the object is passed and the called function directly works on the actual object used in the call.

When friend declaring a friend function the keywords friends appear in the prototype declaration of the function inside the class definition but not in the actual function declaration + definition when

↳ A func. may be declared friend of more than one class.

↳ it does not have the class scope, rather it depends upon its original declaration & definition.

↳ it does not require an object for invoking it. It can be invoked like a normal function.

↳ Since it is not a member func. (it is a friend func.) it cannot access the members of the class directly & has to use an object name & membership operator (.)

↳ A member func. of the class operates upon the members of the object used to invoke it, while a friend

↳ members of one class can be friend functions of another class. (using ::).

```
class X
{
  int fun1();        // member function of X
};
class Y
{
  friend int X :: fun1();   // fun1 of class X is now a friend of Y class.
};
```

function operates upon the object(s) passed to its arguments.

Scope (::) Resoulution operator

↳

Scope Resoulution is useful to distinguish b/w class membre name & other name. if global name & class membre are same name then the global variable become hidden.

```
#include<iostream.h>
#include<conio.h>
int a=10;
void main()
{
clrscr();
int a=15;
cout<<"a=" <<a;
cout<<"\n ::a=" <<a;
getch();
}
```

O/P a=15 , ::a=10

Memory Allocation of object :

- memory space for object is ~~allowe~~ allocated when they are declared & not when the class is defined.
- member func. are created & placed in array space only once when the class is defined. The memory space is allocated for objects data member only when the objects are declared. No separate space is allocated for member function when object are created.

Ex:

```
#include<iostream.h>
#include<conio.h>
class employee
{
char name[20];
int emp_code;
char address[50];
public:
void readdata();
void dispdata();
};
```

Scanned by CamScanner

(5)

```
void employee :: readdata()
{
cout<<"\n Info. of empolyee";
cout<<"\n Name:";
cin>>name; Or gets (name);
cout<<"\n employee code";
cin>>emp_code;
cout<<"\n Add";
gets (address);
}
void employee :: display data()
{
cout<<"\n Info employee";
cout<<"\n Name:"<< name;
cout<<"\n employee:" << employe code;
cout<<"\n Address:"<< address;
}
void main()
{ employee emp[20];
clrscr();
```

Scanned by CamScanner

```
for(int i=0; i<20; i++)
{
  emp[i].readdata();
  for(i=0; i<20; i++)
  emp[i].display();
}
}
```

<u>Question:</u> Define a class student with the following specifications:

private members of class student

adm no	integer
sname	20 characters
eng, math, science	float
total	float
ctotal()	A function to calculate eng+math+science with float return type.

public member func. of class student

↳ Takedata() function to accept values for admno, sname, eng, math, science & invoke ctotal() to calculate total.

↳ Showdata() function to display all the data members on the screen.

Scanned by CamScanner

(25)

```
class Student
{ int admno;
  char sname[21];
  float eng, math, science;
  float total;
  float ctotal()
  { float tot = eng+math+science;
    return tot;
  }
  public:
  void Takedata();
  void showdata();
};
void student::Takedata()
{
cout << "\n enter Admission No:";
cin>> admno;
cout << "\n enter Name";
cin.getline(sname,21);
cout << "\n marks";
cout << "English:";
```

Scanned by CamScanner

```
cin>> eng;
cout<< "Maths:";
cin>>math;
cout<< "Science:";
cin>> science;
cout<< endl;
total = total();
}
void student :: show data()
{ cout<< "Admission No:" << admno<<endl;
  cout<<"\n Name :" << sname<<endl;
  cout<<"\n\t marks\n";
  cout<<"English :" << eng << \t maths : " << math
      <<"\t science:" << science<<endl;
  cout<<"\n total:" << total<<endl;
}
```

Scanned by CamScanner

Example: Define a class serial in C++ with the following specifications:

private members of class serial

- ↳ serialcode integer
- ↳ Title 20 characters
- ↳ Duration float
- ↳ Noofepisode integer

public member function of class serial.

- ↳ A constructor function to initialise Duration as 30 & Noofepisode as 10.
- ↳ Newserial() function to accept values for serialcode & Title.
- ↳ Otherentries() function to assign the values of Duration & Noofepisodes with the help of corresponding values passed as parameters to this function.
- ↳ display() function to display all the data members on the screen.

Scanned by CamScanner

```
class serial
{ int serialcode;
  char Title[20];
  float Duration;
  int Noofepisodes;
  public:
   serial()
  { Duration = 30;
    Noofepisodes = 10;
  }
  void Newserial()
  { cout<<"\n enter the code for serial";
    cin>>serial code;
    cout<<"\n enter the title of a serial";
    gets(Title);
  }
  void otherentries(float dur, int noe)
   { Duration = dur;
     Noofepisode = noe;
   }
```

Scanned by CamScanner

28

```
void display()
{ cout<<"\n serial code :"<<serial code;
  cout<<"\n Title :<< Title;
  cout<<"\n Duration : << Duration;
  cout<<"\n No. of episodes :"<< No.of episodes;
}
};
```

Scanned by CamScanner

Inheritance:

↳ Inheritance is the capability of one class to inherit properties from another class.

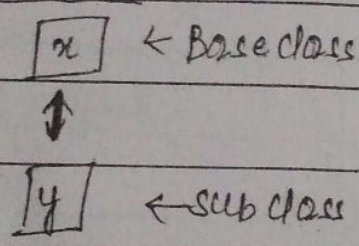
Need of Inheritance

↳ Capability to express the inheritance Relationship

↳ idea of Reusability

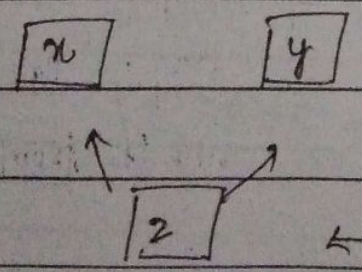
↳ one reason is transitive nature of inheritance.

1. SINGLE Inheritance:



When a subclass inherits only from one base class, it is known as a single inheritance.

2. Multiple Inheritance



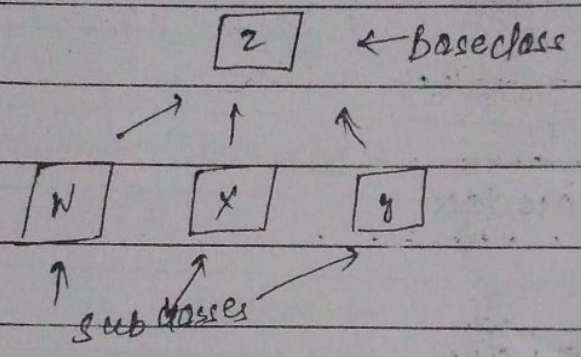
Scanned by CamScanner

(2)

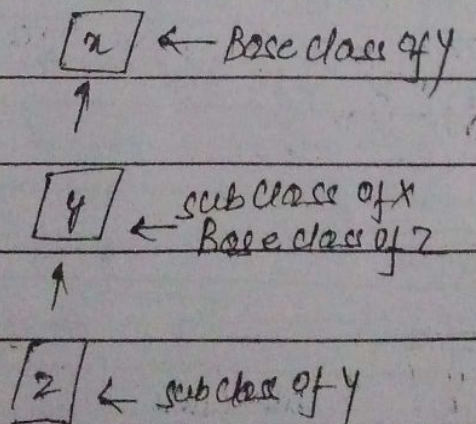
(b) Multiple Inheritance when a subclass inherit from multiple base classes it is known as multiple inheritance.

3. Hierarchical Inheritance:

When many subclasses inherit from a single base classes, it is known as Hierarchical inheritance.



4. Multilevel:

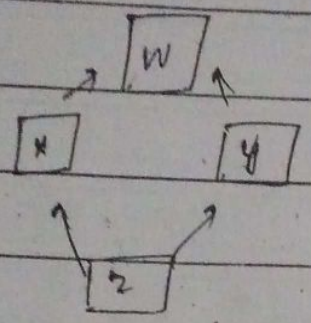


The transitive nature of inheritance is reflected by this form of inheritance. When a subclass inherits from a class that itself inherits from another class.

Scanned by CamScanner

5. Hybrid Inheritance:

When a subclass inherits from multiple base classes & all of its base classes inherit from a single base class, its form of inheritance.



ex:

```
class base
{int i, j;
 void show()
 {
 cout << "private \n" << i+j;
 }
 protected:
  void disp()
  {
  cout << "protected \n" << i+j;
  }
 public:
```

(8)

```
base()
{i=2;j=5;
}
void print()
{show();
disp();
cout<<"public\n"<<i+j;
}
};
void main()
{
Base ob;
ob.disp();
ob.show();
ob.print();
}
```

Scanned by CamScanner

ex: class derived : public Base

```
{ void copy()
{
base::print();
}
};
void main()
{ base ob;
  derived obl;
  obl.print();
  obl.copy(); // is accessible because it is private fn.
```

↳ SINGLE INHERITANCE

Syntax:

```
class derived_classname : visibility-mode baseclassname.
{
:
};
```

↳ MULTI INHERITANCE

```
class derived-classname : vis-mode base1, vis-mode base2.
{
:
};
```

Scanned by CamScanner

(3) #

```
Class Base1
{ public:
  void print()                    Base1
{
cout << "Base class";
}
};

Class Base2
{
public:
void show()                       Base2
{
cout << "printing base 2";
}
};
class derived: public Base1 , public Base2;
{ public:
    void show1()
    {
    cout << "show";
    }
};
```

Scanned by CamScanner

```
void main()
{
derived ob;
ob.print();
ob.show();
}
```

O/P

Base class 1

printing class 2.

Ex:

```
class Base
{ public:
  int a;
  Base()
  { a=10;
  }
};
class d1 : public Base
{ public:
  int b;
  d1()
{ b=20;
} };
```

Scanned by CamScanner

(5)

```
class d2 : public d1
  { public :
    void sum ()
  {
  int total
  total = a+b ;
  cout << total ;
  }
  };
void main ()
{ d2 ob;
  ob.sum () ;
  getch () ;
  }
```

O/P

30

Scanned by CamScanner

Function Overloading

```
#include<iostream.h>
class print data
{
public:
void print (int i)
{
cout<<"Printing int: "<<i<<endl;
}
void print (double f)
{
cout<<"Printing Float: "<<f<<endl;
}
void print (char* c)
{
cout<<"printing characters"<<c<<endl;
}
};
```

Scanned by CamScanner

```
int main(void)
{
printdata pd;
  pd.print(5);
  pd.print(500.263);
  pd.print("Hello");
  return 0;
}
```

Visibility

Public Inheritance

Base access specifier	Derived access specifier	Derived class access?	Public access?
Public	Public	Yes	Yes
Private	Private	No	No
Protected	Protected	Yes	No

Private Inheritance

Base Acess specifier	Derived access specifier	Derived class acess?	Public access?
Public	Protected	Yes	No
Private	Private	No	No
Protected	Protected	Yes	No

Scanned by CamScanner

Protected Inheritance

Base access specifier	Derived access specifier	Derived class access?	Public access?
Public	Protected	Yes	No
Private	Private	No	No
Protected	Protected.	NO	NO

Scope Rules of classes

	Elements	Scope	Description
class	Global	Global	This class type is globally available to all the functions with in a prog. ∴ the objects of this class type can be created from any function within the prog.
	local	local	This class type is locally available to the function in which the class defn occurs. This objects of the class type can be created only within the function that defines this class type.

Scanned by CamScanner

Object	Global	Global scope	: This object can be used anywhere in the prog by any function.
	local	local scope	: This object can be used only ~~by the memes within the~~ function that declares it.
class members:	Private :	class scope	: These members can be accessed only by the members functions of the class. These can not be accessed directly by using object.
	Protected:	-do-	-do-
	Public:	Global for global objects local for local objects	: This scope of public members depend upon the referencing objects. If the referencing object is local, the scope of public members is local. If the referencing object is global, the scope of public member is global.

Scanned by CamScanner

Accessibility in Public Inheritance

Accessibility	Private	Protected	Public
Accessible from own class?	Yes	Yes	Yes
Accessible from derived class?	NO	Yes	Yes
Accessible outside derived class?	NO	NO	Yes

Accessibility in Protected Inheritance

Accessibility	Private	Protected	Public
Accessible from own class?	Yes	Yes	Yes
Accessible from derived class?	NO	Yes	Yes
Accessible from outside derived class?	NO	NO	NO

Accessibility in Private Inheritance

Accessible from own class?	Yes	Yes	Yes
Accessible from derived class?	NO	Yes	Yes
Accessible from outside derived class?	NO	NO	NO.

Scanned by CamScanner

Friend Function :→ if a function is defined as a friend function then, the private & protected data of class can be accessed from that function. The compiler knows a given function is a friend function by its keyword friend.

→ The declaration of friend function should be made inside the body of class starting with keyword friend.

```
#include<iostream.h>
class Distance
{ private:
    int meter;
public:
  distance(): meter(0) { }
  friend int function (Distance)
    };
int func(distance d)
{ d.meter=5;
    return d.meter;
} int main()
  distance D;
  Cout<<"distance:"<<fun(D);
  return 0;
  }
```

[class class_name
{ friend return type function_name (arguments/s);
... }

CLASSTIME Page No.
Date / /

PRE - PROCESSORS

Scanned by CamScanner

14 THE PREPROCESSOR

Key Terms

Preprocessor | Macro substitution | Conditional Compilation | Stringizing Operator | Macro call

14.1 INTRODUCTION

C is a unique language in many respects. We have already seen features such as structures and pointers. Yet another unique feature of the C language is the *preprocessor*. The C preprocessor provides several tools that are unavailable in other high-level languages. The programmer can use these tools to make his program easy to read, easy to modify, portable, and more efficient.

The preprocessor, as its name implies, is a program that processes the source code before it passes through the compiler. It operates under the control of what is known as *preprocessor command lines or directives*. Preprocessor directives are placed in the source program before the main line. Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions (as per the directives) are taken and then the source program is handed over to the compiler.

Preprocessor directives follow special syntax rules that are different from the normal C syntax. They all begin with the symbol # in column one and do not require a semicolon at the end. We have already used the directives **#define** and **#include** to a limited extent. A set of commonly used preprocessor directives and their functions is given in Table 14.1.

Table 14.1 *Preprocessor Directives*

Directive	*Function*
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies the files to be included
#ifdef	Test for a macro definition
#endif	Specifies the end of #if.
#ifndef	Tests whether a macro is not defined.
#if	Test a compile-time condition
#else	Specifies alternatives when #if test fails.

Scanned by CamScanner

These directives can be divided into three categories:

1. Macro substitution directives.
2. File inclusion directives.
3. Compiler control directives.

14.2 MACRO SUBSTITUTION

Macro substitution is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. The preprocessor accomplishes this task under the direction of **#define** statement. This statement, usually known as a *macro definition* (or simply a macro) takes the following general form:

#define *identifier string*

If this statement is included in the program at the beginning, then the preprocessor replaces every occurrence of the **identifier** in the source code by the string. The keyword **#define** is written just as shown (starting from the first column) followed by the *identifier* and a *string*, with at least one blank space between them. Note that the definition is not terminated by a semicolon. The *string* may be any text, while the *identifier* must be a valid C name.

There are different forms of macro substitution. The most common forms are:

1. Simple macro substitution.
2. Argumented macro substitution.
3. Nested macro substitution.

Simple Macro Substitution

Simple string replacement is commonly used to define constants. Examples of definition of constants are:

```
#define    COUNT      100
#define    FALSE      0
#define    SUBJECTS   6
#define    PI         3.1415926
#define    CAPITAL    "DELHI"
```

Notice that we have written all macros (identifiers) in capitals. It is a convention to write all macros in capitals to identify them as symbolic constants. A definition, such as

```
#define  M  5
```

will replace all occurrences of M with 5, starting from the line of definition to the end of the program. However, a macro inside a string does not get replaced. Consider the following two lines:

```
total = M * value;
printf("M = %d\n", M);
```

These two lines would be changed during preprocessing as follows:

```
total = 5 * value;
printf("M = %d\n", 5);
```

Notice that the string **"M=%d\n"** is left unchanged.

Scanned by CamScanner

A macro definition can include more than a simple constant value. It can include expressions as well. Following are valid definitions:

```
#define    AREA     5 * 12.46
#define    SIZE     sizeof(int) * 4
#define    TWO-PI   2.0 * 3.1415926
```

Whenever we use expressions for replacement, care should be taken to prevent an unexpected order of evaluation. Consider the evaluation of the equation

```
ratio = D/A;
```

where D and A are macros defined as follows:

```
#define  D    45 – 22
#define  A    78 + 32
```

The result of the preprocessor's substitution for D and A is:

```
ratio = 45–22/78+32;
```

This is certainly different from the expected expression

(45 – 22)/(78+32)

Correct results can be obtained by using parentheses around the strings as:

```
#define  D    (45 – 22)
#define  A    (78 + 32)
```

It is a wise practice to use parentheses for expressions used in macro definitions.

As mentioned earlier, the preprocessor performs a literal text substitution, whenever the defined name occurs. This explains why we cannot use a semicolon to terminate the #define statement. This also suggests that we can use a macro to define almost anything. For example, we can use the definitions

```
#define    TEST     if (x > y)
#define    AND
#define    PRINT    printf("Very Good. \n");
```

to build a statement as follows:

TEST AND PRINT

The preprocessor would translate this line to

```
if(x>y) printf("Very Good.\n");
```

Some tokens of C syntax are confusing or are error-prone. For example, a common programming mistake is to use the token = in place of the token == in logical expressions. Similar is the case with the token &&.

Following are a few definitions that might be useful in building error free and more readable programs:

```
#define    EQUALS      ==
#define    AND         &&
#define    OR          ||
#define    NOT_EQUAL   !=
#define    START       main() {
#define    END         }
#define    MOD         %
#define    BLANK_LINE  printf("\n");
#define    INCREMENT   ++
```

Scanned by CamScanner

An example of the use of syntactic replacement is:

```
START
… …..
… …..
if(total EQUALS 240 AND average EQUALS 60)
INCREMENT count;
… …..
… ….
END
```

Macros with Arguments

The preprocessor permits us to define more complex and more useful form of replacements. It takes the form:

#define identifier(f1, f2, ... fn) string

Notice that there is no space between the macro *identifier* and the left parentheses. The identifiers f1, f2, … … .,fn are the formal macro arguments that are analogous to the formal arguments in a function definition.

There is a basic difference between the simple replacement discussed above and the replacement of macros with arguments. Subsequent occurrence of a macro with arguments is known as a *macro call* (similar to a function call). When a macro is called, the preprocessor substitutes the string, replacing the formal parameters with the actual parameters. Hence, the string behaves like a template.

A simple example of a macro with arguments is

#define CUBE(x) (x*x*x)

If the following statement appears later in the program

```
volume = CUBE(side);
```

Then the preprocessor would expand this statement to:

```
volume = (side * side * side );
```

Consider the following statement:

```
volume = CUBE(a+b);
```

This would expand to:

```
volume = (a+b * a+b * a+b);
```

which would obviously not produce the correct results. This is because the preprocessor performs a blind test substitution of the argument a+b in place of x. This shortcoming can be corrected by using parentheses for each occurrence of a formal argument in the *string*.

Example:

#define CUBE(x) ((x) * (x) *(x))

This would result in correct expansion of **CUBE(a+b)** as:

```
volume = ( (a+b) * (a+b) * (a+b));
```

Remember to use parentheses for each occurrence of a formal argument, as well as the whole *string*.

Some commonly used definitions are:

#define MAX(a,b) (((a) > (b)) ? (a) : (b))

#define MIN(a,b) (((a) < (b)) ? (a) : (b))

Scanned by CamScanner

```
#define    ABS(x)        (((x) > 0) ? (x) : (–(x)))
#define    STREQ(s1,s2)  (strcmp((s1,) (s2)) == 0)
#define    STRGT(s1,s2)  (strcmp((s1,) (s2)) > 0)
```

The argument supplied to a macro can be any series of characters. For example, the definition

#define PRINT(variable, format) printf("variable = %format \n", variable)

can be called-in by

```
PRINT(price x quantity, f);
```

The preprocessor will expand this as

```
printf( "price x quantity = %f\n", price x quantity);
```

Note that the actual parameters are substituted for formal parameters in a macro call, although they are within a string. This definition can be used for printing integers and character strings as well.

Nesting of Macros

We can also use one macro in the definition of another macro. That is, macro definitions may be nested. For instance, consider the following macro definitions.

```
#define    M            5
#define    N            M+1
#define    SQUARE(x)    ( (x) * (x) )
#define    CUBE(x)      (SQUARE (x) * (x))
#define    SIXTH(x)     (CUBE(x) * CUBE(x))
```

The preprocessor expands each **#define** macro, until no more macros appear in the text. For example, the last definition is first expanded into

((SQUARE(x) * (x)) * (SQUARE(x) * (x)))

Since **SQUARE (x)** is still a macro, it is further expanded into

((((x)*(x)) * (x)) * (((x) * (x)) * (x)))

which is finally evaluated as x^6.

Macros can also be used as parameters of other macros. For example, given the definitions of M and N, we can define the following macro to give the maximum of these two:

#define MAX(M,N) (((M) > (N)) ? (M) : (N))

Macro calls can be nested in much the same fashion as function calls. Example:

```
#define    HALF(x)      ( (x)/2.0)
#define    Y            HALF(HALF(x))
```

Similarly, given the definition of MAX(a,b) we can use the following nested call to give the maximum of the three values x,y, and z:

MAX (x, MAX(y,z))

Undefining a Macro

A defined macro can be undefined, using the statement

#undef *identifier*

This is useful when we want to restrict the definition only to a particular part of the program.

Scanned by CamScanner

14.3 FILE INCLUSION

An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions. This is achieved by the preprocessor directive

#include "filename"

where *filename* is the name of the file containing the required definitions or functions. At this point, the preprocessor inserts the entire contents of *filename* into the source code of the program. When the *filename* is included within the double quotation marks, the search for the file is made first in the current directory and then in the standard directories.

Alternatively this directive can take the form

#include <filename>

without double quotation marks. In this case, the file is searched only in the standard directories.

Nesting of included files is allowed. That is, an included file can include other files. However, a file cannot include itself.

If an included file is not found, an error is reported and compilation is terminated.

Let use assume that we have created the following three files:

SYNTAX.C	contains syntax definitions.
STAT.C	contains statistical functions.
TEST.C	contains test functions.

We can make use of a definition or function contained in any of these files by including them in the program as:

```
#include     <stdio.h>
#include     "SYNTAX.C"
#include     "STAT.C"
#include     "TEST.C"
#define   M      100
main ()
{
    ------------
    ------------
    ------------
}
```

14.4 COMPILER CONTROL DIRECTIVES

While developing large programs, you may face one or more of the following situations:

1. You have included a file containing some macro definitions. It is not known whether a particular macro (say, TEST) has been defined in that header file. However, you want to be certain that Test is defined (or not defined).
2. Suppose a customer has two different types of computers and you are required to write a program that will run on both the systems. You want to use the same program, although certain lines of code must be different for each system.

Scanned by CamScanner

3. You are developing a program (say, for sales analysis) for selling in the open market. Some customers may insist on having certain additional features. However, you would like to have a single program that would satisfy both types of customers.
4. Suppose you are in the process of testing your program, which is rather a large one. You would like to have print calls inserted in certain places to display intermediate results and messages in order to trace the flow of execution and errors, if any. Such statements are called 'debugging' statements. You want these statements to be a part of the program and to become 'active' only when you decide so.

One solution to these problems is to develop different programs to suit the needs of different situations. Another method is to develop a single, comprehensive program that includes all optional codes and then directs the compiler to skip over certain parts of source code when they are not required. Fortunately, the C preprocessor offers a feature known as *conditional compilation*, which can be used to 'switch' on or off a particular line or group of lines in a program.

Situation 1

This situation refers to the conditional definition of a macro. We want to ensure that the macro TEST is always defined, irrespective of whether it has been defined in the header file or not. This can be achieved as follows:

```
#include    "DEFINE.H"
#ifndef     TEST
#define     TEST 1
#endif
... ...
```

DEFINE.H is the header file that is supposed to contain the definition of **TEST** macro. The directive.

#ifndef TEST

searches for the definition of **TEST** in the header file and *if not defined*, then all the lines between the **#ifndef** and the corresponding **#endif** directive are left 'active' in the program. That is, the preprocessor directive

 # define TEST is processed.

In case, the TEST has been defined in the header file, the **#ifndef** condition becomes false, therefore the directive **#define TEST** is ignored. Remember, you cannot simply write

 # define TEST 1

because if **TEST** is already defined, an error will occur.

Similar is the case when we want the macro **TEST** never to be defined. Looking at the following code:

```
... ... ...
#ifdef      TEST
#undef      TEST
#endif
... ...
... ...
```

This ensures that even if **TEST** is defined in the header file, its definition is removed. Here again we cannot simply say

#undef TEST

because, if TEST is not defined, the directive is erroneous.

Scanned by CamScanner

Situation 2

The main concern here is to make the program portable. This can be achieved as follows:

```
... ...
... ...
main()
{
    ... ...
    ... ...
#ifdef IBM_PC
{
    ... ...
    ... ...          code for IBM_PC
    ... ...
}
#else
{
    ... ...
    ... ...          code for HP machine
    ... ...
}
#endif
    ... ..
    ... ..
}
```

If we want the program to run on IBM PC, we include the directive

#define IBM_PC

in the program; otherwise we don't. Note that the compiler control directives are inside the function. Care must be taken to put the # character at column one.

The compiler complies the code for IBM PC if **IBM-PC** is defined, or the code for the HP machine if it is not.

Situation 3

This is similar to the above situation and therefore the control directives take the following form:

```
#ifdef    ABC
        group-A lines
#else
        group-B lines
#endif
```

Group-A lines are included if the customer **ABC** is defined. Otherwise, group-B lines are included.

Scanned by CamScanner

Situation 4

Debugging and testing are done to detect errors in the program. While the Compiler can detect syntactic and semantic errors, it cannot detect a faulty algorithm where the program executes, but produces wrong results.

The process of error detection and isolation begins with the testing of the program with a known set of test data. The program is divided down and **printf** statements are placed in different parts to see intermediate results. Such statements are called debugging statements and are not required once the errors are isolated and corrected. We can either delete all of them or, alternately, make them inactive using control directives as:

```
- -
- -
#ifdef TEST
{
        printf("Array elements\n");
        for (i = 0; i< m; i++)
            printf("x[%d] = %d\n", i, x[i]);
}
#endif
- ..
- ..
#ifdef  TEST
        printf(...);
#endif
- -
```

The statements between the directives **#ifdef** and **#endif** are included only if the macro **TEST** is defined. Once everything is OK, delete or undefine the **TEST**. This makes the **#ifdef TEST** conditions false and therefore all the debugging statements are left out.

The C preprocessor also supports a more general form of test condition - **#if** directive. This takes the following form:

```
#if constant expression
{
        statement-1;
        statement-2;
        - -
        - -
}
#endif
```

The *constant-expression* may be any logical expression such as:

TEST <= 3

(LEVEL == 1 || LEVEL == 2)

MACHINE == 'A'

If the result of the constant-expression is nonzero (true), then all the statements between the **#if** and **#endif** are included for processing; otherwise they are skipped. The names **TEST, LEVEL**, etc. may be defined as macros.

Scanned by CamScanner

14.5 ANSI ADDITIONS

ANSI committee has added some more preprocessor directives to the existing list given in Table 14.1. They are:

#elif	Provides alternative test facility
#pragma	Specifies certain instructions
#error	Stops compilation when an error occurs

The ANSI standard also includes two new preprocessor operations:

#	Stringizing operator
##	Token-pasting operator

elif Directive

The #elif enables us to establish an "if..else..if.." sequence for testing multiple conditions. The general form of use of **#elif** is:

```
#if  expression 1
     statement sequence 1
     #elif    expression 2
              statement sequence 2
              .....
              .....
                  #elif    expression N
                           statement sequence N
     #endif
```

For example:

```
#if   MACHINE == HCL
      #define FILE "hcl.h"
      #elif  MACHINE == WIPRO
             #define FILE "wipro.h"

             #elif  MACHINE == DCM
                    #define FILE "dcm.h"
#endif
#include FILE
```

#pragma Directive

The **#pragma** is an implementation oriented directive that allows us to specify various instructions to be given to the compiler. It takes the following form:

#pragma *name*

where, *name* is the name of the **pragma** we want. For example, under Microsoft C,

#pragma loop_opt(on)

causes loop optimization to be performed. It is ignored, if the compiler does not recognize it.

Scanned by CamScanner

#error Directive

The **#error** directive is used to produce diagnostic messages during debugging. The general form is

#error ***error message***

When the **#error** directive is encountered, it displays the error message and terminates processing. Example.

```
#if !defined(FILE_G)
#error NO GRAPHICS FACILITY
#endif
```

Note that we have used a special processor operator **defined** along with **#if**. **defined** is a new addition and takes a *name* surrounded by parentheses. If a compiler does not support this, we can replace it as follows:

#if !defined	by	#ifndef
#if defined	by	#ifdef

Stringizing Operator

ANSI C provides an operator # called *stringizing operator* to be used in the definition of macro functions. This operator allows a formal argument within a macro definition to be converted to a string. Consider the example below:

```
#define sum(xy) printf(#xy " = %f\n", xy)
main()
{
    ...
    ...
    sum(a+b);
    ...
}
```

The preprocessor will convert the line

```
sum(a+b);
```

into

```
printf("a+b" "=%f\n", a+b);
```

which is equivalent to

```
printf("a+b =%f\n", a+b);
```

Note that the ANSI standard also stipulates that adjacent strings will be concatenated.

Token Pasting Operator

The token pasting operator **##** defined by ANSI standard enables us to combine two tokens within a macro definition to form a single token. For example:

```
#define combine(s1,s2) s1 ## s2
main()
{
printf("%f", combine(total, sales));
```

Scanned by CamScanner

Preprocessor

I Macro

① Macro Substitution

1.1 → Simple macro Substitution (# define)

1.2 → Augmented Macros

#define identifier(f_1, f_2 ... f_n) string

1.3 → Nesting of Macro

② → Undefining a Macro

#undef identifier

II Header file Inclusion

II.1 → # include <filename>

III Conditional Compilation

III.1 → #ifdef

III.2 → #ifndef

III.3 → #if

III.4 → #else

III.5 → #endif

IV Other directives

IV.1 → #undef

IV.2 → #Pragma

2.1 → #pragma Startup fun()

2.2 → #Pragma exit fun()

3.1 → #error directive

#error error message

3.2 → Stringizing operator #

3.3 → ## token Pasting operator

Scanned by CamScanner

Programs

(1) Define a macro that receives an array and the number of elements in the array as arguments. Write a program using this macro to print out the elements of an array.

(2) Write a program to illustrate the use of stringizing operator.

(3) Write a nested macro that gives the minimum of three values.

(4) Write symbolic constants for { and } and printing a blank line. Write a short program to illustrate the use of these symbolic constants.

✓(5) Define a macro with one parameter to compute the volume of a sphere. Write a program using this macro to compute the volume for spheres of radius 5, 10 and 15 meters.

Q(1) Enumerate the difference between functions and parameterized macros.

Q(2) What is a macro & how it is different from C variable name. Explain the advantages of using macro definitions in pgm.

Q3- Explain the facilities provided by the C preprocessor with examples?

Scanned by CamScanner

Q/ Distinguish b/w #ifdef
#if and
#undef

↳ Token Pasting Operator (##)

```
#define combine (S1,S2) S1##S2
main()
{ ___
  ___
  Printf("%f", combine(total, sales));
  ___
}
```

* The statement → Printf will transforms to Printf("%f", totalsales);

Eg → Consider another macro definition :

```
#define Print(i) Printf("a" #i "=%f",
                        a##i)
```

The macro will convert the statement :
Print#(5); into statement
Printf("a5 = %f", a5)

Debugging statements :-

Debugging and testing are done to detect errors in the program.

→ While the Compiler can detect syntactic and semantic errors, it cannot detect a faulty algorithm where the program executes, but in such case it produces wrong results.

→ Test data :- It is the input to test or detect errors. The program is divided into some segments and printf statements are placed in different parts to see the intermediate results. Such statements are called debugging statements. These statements are not required once the errors are isolated and corrected.

∴ C Preprocessor supports a general form of test condition → #if directive

```
#if Constant expression
{   st1;
    st2;
    ____
    ____
}#endif
```

* The constant expression may be any logical expression such as →

```
TEST <=3
(LEVEL ==1 || LEVEL ==2)
MACHING == 'A'
```

* If the result of the constant-expression is non zero (true), then all the statements between the #if and #endif are included for processing; otherwise they are skipped.

Scanned by CamScanner