PROGRAMMING IN 'C++'

INTRODUCTION:

The C++ programming language was developed at AT & T Bell Laboratories in the early 1980's by Bjarne Stroustrup. The features of C++ were a combination of the object-oriented features of a language called simula 67 and the power and elegance of C. Therefore, C++ is an extension of C with a major addition of the class construct features of simula 67.

C++ is a versatile language for handling very large programs. It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real-life application systems.

COMMENTS: In C++, a new symbol for comment '**//**' (double slash) is introduced. A comment starts with a double slash symbol and terminals at the end of the line. Note that there is no closing symbol. The 'C' comment symbols '/*' and '*/' are also valid in C++ and can be used for multi-line comments.

The main() function:

```
// Ex-1: PGM ILLUSTRATES THE FUNCTION main()
#include<iostream.h>
void main()
{
 cout<<"Hello Dear";
}
```

OUTPUT: Hello Dear

Here, #include<iostream.h> causes the contents of the file iostream.h to be added to the program. It contains declarations for the identifier *cout, cin* and operator *<<* and *>>.* This header file must be included at the beginning of all programs that use input/output statements.

In above program *cout* statement introduces two features specific to C++, cout and *<<.* The identifier *cout* is a predefined object that represents the standard output stream in C++.

A **stream** is an abstraction that refers to a flow of data.

The operator '<<' is called the *insertion* operator or *put to* operator. It inserts the contents of the variable on its right, to the object on its left. The operator '<<' has a simple interface, i.e., it is not necessary to specify the data type of the variable on its right. The insertion operation automatically assumes the data type of the variable.

Usage of the identifier 'cin':

The *cin* object is an instance of the class, istream. The class, istream is associated with the standard input device. This means that the *cin* object is capable of accepting input from the keyboard. The *cin* object also contents the input operator '*>>*', which is used to obtain data from the standard input device and place it in a variable.

Built-in Data types in C++:

char - Occupies 1 Byte - To store characters and strings
int - Occupies 2 Bytes - To store integers
float - Occupies 4 Bytes - To store numbers with decimals.

The *if..else* construct:

The if conditional construct, is followed by a logical expression in which data is compared and a decision is made, based on the result of the comparison.

Syntax:

```
if(boolean_expression)
{
  stmt(s);
}
else
{
  stmt(s);
}
```

```
// EX-2: PGM ILLUSTRATES THE if-else CONSTRUCT
#include<iostream.h>
#include<conio.h>
void main()
{
  char ch;
```

```
 clrscr();
 cout<<"Enter a character between A to Z(Upper Case): ";
 cin>>ch;
 if(ch=='A' || ch=='E' || ch=='I' || ch=='O' || ch=='U')
   cout<<"Given character is vowel";
  else
   cout<<"Given character is consonent";
 getch();
}
```

Note:

#include<conio.h> performs the function *clrscr()* and *getch()*

In an *if..else* block of statements, the condition is evaluated first. If the condition is true, that is the value is non-zero, the statements in the immediate block are executed. If the condition is false, that is the value is 0, the statements in the else block are executed.

CONTROL STRUCTURES

***do..while* loop:**

This construct executes the body of the loop exactly once, and then evaluates the test condition. If the loop condition is true, the execution is repeated, otherwise the loop ends and control is passed to the statement following the loops.

syn:

```
do
{
  stmt(s);
}while(boolean_expression);
```

```
//   EX-3: PGM ILLUSTRATES do..while LOOP
#include<iostream.h>
#include<conio.h>
void main()
{
 char ch,ans;
 int asci;
 clrscr();
 do
 {
  cout<<"\n\n\n\n\n\t\t\t   Enter an alphabet: ";
  cin>>ch;
  asci=ch;
  cout<<"\n\t\t\tThe ASCII value of "<<ch<<" is "<<asci<<endl;
  getch();
  clrscr();
  cout<<"\n\t\t  Do you want to continue... (y/n): ";
  cin>>ans;
  clrscr();
 }
 while(ans=='y' || ans=='Y');
 cout<<"\n\n\t\t\t \"THANKS FOR USING\"";
 getch();
}
```

***while* loop:**

The *while* loop is a looping construct, which executes until the evaluating condition becomes false. The evaluating condition has to be a logical expression and must return a true or false value. The variable that is checked in the Boolean expression is called the loop control variable. Sometimes, there may not be any loop control variable, for example, while(1).

syn:

```
while(boolean_expression)
{
  statement(s);
}
```

***for* loop:**

The *for* statement consists of the keyword ***for***, followed by three expressions in parenthesis separated by semicolons. They are the initialization expression, test expression and the increment/decrement expression.

syn:

```
for(initialization; test_expression; iteration)
{
  statement(s);
```

```
        }
//  EX-4: PGM TO PRINT ASCII VALUE OF 20 CHAR. USING for LOOP
#include<iostream.h>
#include<conio.h>
void main()
{
  char ch='a';
  int i, asci;
  clrscr();
  for(i=0;i<20;i++)
  {
   asci=ch;
   cout<<"\n\t The ASCII value of "<<ch<<" is "<<asci;
   ch++;
  }
  getch();
}
```

***switch..case* statement:**

C++ has a built-in multiple branch selection statement called ***switch***. This statement successively tests the value of an expression against a list of constants. When a match is found, the statements associated with that condition are executed.

```
//  EX-5: PGM ILLUSTRATES while, do..while, for AND switch STATEMENT
#include<dos.h>
#include<iostream.h>
#include<conio.h>
void main()
{
  int n1,n2,x;
  char ch='y';
  clrscr();
  while(ch=='y' || ch=='Y')
  {
   cout<<"\n\t Enter the first number: ";
   cin>>n1;
   do
    {
     cout<<"\n\t Enter the second number (It should less than first no.): ";
     cin>>n2;
    }while(n2>=n1);
    cout<<"\n\t\t 1. Add two nos.";
    cout<<"\n\t\t 2. Subtract two nos.";
    cout<<"\n\t\t 3. Multiply two nos.";
    cout<<"\n\t\t 4. Divide two nos.";
    cout<<"\n\n\t\t Enter your choice : ";
    cin>>x;
    clrscr();
    gotoxy(30,12);
    switch(x)
    {
       case 1: cout<<" SUM = "<<n1+n2;
                       break;
       case 2: cout<<" DIFFERENCE = "<<n1-n2;
                       break;
       case 3: cout<<" PRODUCT = "<<n1*n2;
                       break;
       case 4: cout<<" QUOTIENT = "<<n1/n2;
                       break;
     default : for(int i=0;i<5;i++)
                       {
                        clrscr();
                        delay(200);
                        gotoxy(26,12);
                        cout<<"INVALID CHOICE !!! ";
                        delay(300);
                       }
                       break;
    }
    cout<<"\n\n\t\t  Do you want to continue..(y/n): ";
    cin>>ch;
```

```
   clrscr();
  }
}
```

ARRAYS:

We discussed briefly about an array in 'C'. Like, in 'C++' also,

Initialization of single dimensional array is int I[5]={ 2,4,6,8,10 };
Initialization of two-dimensional character array is

```
        char arr[][8] = {"KISHORE", "NAVEENA", "VINOTHA"};
                          (OR)
        char arr[3][8] = {
                           { 'K', 'I', 'S', 'H', 'O', 'R', 'E', '\0' },
                           { 'V', 'I', 'N', 'O', 'T', 'H', 'A', '\0' },
                           { 'N', 'A', 'V', 'E', 'E', 'N', 'A', '\0' }
                          };
```

```
//  EX-6: PGM ILLUSTRATES TWO-DIMENSIONAL CHARACTER ARRAY
#include<iostream.h>
#include<conio.h>
void main()
{
  char arr[][8] = {
                    {'V', 'I', 'N', 'O', 'T', 'H', 'A','\0'},
                    {'N', 'A', 'V', 'E', 'E', 'N', 'A','\0'}
                  };
  clrscr();
  cout<<"\nFirst String is : "<<arr[0]<<endl;
  cout<<"Second String is : "<<arr[1]<<endl;
  cout<<"Third character in first string is: "<<arr[0][2]<<endl;
  getch();
}
```

```
//  EX-7: PGM ILLUSTRATES TWO DIMENSIONAL INTEGER ARRAY
#include<iostream.h>
#include<conio.h>
void main()
{
  int month, nod;
  int days[12]={ 31,28,31,30,31,30,31,31,30,31,30,31 };
  clrscr();
  lbl:
  cout<<"\n Enter the month (1 to 12): ";
  cin>>month;
  if(month>12)
   {
    cout<<"\nINVALID MONTH";
    goto lbl;
   }
   cout<<"\n Enter the day(1 to 31): ";
   cin>>nod;
   if(nod>31)
    {
     cout<<"\nINVALID DATE";
     return;
    }
   for(int j=0;j<month-1;j++)
   nod+=days[j];
   cout<<"\n\t Total days from 1st Jan: "<<nod<<endl;
   getch();
}
```

POINTERS:

At the time of execution itself, the memory allocates the required space. This can only be achieved with the help of a special type of variable, a *pointer variable*, also called a ***pointer***. A *pointer* is a variable that stores the memory address of another variable.

Advantages:

- Pointers allow direct access to individual bytes in the memory. Thus, data in the memory is accessed faster than through an ordinary variable. This speeds up the execution of programs.
- Pointers allow the program to allocate memory dynamically, only when required, with the help of the *new* operator. They also allow the program to free the memory when it is no longer required. This is done with the help of the *delete* operator. You will learn about these operators later.

Uses:

- In database program, where the no. of records to be stored is unknown at the time of designing the database.
- In word-processing programs (the no. of words stored by the program in memory is unknown at the time of writing the program. This is also known as the *compile-time* of a program. *Run-time* implies that the program is executing).
- Mostly used in any software like an operating system, which interacts directly with the hardware.

```
//  EX-8: PGM ILLUSTRATES THE USE OF POINTERS
#include<iostream.h>
#include<conio.h>
void main()
{
  int a,b;
  int *pt;
  a=50;
  b=80;
  clrscr();
  pt=&a;
  cout<<*pt<<endl;
  *pt = *pt+b;
  cout<<*pt<<endl;
  cout<<a<<endl;
  getch();
}
```

POINTERS AND ARRAYS:

The name of an array is resolved by the compiler to yield the *base address* (address of the element number[0] of the array). The base address of an array cannot be changed because it is a constant value.

Assume that a pointer is assigned the address of the starting element of an integer array. If the value in the pointer is incremented by one, the pointer will point to the next element of the array, as follows.

*(ptr+1)

is resolved by the compiler to

ptr+(1*sizeof(int))

Ex: ptr=1010+(1*2) (Assume that the address of the first element is 1010)
ptr=1012 (Address of the second element)

```
//  EX-9: PGM ILLUSTRATES POINTERS AND ARRAYS
#include<iostream.h>
#include<conio.h>
int n[5]= { 5,10,15,20,25 };
void main()
{
  int *ptr;
  ptr=n;
  clrscr();
  cout<<*(ptr+2)<<endl;
  getch();
}
```

DYNAMIC MEMORY ALLOCATION

THE NEW AND DELETE OPERATOR:

In C++ the operator *new* allocates memory dynamically. The *new* operator determines the type of the receiving pointer and converts it's return value appropriately. Memory is allocated. To release the memory allocated, C++ uses the *delete* operator.

The *new* operator, when used with the pointer to a data type, allocates memory for the item and assigns the address of that memory to the pointer. The *delete* operator does the reverse. It de-allocates the memory allocated to the variable.

Syn: data_type *var = new data_type;

```
                    (or)
          data_type *var;
          var = new data_type;
//EX-10:   PGM ILLUSTRATES NEW & DELETE OPERATOR FOR PRIMITIVE DATA TYPES 'INT' AND 'FLOAT'
#include<iostream.h>
#include<conio.h>
void main(void)
{
  int *i=new int;
  *i=10;
  float *f=new float(10.152);
  clrscr();
  cout<<endl<<"Value of i = "<<*i;
  cout<<endl<<"Value of f = "<<*f;
  delete i;
  delete f;
  getch();
}


// EX-11: PGM ILLUSTRATES NEW & DELETE OPERATOR FOR PRIMITIVE DATA TYPE  'CHAR'
#include<iostream.h>
#include<conio.h>
void main()
{
 char *name;
 name=new char[20];   // It allocates 20 character space to the var 'name'
 clrscr();
 cout<<"Enter your name: "<<endl;
 cin>>name;
 cout<<name;
 delete[] name;
 getch();
}
```

In above example, we learned about, how to allocate and de-allocate memory dynamically for built-in data types. This can also be done for user-defined data types.

```
//  PGM-12:  ILLUSTRATES NEW & DELETE OPERATOR FOR USER DEFINED DATA TYPE  'CLASS'
#include<conio.h>
#include<iostream.h>
#include<process.h>
class dyn
{
 private:
  int x;
  int y;
 public:
  void set(int a,int b)
  {
   x=a;
   y=b;
  }
  void get()
  {
   cout<<endl<<"x = "<<x;
   cout<<endl<<"y = "<<y;
  }
};

void main()
{
  clrscr();
  dyn *ptr=new dyn;
  if(!ptr)
  {
   cout<<endl<<"ERROR: In Memory Allocation";
   exit(2);
  }
  ptr -> set(14,10);          // Here, '->' is a structure pointer operator
```

```
 ptr -> get();
 delete ptr;
 getch();
}
```

Note: The operator **->**, used for accessing class members with pointers.

FUNCTIONS:

A function is a set of statements that perform a specific task. The specified task is repeated each time when the function is called. In large computing tasks, functions are break into smaller ones. They work together to accomplish the goal of the whole program. Moreover, functions increase the modularity of the program and reduce code redundancy.

Functions must be defined outside any other enclosing block where a block is delimited by braces '{ }'. The following terms are associated with functions.

* Function declaration - Introduces the function in the program.
* Function definition - It contains the function code.
* Function call - It used to execute the function.
* Function parameters - Data that is passed to the function before it starts execution.
* Function return type - Data type of the value returned after the function has completed the execution.
* Calling function - Function that executes another function.
* Called function - Function that is executed by another function.

```
// EX-13: TO FIND THE VOLUME OF CYLINDER USING FUNCTION
#include<iostream.h>
#include<conio.h>
float volume(float r, float h);
void main()
{
 float x,y,z;
 clrscr();
 cout<<endl<<"Enter the value for radius: ";
 cin>>x;
 cout<<endl<<"Enter the value for height: ";
 cin>>y;
 z=volume(x,y);
 cout<<endl<<"Volume of the cylinder: "<<z<<endl;
 getch();
}

float volume(float r,float h)
{
 float v;
 v=((1.0/3.0)*((22.0/7.0)*r*r*h));
 return(v);
}
```

getline():

The member function of the *cin* object *getline()* is used to accept data from the standard input device. It provides the flexibility of retrieving strings with embedded spaces, from the keyboard, which is not possible using the operator, '>>', of the *cin* object.

Syn: cin.getline(<string_name>,<size_of_string+1>);

Example:

```
char name[21];
cin.getline(name,21);
```

ignore():

If you need to invoke the *getline()* function immediately after a call to the operator, '>>', of the *cin* object, the new line character needs to be extracted from the input buffer. This is done by calling the *ignore()* function.

Note: The functions getline() and ignore() are comes under the header file iostream.h

COMPARISON OF STRINGS:

The function *strcmp()* compares the two strings supplies as its parameters and returns an integer value. To use this function, you need to include the file, *string.h*.

- If the first argument is less than the second, strcmp() returns a value less than zero (negative number).
- If the first argument is equal to the second, the function returns a value equal to zero.
- If the first argument is greater than the second, the function returns a value greater than zero (positive number).

Syn: strcmp(<string1>,<string2>);

```
// Ex-14: PGM ILLUSTRATES THE FUNCTION strcmp()
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{
 clrscr();
 char s1[]="GOD";
 char s2[]="GOD";
 cout<<strcmp(s1,s2)<<endl;
 char s3[]="sumi";
 char s4[]="SUMI";
 cout<<strcmp(s3,s4)<<endl;
 char s5[]="SUJI";
 char s6[]="SUMI";
 cout<<strcmp(s5,s6)<<endl;
 getch();
}
```

CLASSES AND OBJECTS:

This topic gives a brief introduction about the most important features of C++ - Classes and Objects, and proceeds to discuss them in detail. The real strength of C++ lies in the fact that it supports object-orientation. The fundamental idea behind object-orientation is to combine both data members(variables) and the functions that operate on the data into a single unit called an ***object***.

In real-life, every object having an identity (name of the object), certain attribute a behavior, and a state. For example, a book has a name (identity). It has certain attributes like color, size, content, and no, of pages. It has certain functions that can be performed on it, line reading and writing the book. An object also has a state that is governed by its attributes. For example, on changing the contents of a book, the state of a book changes. The data and functions are encapsulated into a single entity called a book. This storage of data and functions together in one entity is called ***data encapsulation***.

An object's function provides a way to access and manipulate data in the class safely. Class data or functions are kept hidden and are safe from accidental alteration or use. This kind of protection of data from external alteration, except by the use of public functions, is called ***data hiding.***

Data encapsulation and data hiding are the key terms used for describing an object-oriented language.

In C++, the functions of an object are called *member functions*. They are called methods in other object-oriented language Java.

CLASS:

The user-defined data type, *class* is a new data type that is created to solve a particular kind of problem. Once a class is created, anyone can use it without knowing the specifics of how it works or even how a class is built.

The *class* key word is used to declare a class. The braces are used to indicate the start and end of a class body. *Member variables* and *member functions* are declared inside the class body. A semicolon is used to end of the declaration.

Syn:
```
class <class_name>
{
 . . . . . .
 . . . . . .
};
void main()
{
 <class_name> <object_name>;
 . . . . . .
 . . . . . .
}
```

Access Specifiers:

An access specifier is used to determine whether any other class or function can access the member variables and functions of a particular class. C++ supports three access specifiers.

- public
- private
- protected

Public Access Specifier:

The public access specifier allows a class to expose its member variables and member functions to other functions and objects.

```
// Ex-15: PGM ILLUSTRATES THE ACCESS SPECIFIER public
#include<iostream.h>
#include<conio.h>
class rainbow
{
public:
 char *color;
 void display()
 {
  gotoxy(27,10);
  cout<<"What is Rainbow?: ";
  cin>>color;
  clrscr();
 }

 void displayout()
 {
  clrscr();
  gotoxy(32,12);
  cout<<color;
  }
};

void main()
 {
  rainbow obj;
  clrscr();
  obj.display();
  obj.displayout();
  getch();
 }
```

Note: The **.** operator is used to access member data and functions.

In above example, the variable color can be accessed in any function. The function display(), would be accessible from anywhere in the program by any other function.

Private Access Specifier:

The private access specifier allows a class to hide its member functions from other class objects and functions.

```
//  Ex-16: PGM ILLUSTRATES THE PRIVATE ACCESS SPECIFIER
#include<iostream.h>
#include<conio.h>
class rainbow
{
 private:
  char color[10];
 public:
  void accept()
  {
   cout<<"which one is a first color of the Rainbow?: ";
   cin>>color;
  }
  void show()
  {
   cout<<"The first color of the Rainbow : "<<color<<endl;
  }
};

void main()
```

```
{
rainbow vibgyor;
clrscr();
vibgyor.accept();
vibgyor.show();
/*cin>>vibgyor.color;       (This stmt. gives an error, because  private members
                              cannot be accessed outside the class definition.)*/
getch();
}
```

Note: In the above example, the functions accept() and show(), can be called from the object vibgyor, defined in main() function. Whereas the variable, color, cannot be accessed through the object vibgyor, since it is a private member variable.

The default access specifier for a class member is private, even though it has not been specified explicitly.

Ex:
```
class Rainbow
{
 char color[10];
 public:
 . . . . . .
 . . . . . .
};
```

CONSTRUCTORS AND DESTRUCTORS

CONSTRUCTORS:

Constructors are member functions of any class, is used for initialization (to initialize variable, to allocate memory).

Assume that each time an object of the date class is created, it is to be initialized with the date 01-01-2002. The date within the class (any class) cannot be initialized at the time of declaration. This is because the declaration of a class serves only as a template, and no memory is allocated at the time of declaration.

This problem can be solved by initialization of function i.e., called a constructor. This function (constructor) will be automatically invoked as soon as an object is created, i.e., the programmer does not have to specifically call the function to initialize the variables.

```
// Ex-17: PGM ILLUSTRATES THE USE OF CONSTRUCTORS
#include<conio.h>
#include<iostream.h>
class date
{
 private:
  int d;
  int m;
  int y;
 public:
  date()              // constructor
  {
   d=01;
   m=01;
   y=2002;
  }
  void display()
  {
   cout<<endl<<d<<"-"<<m<<"-"<<y<<endl;
  }
};

void main()
{
 clrscr();
 date obj;
 obj.display();
 getch();
}
```

In above example, as soon as an object, *obj* is created, the constructor will be invoked and the values d,m,y initialized to the required date 01-01-2002.

Note:

The constructor has not any parameter and does not return any value. Constructors are an exception to the general rule in C++ that all functions must specify a return type and also return a value of the specified type. Moreover, since it is to be invoked automatically. It is declared in the public section of the class.

A constructor can also take parameters. A constructor that does not take parameters is called the *default constructor* of a class. Apart from this, assume that when an object of the date class is created, the members are to be initialized with values specified by the user.

```
// Ex-18: CONSTRUCTORS – PASSING VALUES THROUGH AN OBJECT
#include<iostream.h>
#include<conio.h>
class date
{
 private:
  int d;
  int m;
  int y;
 public:
  date(int dd, int mm, int yy)
  {
   d=dd;
   m=mm;
   y=yy;
  }
  void display()
  {
   cout<<endl<<d<<"-"<<m<<"-"<<y<<endl;
  }
 };

void main()
{
 clrscr();
 date obj(1,1,2002);
 obj.display();
 getch();
}
```

Note:

A single class definition could include any no. of constructors, which will invoked depending on the type, number and sequence of parameters. This is a special feature of C++, called overloading, and is discussed in detail later.

DESTRUCTOR:

A destructor is complementary to a constructor, is used to de-initialize the objects when they are destroyed. A destructor is automatically invoked when an object of a class goes out of scope or when the memory occupied by it is de-allocated using the delete operator. Therefore, a programmer is relieved of the task of clearing the memory space occupied by a data member every time an object goes out of scope.

Declaration of destruction has same name as its class but prefixed with a ~(tilde) symbol. A class can have only one destructor. It cannot take arguments, specify a return value, or explicitly return a value. The overloading of destructor is not possible.

```
// Ex-19:PGM ILLUSTRATES DESTRUCTOR
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
class point
 {
 private:
  int x,y;
 public:
  point(int a, int b)
  {
   x=a;
   y=b;
  }
 void get()
 {
  cout<<"x="<<x<<endl;
  cout<<"y="<<y<<endl;
```

```
 }
~point()
 {
  cout<<"Job is over"<<endl;
 }
};

void main(void)
{
 clrscr();
 point p(10,20);
 p.get();
// getch();
}
```

LIFE CYCLE OF AN OBJECT:

```
// Ex-20: PGM ILLUSTRATES LIFE CYCLE OF AN OBJECT
#include<iostream.h>
#include<conio.h>
class test
{
 public:
  test()
  {
    cout<<endl<<"Constructor invoked";
  }

  ~test()
  {
   cout<<endl<<"Destructor invoked";
  }
};

test obj1;
void main()
{
//clrscr();
  cout<<endl<<"Main() begins";
  test obj2;
  {
   cout<<endl<<"Inner block begins";
   test obj3;
   cout<<endl<<"Inner block ends";
  }
  cout<<endl<<"Main() ends";
//getch();
}
```

Note: - obj1 – Global scope, it's destructor is invoked when the program ends.
- obj2 – Function scope, it's destructor is invoked when main() ends.
- obj3 – Block scope, it's destructor is invoked when inner block ends.

SCOPE RESOLUTION OPERATOR:

Assume that in a huge class with a lot of member data and methods, the member functions are defined within the class definition itself. This may make the class definition messy and unreadable. The scope resolution operator (**: :**) separates the body of the function from the body of the class(Simply we can say as, the function declared in a class, but definition is outside the class i.e., in S.R.O).

Using the scope operator, the programmer can define a member function outside the class definition, without the function losing its connections with the class itself.

Syn:

```
return_type  class_name :: function_name()
{
 . . . . . .
 . . . . . .
}
```

```
//  Ex-21: PGM ILLUSTRATES THE USE OF S.R.O.
```

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class product
{
 private:
  int pno,qty;
  char pname[25];
  float rpu;
 public:
  void setp(int mpno, char* mpname, int mqty, float mrpu);
  void getp();
  float getamt();
};

void product::setp(int mpno, char* mpname, int mqty, float mrpu)
{
 pno=mpno;
 strcpy(pname,mpname);
 qty=mqty;
 rpu=mrpu;
}

void product::getp()
{
 cout<<"\n\n\t\t"<<pno<<"  "<<pname<<"  "<<qty<<"  "<<rpu<<"  ";
}

float product::getamt()
{
 return(qty*rpu);
}

void main(void)
{
 product ob;
 clrscr();
 ob.setp(25,"HAMAM",12,9.25);
 ob.getp();
 cout<<ob.getamt()<<endl;
 getch();
}
```

INHERITANCE

Inheritance is the concept by which the properties of one entity are available to another. In C++, inheritance is the property by which the objects of a derived class possess copies of the data members and member functions of the base class.

A class that inherits or derives attributes from another class is called a *derived class* or *sub class* and the class from which it is derived is called *a base class* or *super class*. Each instance of a derived class includes most of the attributes of the base class. Hence, a derived class has a larger set of attributes than its base class.

Any class can be a base class. More than one class can inherit attributes from a single base class, and a derived class can be a base class to another class.

Syn:

```
class <base_class>
{
 . . . . . .
 . . . . . .
};
class <derived_class : <access_specifier> <base_class>
{
 . . . . . .
 . . . . . .
}
```

In the above declaration, the single colon (:) is used to specify that the class being declared is derived from another class (the name of which is specified after the colon).

Access specifier/Qualifier:

- **private** : If the members of a class declared as private, can not be accessed from its sub- classes and other classes.

- **public** : If the members of a class declared as public, can be accessed from its sub-class as well as from other classes also.
- **protected** : If the members of a class declared as protected can be accessed from the derived classes, but cannot be accessed from anywhere in other classes.

If the *private* qualifier is used when deriving a class, all
private members of base class becomes *private* in derived class.
public members of base class becomes *private* in derived class.
protected members of base class becomes *private* in derived class.

If the *public* qualifier is used when deriving a class, all
private members of base class becomes *private* in derived class.
public members of base class becomes *public* in derived class.
protected members of base class becomes *protected* in derived class.

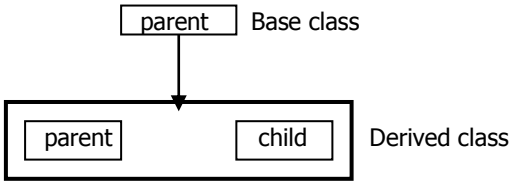
If the *protected* qualifier is used when deriving a class, all
private members of base class becomes *private* in derived class.
public members of base class becomes *protected* in derived class.
protected members of base class becomes *protected* in derived class.

TYPES OF INHERITANCE:

1. Single Inheritance
2. Multiple Inheritance
3. Multi-level Inheritance

SINGLE INHERITANCE:

If the derived class inherits the properties from one base class , is called a *single inheritance*.



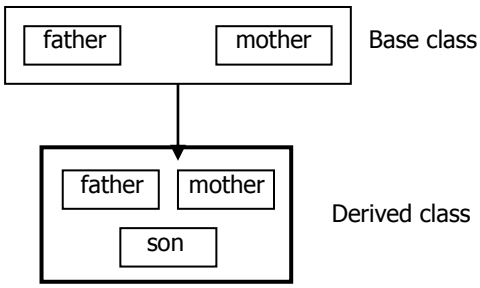
Consider, parent having the properties in Base class and inherited to Derived class. Now, Derived class having the properties of Base class and properties of child, which is exist already in derived class.

```
// Ex-22: PGM ILLUSTRATES SINGLE INHERITANCE
#include<conio.h>
#include<iostream.h>
class parent
{
 private:
  int a,b;
 public:
  void set()
  {
   a=30;
   b=60;
  }
  void get()
  {
   cout<<" "<<a<<" "<<b<<" ";
  }
};

class child : public parent
{
 private:
  int c;
 public:
  void set1()
  {
   c=90;
  }
  void get1()
  {
   cout<<c;
  }
```

```
};

void main()
{
 clrscr();
 child ob;
 ob.set();
 ob.set1();
 ob.get();
 ob.get1();
 getch();
}

// Ex-23: PGM ILLUSTRATES SINGLE ILLUSTRATES (PASSING VALUE THROUGH AN OBJECT)
#include<conio.h>
#include<iostream.h>
class parent
{
 protected:
  int x;
};

class child : public parent
{
 private:
  int y;
 public:
  void set(int a, int b)
  {
   x=a;
   y=b;
 }
  void get()
  {
   cout<<"x= "<<x<<"\ny="<<y;
  }
};

void main()
{
 clrscr();
 child ob;
 ob.set(5,10);
 ob.get();
 getch();
}
```

MULTIPLE INHERITANCE:

If the derived class inherits the properties from more than one base class, is called *multiple inheritance*.

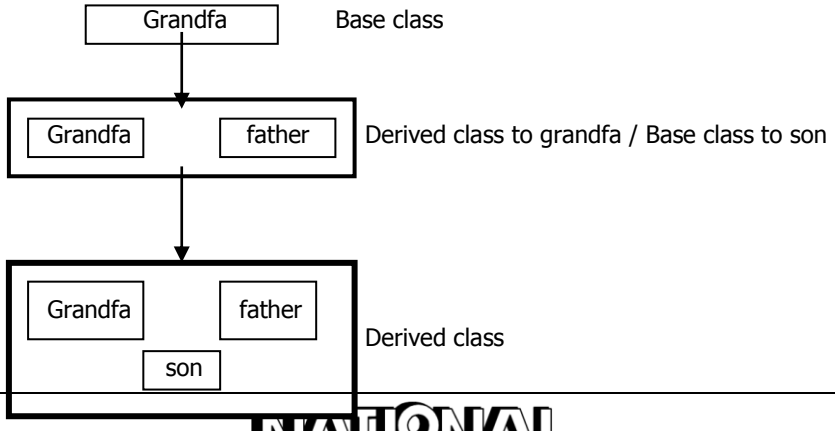



```
// Ex-24: PGM ILLUSTRATES MULTIPLE INHERITANCE
#include<iostream.h>
#include<conio.h>
#include<string.h>
class father
{
 private:
  char fname[20];
 public:
  void set1(char* name1)
  {
```

```
  strcpy(fname,name1);
 }
 void get1()
 {
  cout<<"Father name = "<<fname<<endl;
 }
};

class mother
{
 private:
  char mname[20];
 public:
  void set2(char* name2)
  {
   strcpy(mname,name2);
  }
  void get2()
  {
   cout<<"Mother name = "<<mname<<endl;
  }
};

class son : public father, public mother
{
 private:
  char sname[20];
 public:
  void set3(char* name3)
  {
   strcpy(sname,name3);
  }
  void get3()
  {
   cout<<"Son name = "<<sname<<endl;
  }
};

void main(void)
{
 son object;
 object.set1("Rathinavel");
 object.set2("Meena");
 object.set3("Kishore");
 clrscr();
 object.get1();
 object.get2();
 object.get3();
 getch();
}
```

MULTILEVEL INHERITANCE:

The properties of Base class(Grandfa) are inherited to derived class(father), which can consider as base class to another derived class(son). The following diagram shows this.



In above diagram, the base class Grandfa having some properties and derived to the derived class father. Then the properties of both Grandfa and father, derived to the class son.

So, Grandfa is a direct base class to the derived class father and indirect base class to the derived class son. Also the class father is a direct base class to the derived class son.

Therefore, the class son having the properties of Grandfa, father and it's own.

```
//   Ex-25: PGM ILLUSTRATES MULTILEVEL INHERITANCE
#include<iostream.h>
#include<conio.h>
class Grandfa
{
 protected:
  int p;
};

class father : public Grandfa
{
 private:
  int q;
 public:
  void setfather(int b)
  { q=b; }

  void getfather()
  { cout<<"father = "<<q<<endl; }
};

class son : public father
{
 private:
  int r;
 public:
  void setson(int a, int c)
  {
   p=a;
   r=c;
  }
  void getson()
  {
   cout<<"Son = "<<r<<endl<<"Grandfa = "<<p<<endl;
  }
};

void main(void)
{
 son ob;
 ob.setson(75,25);
 ob.setfather(50);
 clrscr();
 ob.getson();
 ob.getfather();
 getch();
}
```

FRIEND FUNCTION:

When using two functions for accepting data and displaying it in Base class, we would be the overhead involved in making function calls each time the input is accepted from the user and displayed. To reduce the number of unnecessary function calls would be to make the member data itself public. But this would defeat the whole purpose of encapsulation. The answer to this problem is a *friend* function. A function declared as a friend function of the class acts like just that a friend of the class. It is not a member of the class but it can access all public, protected and private members of the class. A friend function can be declared within the class declaration as follows.

```
friend void get(void);
```

Once this is done, the body of the function can be defined anywhere in the program. This function acts as a member function with respect to the class (i.e., it can access all private members) and as an ordinary function with respect to the rest of the program (i.e., it need not be referenced through an object though it is declared as part of the class).

Note:

A friend function can be either a global function or can be a member function of another class. Since a friend function is not part of the class, it can be declared anywhere in the private, protected and public section of the class without its visibility getting affected.

```
// Ex-26: A CLASS IS FRIENDSHIP TO MEMBER FUNCTION
#include<iostream.h>
#include<conio.h>
class demo
{
  private:
   int x;
  public:
   void set(int a);
   friend void get(demo ob);
                        //Here class demo is friendship to member function get()
};
void demo :: set(int a)
  { x = a; }
void get(demo ob)
  { cout<<"x = "<<ob.x<<endl; }

void main()
{
  demo obj;
  obj.set(50);
  clrscr();
  get(obj);
  getch();
}

//  Ex-27: A CLASS IS FRIENDSHIP TO ANOTHER CLASS
#include<conio.h>
#include<iostream.h>
class one
{
  friend class two;
  private:
     int x;
  public:
   void set(int a)
     { x = a; }
};
class two
{
 public:
  void get(one ob)
   { cout<<"x = "<<ob.x<<endl; }
};

void main(void)
{
 one ob1;
 two ob2;
 ob1.set(75);
 clrscr();
 ob2.get(ob1);
 getch();
}
```

POLYMORPHISM

FUNCTION SIGNATURE:

The signature of function, depends on the type, no. of parameters and it's sequence. If the type of parameter or no. of parameter or sequence of parameter differs then the signature also differ.

Examples:

```
void fun(int;
void fun(char);
```

Here, the type of parameter differ.

```
void fun(int);
void fun(int, float);
```
Here, both type and no. of parameter differ.

```
void fun(char);
void fun(char, char);
```
Here, the no. of parameter differ.

Note: The return type is not a part of function's signature. Hence, the following two declarations cannot occur in the same program.

```
void fun();
int fun();
```

FUNCTION OVERRIDING:

The derived class contains same name and same signature as that of base class function, then the derived class function is said to override the base class function.

The scope resolution (: :) operator can be used to access base class functions through an object of the derived class.

```
// Ex-28: PGM ILLUSTRATES FUNCTION OVERRIDING
#include<iostream.h>
#include<conio.h>
class base
{
 private:
  int x;
 public:
  void set(int a)
   { x=a; }
  void get()
   { cout<<" x = "<<x<<endl; }
};

class derived : public base
{
 private:
  int y;
 public:
  void set(int c)
   { y=c; }
  void get()
   { cout<<" y = "<<y<<endl; }
};
void main(void)
{
 derived obj;
 obj.base::set(20);
 obj.set(40);
 clrscr();
 obj.base::get();
 obj.get();
 getch();
}
```

FUNCTION OVERLOADING:

When more than one member function with the same name is declared in a class, the function is said to be overloaded in that class. The scope of the overloaded function is restricted to a particular class. Other classes might use the same name again, overloading it differently or even not overloading it at all. These classes have entirely separate scopes, and the compiler keeps them separated.

Overloaded functions need to differ in signature, not in function name.

```
//  Ex-29: PGM ILLUSTRATES FUNTION OVERLOADING
#include<iostream.h>
#include<conio.h>
class overlod
{
 private:
  int no;
```

```
 public:
  void show(int n)
   {  no=n;  }
  int show()
   {  return no; }
};

void main()
{
 overlod ob;
 ob.show(14);
 int i=ob.show();
 clrscr();
 cout<<i<<endl;
 getch();
}
```

Here, one function is used to read and the other one to write a value of variable. This is the common use of overloaded functions. An overloaded function need not be defined as inline.

CONSTRUCTOR OVERLOADING:

Apart from performing the special role of initialization, constructors are similar to other functions. This includes function overloading. Overloaded constructors are commonly used in C++.

```
// Ex-30: PGM ILLUSTRATES CONSTRUCTOR OVERLOADING
#include<conio.h>
#include<iostream.h>
class calculate
{
  private:
    int number1, number2, tot;
  public:
    calculate();                // Default Constructor
    calculate(int, int);        // Constructor with two arguments
    void input(int, int);
    void add();
    void disp();
};

calculate :: calculate()       // Default Constructor
 {
   number1=number2=tot=0;
 }
calculate :: calculate(int num1, int num2) // Argument Constructor Definition
 {
   number1 = num1;
   number2 = num2;
   tot = 0;
 }
void calculate :: input(int num1, int num2)
 {
   number1 = num1;
   number2 = num2;
 }

void calculate :: add()
 {
   tot = number1 + number2;
 }
void calculate :: disp()
 {
   cout<<"The sum of two number is "<<tot<<endl;
 }

void main()
{
   calculate cal1;             // Default Constructor Invoked
   calculate cal2(10,4);               // Argument Constructor Invoked
   calculate *calptr;
   calptr = new calculate(14,10);
```

```
      // Pointer is initialized with address and values assigned to member data
  cal2.add();                 // Add function invoked to initialize member data
  clrscr();
  cal2.disp();
  calptr->add();
  calptr->disp();
  delete calptr;
  getch();
}
```

Here, the object *cal1* is a default constructor is invoked because *cal1* is created without arguments. But, for the object *cal2*, and the pointer *calptr*, the two-argument constructors are invoked as they are created with two arguments.

Note:

A class can have many constructors, each differing in the number, type and order of arguments. As mentioned earlier, the number, type and order of arguments are collectively referred to as the signature of the function.

OPERATOR OVERLOADING:

Operators are overloaded to increase the scope of the operator. It is used to make abstract data types (ADTs) more natural and similar to built-in data types.

Operator overloading refers to giving additional meaning to the normal C++ operators when they are applied to ADTs. Only a predefined set of C++ operators can be overloaded. An operator can be overloaded by defining a function for it. The function for the operator is declared by using the keyword *operator*.

CLASSIFICATION OF OPERATORS:

1. Binary operators.
2. Unary operators.

OVERLOADING BINARY OPERATORS:

Binary operators are those which work on two operands. Examples of binary operators are as follows.

Arithmetic operator	:	+, -, *, /, %
Arithmetic assignment operator	:	+=, -=, *=, /=
Comparison operator	:	<, >, <=, >=, ==, !=

Syn: return_type operator +(parameters)

```
//  Ex-31: OVERLOADING BINARY OPERATORS
#include<conio.h>
#include<iostream.h>
class Load
{
  private:
   int x,y;
  public:
    Load()
     { }
    Load(int a, int b)
     {
        x=a;
        y=b;
     }
    void get()
     {
        cout<<"x = "<<x<<"  "<<"y = "<<y<<endl;
     }
    Load operator +(Load ob);
};

Load Load :: operator + (Load ob)
 {
   Load l;
   l.x = x+ob.x;
   l.y = y+ob.y;
   return l;
 }


void main(void)
{
  Load ob1(2,4),ob2(6,8),ob3;
```

```
 clrscr();
 ob1.get();
 ob2.get();
 ob3=ob1+ob2;
 ob3.get();
 getch();
}
```

OVERLOADING UNARY OPERATOR:

Unary operators are those operators that act on a single operand. Examples of unary operators are '**++**' and '**- -**'.

```
// Ex-32: OVERLOADING UNARY OPERATOR
#include<iostream.h>
#include<conio.h>
class distance
{
 int fage;
 float mage;
public:
 distance(int f, float m);
 void operator ++(void);
 void display();
};

distance :: distance(int f, float m)
{
fage = f;
mage = m;
}
void distance :: operator ++(void)
{
 fage++;
 ++mage;
}
void distance :: display()
{
 cout<<"Age of female "<<fage<<endl;
 cout<<"Age of male "<<mage<<endl;
}

void main()
{
 distance dist(18,21);
 ++dist;
 clrscr();
 dist.display();
 getch();
}
```

Prefix and Postfix:

The ++ operator, overloaded in the above program cannot be used as a postfix operator. The compiler invokes the operator function with an int argument for the postfix application of the operator. For example,

```
int operator ++(int);
```

Here, the argument int is a dummy argument, which is used only to distinguish between the prefix and the postfix operators.

x++ will be resolved as x.operator ++(0)

++x will be resolved as x.operator ++()

DYNAMIC POLYMORPHISM

INTRODUCTION:

Dynamic polymorphism includes dynamic or late binding. Virtual, Pure virtual functions and abstract classes are also discussed here. The association of an object to its corresponding class at compile time, is called *static binding.* Dynamic or Late binding means, an object can be created and dynamically during runtime based on certain conditions.

Dynamic polymorphism is nothing but late or dynamic binding. Late binding is implemented using virtual functions and base class pointers.

VIRTUAL FUNCTIONS:

To enable late binding, a function is declared as *virtual*. A virtual function is a function that is declared as virtual in a base class and redefined in the derived class. To declare a function as virtual, its declaration is predicted by the keyword *virtual*. The redefinition of the function in the derived class overrides the definition of the function in the base class. A base class pointer can be used to point to any class, derived object that contains a *virtual function*. C++ determines which version of that function to call based upon the type of object pointed to by the pointer. Thus, when different objects are pointed to, different versions of the virtual function are executed.

```
// Ex-33: VIRTUAL FUNCTION
#include<iostream.h>
#include<conio.h>
class Base
{
public:
 virtual void display()
 {
  cout<<"\n\t Base class virtual function"<<endl;
 }
};

class Derived1: public Base
{
 public:
  void display()
  {
   cout<<"\n\t Derived1 class display function"<<endl;
  }
};

class Derived2: public Base
{
 public:
  void display()
  {
   cout<<"\n\t Derived2 class display function"<<endl;
  }
};

void main()
{
 clrscr();
 Base *b;
 b=new Base;
 b->display();
 b=new Derived1;
 b->display();
 b=new Derived2;
 b->display();
 getch();
}
```

PURE VIRTUAL FUNCTIONS:

When a derived class does not redefine a virtual function, the version defined in the base class will be executed. In situations like this, a base class may not be able to define an object sufficiently to allow a base class virtual function to be created and in some other situations like all derived classes override a virtual function. To handle these two cases, C++ introduces pure virtual functions. The declaration of the pure virtual function has assignment to zero notation to indicate that there is no definition for the function. The prototype for a pure virtual function is

```
virtual return_type function_name(parameter_list) = 0;
```

When a function is declared as a pure virtual function, all derived class must override the function. If the derived class fails to override the pure virtual function, an error is encountered.

```
//  Ex-34: PURE VIRTUAL FUNCTION
#include<iostream.h>
#include<conio.h>
class number
{
 protected:
  int value;
 public:
```

```
  virtual void show()=0;     // Pure virtual function
  void setvalue(int v)
   { value = v; }
};

class hexa : public number
{
 public:
  void show()
   {
  cout<<"The given number is : "<<value<<endl;
    cout<<"The hexa number is : "<<hex<<value<<endl<<endl;
   }
};

class octal : public number
{
 public:
  void show()
  {
   cout<<"The octal number for 150 is "<<oct<<value<<endl;
  }
};

void main()
{
 octal o;
 hexa h;
 h.setvalue(100);
 clrscr();
 h.show();
 o.setvalue(150);
 o.show();
 getch();
}
```

ABSTRACT CLASSES:

A class that contains at least one pure virtual function (function declared, but not defined) is said to be an *abstract class*. No object of an abstract class can be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived class.

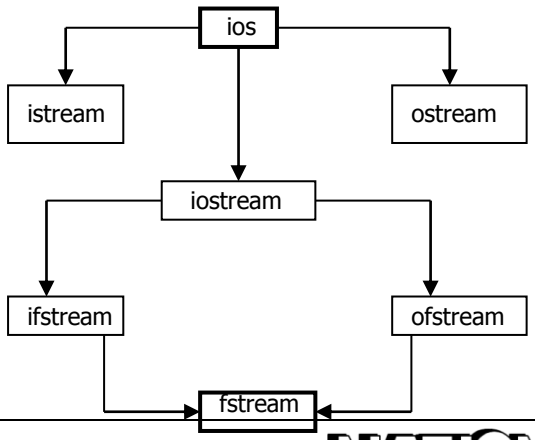
FILE HANDLING

The file loading technique of C++ support file manipulation in the form of *stream* object. The stream objects are predefined in the header file, iostream.h. There are no predefined objects for disk files. All class declaration have to be done explicitly in the program.

Each stream is associated with a particular class, which contains member functions and definitions for dealing with that particular kind of data flow. These are three classes for handling files.

ifstream	-	It is derived from istream, is used for file input.
ofstream	-	It is derived from ostream, is used for file output.
fstream	-	It is derived from iostream, is used for both file input and output.

FILE STREAM CLASS HIERARCHY:



File Input and Output:

To perform file I/O, the header file fstream.h must be included in the program. It defines several classes, including ifstream, ostream and fstream. These classes are derived from ios. So ifstream, ofstream and fstream also have full access to all operations defined by ios.

Open mode Bits:

In an ios class, the bits that are associated with the opening of a file are called open mode bits. The open mode bits state the method in which a file is to be opened. The different file opening modes are:

in	:	Opens a file for reading only.
out	:	Opens a file for writing only.
app	:	Appends at the end of an existing file without deleting the previous details.
ate	:	Seeks to the end-of-file.
trunc	:	Trunctates the file, if it exists.

Reading / Writing Data from / to a Disk File:

get()	:	It is a class member function to retrieves a string, one character at a time.
put()	:	It is a class member function to writes the string, one character at a time.

Note: File streams have open() and close() member functions to open and close a file respectively.

```
//   Ex-35: ASSIGNING VALUES INTO THE FILE AND READ THE SAME
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
 clrscr();
 ofstream out("SYSTEM");
 out<<"KEYBOARD "<<1205.50<<endl;
 out<<"MONITOR "<<3525.75<<endl;
 out<<"CPU "<<7350<<endl;
 out.close();
 ifstream in("SYSTEM");
 char item[20];
 float price;
 in>>item>>price;
 cout<<item<<" "<<price<<"\n";
 in>>item>>price;
 cout<<item<<" "<<price<<"\n";
 in>>item>>price;
 cout<<item<<" "<<price<<"\n";
 in.close();
getch();
}

//  Ex-36: PGM ILLUSTRATES TO WRITE A TEXT FILE
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
#include<stdlib.h>
void main(void)
{
 clrscr();
 ofstream out;
 out.open("demo.dat");
 if(out.fail())
  {
    cout<<"Error: File opening"<<endl;
    exit(0);
  }
  int i;
  while(1)
   {
    i=cin.get();
```

```
   if(i==-1)         //  -1 is equalent to ctrl+Z
     break;
   else
     out.put((char)i);
  }
 out.close();
}

// Ex-37: PGM ILLUSTRATES TO READ THE CONTENT OF EXISTING TEXT FILE
#include<fstream.h>
#include<conio.h>
#include<iostream.h>
#include<process.h>
void main()
{
 clrscr();
 ifstream in;
 in.open("demo.dat");
 if(in.fail())
  {
    cout<<"Error : File opening"<<endl;
    exit(0);
          /* To execute this stmt we should include the header file
                                                        'process.h / stdlib.h' */
  }
  int i;
  while(1)
  {
    i=in.get();
    if(i==-1)
      break;
    else
      cout<<(char)i;
  }
  in.close();
  getch();
}
```

----------→ **END** ←----------