"Hard work is the key to success."

Unit - 1

Introduction to Object Oriented Programming

Definition and History / Development of C++ :- C++ is the extension of C language. It can also be said as :- "C++ is superset of C language." It is a general purpose high-level language which supports Object-Oriented concepts. C++ was developed by Bjarne Stroustrup at AT&T Bell Laboratories, New Jersey, USA in early 1980s.

Since, the class was major addition to the original C language. Therefore, Bjourne Stroustrup initially called the new language, "C with classes". However, later in 1983, this name was changed to "C++".

The most important features that C++ incorporates are class, Inheritance, Polymorphism, Data Abstraction and Encapsulation, Overloading, ..., etc makes C++ a truly Object-Oriented language.

Basic concepts of OOPS :- Following are the concepts used in Object-Oriented Programming:-

i) Objects :-
Object is any real-world entity which has got its own existence. It can be data, record, file, place, person, etc.

ii) Class :-
Class is a collection of objects of similar types. Also, classes are user-defined and behave like inhibit data type in programming.

Note :- Object is an instance of class (occurance).

Example :-

Class	Object
Fruit	Mango, Grapes, Banana.
Fish	Whale, Starfish, Dolphin.
Animal	Lion, Dog, Cat.

Satyadeep 15/05/18

iii) Data Abstraction & Encapsulation :-
Encapsulation is the feature of wrapping data and function together into a single unit. Data Encapsulation

is the important feature of OOPS due to which data is not accessible to the outside world. These data and functions can only be accessed by those functions which are wrapped with it. This insulation of data from direct access by the program is called "Data Hiding" or "Information Hiding".

Abstraction means Representing important functions or properties without going into its detail explanation.

iv) Inheritance :-

Inheritance is the property which child class acquires the properties of its parent class.

Inheritance gives the concept of code reusability. It means Write Once and Use Multiple times. It improves efficiency of program, saves memory space and increases throughput of the program.

v) Polymorphism :-

Polymorphism, comes from greek word which means "Ability to take more than 1 form". Polymorphism plays an imp. part for functions and operators having different behaviour, and characteristics in different instances (situations).

The concept of polymorphism gives:-

a) Function Overloading :-
A function having same name, but different arguments can exhibit different behaviour. This is known as function Overloading.

```
for ex. :-          int sum (int, int);
                    int sum (int, int, int);
Overloaded          int sum (int, int, int, int);
functions                                    :
                                             :
```

b) Operator Overloading :-
An operator having different behaviour in different situations is known as Operator Overloading.

```
for ex. :-   5+6 = 11
             Wel + Come = Welcome
               (Concatenation)
```

[illegible] 16/05/18

vi) Dynamic Binding :- Binding is the process of linking of a procedure call to the code to be executed in response to the call. Dynamic Binding means that the code associated with the given procedure call is not known until at the runtime. It is also known as late Binding.

vii) Message Passing :- A message is a request for execution of a procedure by an object. Therefore, it invokes a function (procedure) in the recieving object which gives the desired output. Message passing means specifying the name of the object, name of the function and the information to be sent.

Advantages of OOPS :- OOPS is the most widely used programming concept. It has many benefit, and produces better quality of software with minimum maintainance cost. Some of the major benefits of OOPS are as follows :-

[illegible] 17/05/18

i) Softwares developed using OOPS concept can be easily upgraded from small systems to large systems.

ii) OOPS emphases on data and therefore it captures detail model during development stage.

iii) In OOPS, work is divided and based on objects.

iv) It easily manages software complexity and any other encountered in program.

v) It eliminates redundant codes and extend the use of existing class i.e, Code Reusability with the help of inheritance concept.

vi) OOPS allows to build program from the standard working module rather than to start writing the code from scratch. This saves development time and ensures higher productivity.

[illegible] 18/05/18

vii) OOPS allows to build safe and secure programs which can't be invaded by the code in other parts of the program through the concept of data hiding.

Date
Page

Application of OOPS :- Application of OOPS ranges in many areas. It finds use in user interface design such as Windows. The promising area of application of OOPS are :-

i) Real-Time Systems
ii) Simulation and Modeling
iii) Object-Oriented Database
iv) Hypertext Hypermedia
v) Artificial Intelligence & Expert System
vi) Neural Network & Parallel Programming
vii) Decision Support System & Office automation
viii) CAD/CAM.

Satyadeep 19/05/18

May, 2018.

Unit - 2

Fundamentals of C++ Programming

Structure of C++ Program :-

Library files / Include files
Class Declaration / Definition
Member functions & Data declaration
Main function

/* C++ Program */

```
#include<iostream.h>
  class Student
 {
    char name[30];
     int age;
   public:
        void getdata();
        void showdata();
  };
void Student :: getdata()
{
   cout << "Enter the Name";
   Cin >> name;
```

```
    cout << "Enter Age : ";
    cin >> age;
}
void Student :: Showdata()
{
    cout << "Name : " << name;
    cout << "Age : " << age;
}
void main()
{
    Student S;
    S.getdata();
    S.Showdata();
}
```

[illegible] 22/05/18

Syntax of class

```
        class <class_name>
        {
          private:        // Visibility modifiers / labels.
            data-type members;
            data-type members-functions;
            - - - - - - - - -

          protected:
            data-type members;
            - - - - - - - - -
            data-type member-functions;
            - - - - - - - - -
            - - - - - - - - -
```

```
public:
        data_type members;
        ---------------
        ---------------
        data_type functions;
        ---------------
};

class <obj1, obj2, obj3, ...>
```

A class has 3 types of visibility modifiers/labels. They are :

i) Private
ii) Protected
iii) Public

i) private :-
Those variables and functions which are declared in private section can only be accessed by the member functions and friend function of this class. The member functions and friend function of this class can always read/write private data members. The private data members are not accessible outside of the class.

Note :- By default, if no visibility modifier is specified, then the data members and member functions of the class are private.

ii) protected :-

The member functions which are declared in protected section can only be accessed by the member functions and friend functions of that class. Also, protected data and functions can be accessed by member functions and friend functions derived from this class. Beyond that protected data are not accessible.

iii) public :-

The members functions which are declared in public section can be accessed by any function out of the class.

Array of Class Objects :-

An array is a user-defined datatype whose members are homogenous and stored in continous memory locations. in practical application, we use array for designing large size of data.

Satyadeep
26/05/18

* W.A.P to read Student particulars such as Roll No., Name, Age, Height, Weight from the user and display its contents on the screen. The class Student is defined as an array of class object.

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

const int Max = 20;
class Student
{
  private:
     char name [30];
     int roll;
     int age;
     int height;
     int weight;
  public:
    inline void getdata()
    {
    cout<<"Enter Name, Roll, Age, Height, Weight";

    cin>>name>>roll>>age>>height>>weight;
    }

  inline void showdata()
  {
    cout<<"Name"<<name;
```

```
        cout << "Roll" << roll;
        cout << "Age " << age;
        cout << "Height " << height;
        cout << "Weight " << weight;
    }
 };
  void  main()
 {
      Student  obj[Max];
        int  i, n;
      cout << "How many Students ";
         cin>>n;
      cout << "Enter following Information";

      for (i=0; i<n; i++)
      {
          obj[i].getdata();
      }
       Cout << "Contents of class";
       for (i=0; i<n; i++)
       {
           obj[i].showdata();
       }
          getch();
        }
```

* W.A.P to calculate simple arithmetic operations such as addition, multiplication, Substraction, division. The input should be user-defined and display its output.

```
#include<iostream.h>
class calculator{
int a, b, c;
public:
  void inp_choice;
  void addition;
  void multiplication;
  void substraction;
  void division;
};

void inp_choice()
{
 cout<< "Enter the numbers";
  cin>>a;
  cin>>b;
cout<<"Enter 1 for addition \n Enter 2
for Multiplication \n Enter 3 for Substraction
\n Enter 4 for division";
cout<<"Enter your choice:";
  int Choice;
  cin>>Choice;
  Switch(choice)
 {
  case 1: addition();
```

```
            break;
    case 2 : multiplication();
                break;
    case 3 : Substraction();
                break;
    case 4 : division();
                break;
    default : "Wrong choice";
 }
  void addition()
  {
      c= a+b;
      cout << "Sum is " << c; }

  void Substraction()
  {
      c= a-b;
      cout<< "Difference is " <<c; }

  void multiplication()
  {
      c= a*b;
      cout << "Product is " <<c; }

  void division()
  {
      c= a/b;
      cout << "Result is " << c; }
```

Date
Page

```
void main()
{
    calculator calc;
    calc.int_choice();
}
```

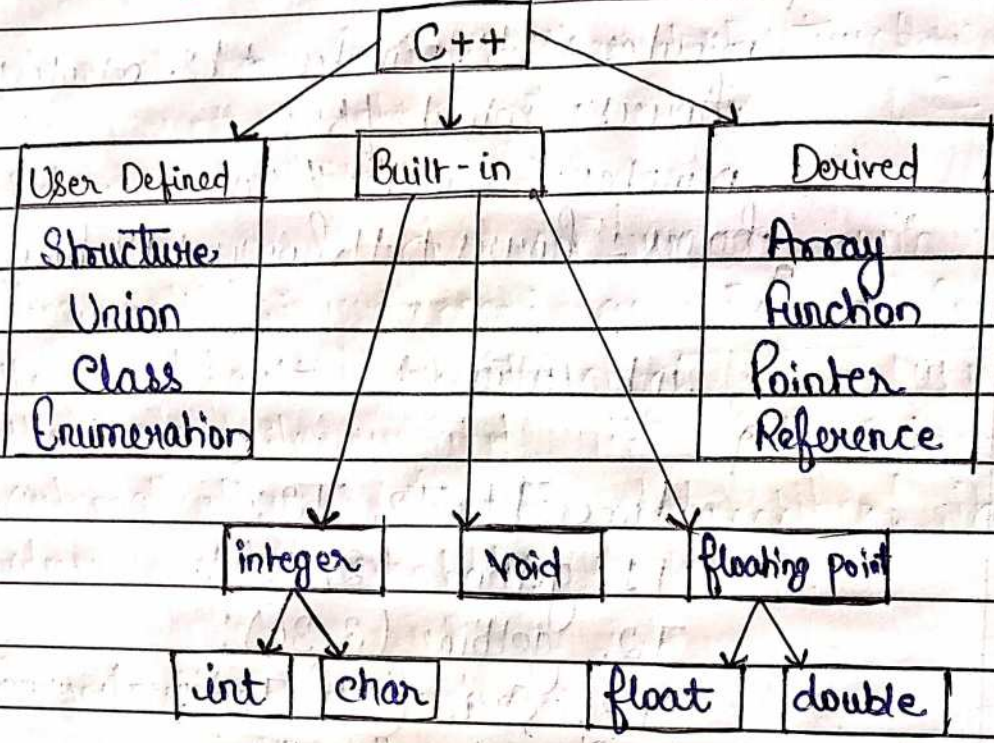
Objects as Function Arguments

```
#include <iostream.h>

class time
{
    int hours;
    int minutes;
    public:
      void gettime (int h, int m)
      {
        hours = h;
        minutes = m;
      }
      void puttime (void)
      {
        cout << "Hours : " << hours;
        cout << " Minutes :  " << minutes;
      }
      void sum (time, time);
};
```

```
void time :: Sum (time t1, time t2)
{
  minutes = t1. minutes + t2. minutes;
  hours = minutes/60;
  minutes = minutes % 60;
  hours = hours + t1. hours + t2. hours;
}
int main ()
{
  time T1, T2, T3;
  T1. gettime (2, 45);
  T2. gettime (3, 30);
  T3. Sum (T1, T2);
  cout << "T1 = ";
  T1. puttime ();
  cout << "T2 = ";
  T2. puttime ();
  cout << "T3 = ";
  T3. puttime ();
  return (0);
}
```

Data Types



Operators in C++

All valid operators of C are valid operators of C++ with same meaning and operations.

i)	::	scope resolution operator
ii)	::*	pointer to member declaration
iii)	→*	access a member using a pointer to object & pointer that to member
iv)	.*	access a member using a pointer to object name & pointer that to member
v)	delete	memory release operator
vi)	endl	line feed operator
vii)	new	memory allocation operator
viii)	setw	field with operator.

Unit-4

Overloading

Function Overloading:

When 2 or more functions having same name but they perform different operations, such a function is known as function overloading. It happens because the internal implementation of every function differs from one another.

Operator Overloading:

When a given operator exhibits different behaviour other than in normal behaviour, then, such an operator is said to be overloaded operator and this is known as Operator Overloading.

Note:- Following are the operators which cannot be overloaded:-

i) Class member access operator (., .*)
ii) Scope resolution operator (::)
iii) Sizeof operator (sizeof)
iv) Conditional operator (?:)

Unit - 3

Constructors and Destructors

Default Constructor

A constructor with no argument is known as Default Constructor. The default constructor is supplied by the compiler and it gets invoked automatically whenever object of a class is created.

Parameterised Constructor

The constructor which accepts argument.

Copy Constructor

Copy constructors are those constructor which accept object of class as argument. A copy constructor is also used to declare and initialise an object from another object.

Destructor:

A destructor is a special member function which is having same name as class name, but it starts with Tilde (~) symbol. A destructor doesn't have any return type, and it doesn't take any parameter. It is called automatically called by the compiler. Its purpose is to release the memory occupied by objects.

Unit-5

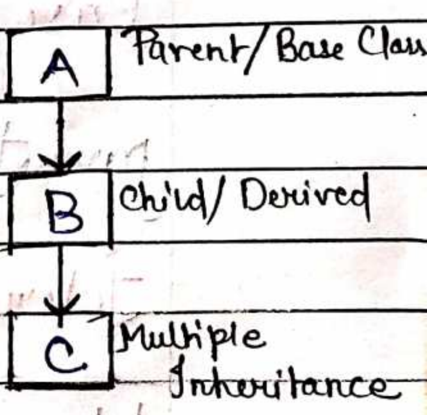
Inheritance

Inheritance is the mechanism through which child class (new class) acquires the property of its parent class.

There are different ways through which child class acquires the property of its base class. Based on this, there are different types of Inheritance :-

i) Single Inheritance :-
When a derived class is having only one parent class, then this type of inheritance is known as Single Inheritance.

ii) Multi-level Inheritance :-
When a child class is derived from another derived class, then it is known as Multilevel Inheritance.



iii) Multiple Inheritance :

When a child class is having more than ~~op~~ one parent class, then it is known as Multiple Inheritance.

$$\begin{tikzcd}
A \arrow[dr] & \cdots \arrow[d] & n \arrow[dl] \\
 & C &
\end{tikzcd}$$

iv) Hierarichal Inheritance :

When a parent class is having more than one derived class, then it is known as Hierarichal Inheritance.

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr, dotted] & & \\
B & C & D & \cdots
\end{tikzcd}$$

Derived Class

Those classes which are having its base/parent/root class is said to be derived class.

$$\begin{tikzcd}
\boxed{A} \arrow[d] & \text{Root / Parent / Base Class} \\
\boxed{B} & \text{Derived / Child class}
\end{tikzcd}$$

Base Class

Those classes which are having no parent is said to base/parent/root class.

Intermediate Class

Those nodes/class which lies in b/w parent class and child class are said to be intermediary class.

$$\begin{tikzcd}
\boxed{A} \arrow[d] & \text{Root class} \\
\boxed{B} \arrow[d] & \text{Intermediate class} \\
\boxed{C} & \text{Child class}
\end{tikzcd}$$

// Program to illustrate the use of endl and setw

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

        int main()
       {
        int BS= 2356000;
        int Allowance= 5320000;
        int Total =2888 0000;

cout<< setw(10)<<"Basic Salary= "<< setw(10)<<BS <<endl;
cout<< setw (10)<<"Allowance= "<<setw(10)<< Allowance << endl;
cout<<setw(10)<<"Total= "<<setw(10)<< Total<<endl;
 getch();
  return 0;
}
```

// Program to show Parameterised Constructor

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

              class integer {
              int m, n;
              public:
               inline integer (int, int);
               void display (void) {
```

```
cout << "m = " << ~~end~~ m << endl;
cout << "n = " << n << endl;
}};

integer :: integer (int x, int y) { //Parameterised
                                    // Constructor
    m = x;
    n = y;
}

int main () {
   clrscr();
   integer int1 (0, 100);
   integer int2 = integer (25, 75);
   cout << " Object1 " << endl;
    int1. display ();

   cout << " Object2 " << endl;
    int2. display ();
     getch();
      return(0);
}
```

// Program to Show Copy Constructor

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

class code {
   private:
    int id;
   public:
```

```
code() {} //Default Constructor
code (int a) { //Parameterized Constructor
  id = a;
}
  //Copy Constructor
code (code &x) {
  id = x.id;
}
  void display(void) {
  cout << id;
}
 };
  int main() {
   clrscr();
   code A(100);
   code B(A);
   Code C = A;
   Code D;
    D = A;
cout << "\n ID of A: ";
   A.display();
cout << "\n ID of B: ";
    B.display();
cout << "\n ID of C: ";
     C.display();
cout << "\n ID of D: ";
     D.display();
  getch();
   return(0);
  }
```

//Program to show action of Destructor

```
#include <iostream.h>
#include <conio.h>
       count
int ~~cout~~ = 0;
class alpha
{
  public:
     alpha()
     {
        count ++;
        cout<<"\n No. of Objects created:"<<count;
// count <<"\n";
     }
   ~alpha()
   {
    cout<<"No. of objects destroyed:"<<count;
    count --;
   }
  };
int main()
{
  cout<<"\n\n Enter main \n";
{ alpha A1, A2, A3, A4;
  cout<<"\n\n Enter block 1 \n";
```

```
    { alpha A6;
    }
    cout << "\n\n Re-enter Main \n";
    getch();
    return (0);
}
```

Limitations of Constructor and Destructors

- Constructors are executed automatically by the compiler whenever an object of a class is created. It initialises the class variables with default value.
- Constructor is a special member function but it doesn't have a return value.
- Constructor can have only those names in which class it belongs.
- Destructors are also special member functions of a class, which is having same name as class name and it is preceded by a tilde (~) symbol.
- Destructors are also executed automatically by the compiler whenever an object is created to destroy those objects.
- Destructor perform the task of memory

cleaning by removing unused objects.

// Function Overloading

```
#include <iostream.h>
#include <conio.h>
class funcOverload
{
 int i, j, k, tot;
 public:
  void sum(int, int)
  {
   cout << "Enter 2 numbers : ";
   cin >> i >> j;
   tot = (i+j);
   cout << " Total = " << tot << endl;
  }
  void sum (int, int, int)
  {
   cout << "Enter three numbers";
   cin >> i >> j >> k;
   tot = (i+j+k);
   cout << " Total = " << tot;
  }
 };
 void main()
 {
  funcOverload f;
  f.sum(5, 10);
f.sum(5, 10, 15);
getch();
}
```

//Operator Overloading

```
#include<iostream.h>
#include<conio.h>
class space
{
  int x, y, z;
  public:
  void getdata(int a, int b, int c);
  void display(void);
  void operator -();
};

void space :: getdata(int a, int b, int c)
{
  x=a;
  y=b;
  z=c;
}
void space :: display(void)
{
   cout<<x<<" ";
   cout<< y<<" ";
   cout<< z<<" ";
}
```

```
void space :: operator - (void)
{
    x = -x;
    y = -y;
    z = -z;
}
int main()
{
    Space S;
    S. getdata (10, -20, 30);
    cout << "S : ";
    S. display ();

    -S; //activates operator - () function

    cout << "S : ";
    S. display ();
    getch();
    return (0);
}
```

Unit-6 Pointers and Virtual Functions

Pointer

A pointer is a programming language object that stores the memory address of another value located in computer memory. A pointer references a location in memory, and obtaining the value stored at that location is known as dereferencing the pointer.

Memory Management using new and delete operators

i) New operator : The new operator is used to create memory for an object of a class. The new keyboard class the function operator new() to obtain storage. The new operator can be used to create object of any data type. Its general form is :-

Pointer variable = New data type

Example :-

```
p = new int;
q = new float;

int *p = new int;
float *q = new float;
```

iii) Delete Operator : When object is no longer required created by new operator is destroyed using delete operators. Its general form is —

Delete Pointer Variable

Example :-

```
delete p;
delete q;
```

Q. W.A.P to create dynamic memory allocation for standard data types int, float, char and double. The pointer variables are to be initiated with some data and contains of the pointer are displayed on the screen.

```
#include<iostream.h>
#include< conio.h>
void main()
{
int * ptr_i = new int (25);
float * ptr_f = new float(-10.1234);
char * ptr_c = new char ('a');
double * ptr_d = new double (1234.5678);

Cout << "\nContents of Pointer" <<endl;
Cout <<"\n Integer : " << *ptr_i << endl;
Cout << "\n Float : " << *ptr_f << endl;
cout << "\n Char : " << *ptr_c << endl;
cout << "\n Double : " << *ptr_d << endl;

delete ptr_i;
delete ptr_c;
delete ptr_f;
delete ptr_d;
getch();
}
```

Virtual functions and Polymorphism

Polymorphism is the property by which objects belonging to different classes are able to respond to the same message but in different forms. It means that polymorphism is the ability to refer to objects without any regard to their classes. Therefore, it necessiates the use of single pointer variable to refer to the objects of different classes. Here, the pointer to base class is used to refer to all the derived objects. However, the pointer to base class. This ambiguity in polymorphism is overcome with the

When a function is made virtual, the compiler determines which function to use at runtime based on the type of object ~~oriented~~ pointed by the base pointer, rather than the type of pointer, therefore by making base pointer to point to different objects we can execute different versions of virtual functions.

/* Virtual Function */

```
#include<iostream.h>
#include<conio.h>
class Base
{
 public:
  void display()
 {
 cout<<"\n Display Base";
 }
  virtual void show()
 {
 cout<<"\n Show Base ";
 }
 };
  class Derived : Public Base
 {
  public :
   void display()
  {
```

```
{ cout << "\n Display Derived";
}
  void show()
{
  cout << "Show Derived" << endl;
}};
  int main()
{
  Base B;
  Derived D;
  Base *bptr;
  cout << "\n bptr Points to Base \n";
  bptr = &B;

  bptr -> display(); //calls Base version
  bptr -> show(); // calls Base version

 cout << "\n\n bptr points to Derived \n";
  bptr = &D;

  bptr -> display();
  bptr -> show();
   getch();
    return(0);
}
```

Friend Functions

Friend function is a special function which is non-member function of a class. It has special privilage to access private and protected data member of a class. Friend function is declared as friend within a class preceded by the keyword "friend", i.e, only Prototype declaration is mentioned inside the class. The actual implementation is somewhere else in the program. We can make any function as friend function which we want to access class private and protected data members. We can declare any number of functions as friend functions which depends on our requirement in the program.

```
/* Friend function */

#include <iostream.h>
#include <conio.h>
  class sample
{
   private :
     int a;
     int b;
  public :
      void setvalue()
    {
```

```
a= 25;
b= 40;
}
friend float mean(Sample S);
};

float mean(Sample S)
{
return float(S.a+S.b)/2.0;
}
int main()
{
Sample x;
x.Setvalue();
cout<<"Mean value = "<<mean(x)<<endl;
getch();
  return(1);
}
```

Static Function

A static function is a class member function. It is declared using the keyword "static". A static function can access to only static members (functions or variables) declared in the same class. A static function can be called using its class name.

```
/* Static Function */
#include<iostream.h>
```

```
#include<conio.h>
class test
{
private:
    int code;
static int count;
public:
   void setcode (void)
{
   code = ++count;
}
void showcode (void)
{
cout<<"Object Number = "<<code<<endl;
}
static void showcount()
{
cout<<"Count = "<<count<<endl;
}
};
   int test :: count;
   int main()
{
   test t1, t2;
   t1.setcode();
   t2. setcode ();

test :: showcode ();

test t3;
t3. setcode();

test:: showcount();
 t1. showcode();
 t2. showcode();
 t3. showcode();
 getch();
  return(5);
}
```

Inline Function

Inline functions are small or single line function in that case we declare these functions as inline function. A function which is declared as inline is preceded by the keyword "inline" and it is a class member function. The objective of class member function is to improve speed of the program. As the function grows larger, the benefit of inline function is lost.

```
/* Inline Function */

#include <iostream.h>
#include <conio.h>
class calc
{
  private :
    int x;
public:
  Inline int sqrt (int a = 5)
  {
    return (a*a);
  }
  Inline int Cube (int a = 5)
  {
    return (a*a*a);
  }};
  int main()
  {
    Calc C;
    cout << "Square root = " << c.sqrt() << endl;
    cout << "Cube = " << c.cube () << endl;
    getch();
    return (0);
  }
```

Macros

Macros are symbolic constraints, it is created by using #define preprocessor directive. #define instructions define value for a symbolic constant for use in a program whenever, a symbolic name is encounted in the program, the compiler substitutes the value associated with the name automatically. #define is a preprocessor compiler directive. Therefore #define lines should not end with a semi-colon. These Symbolic constants are generally written in uppercase letter.

/* Macro */

```
#include<stdio.h>
#include<conio.h>
#define UPPER 150

int main()
{
  clrscr();
 for(long int i=1; i<=UPPER ; i++)
 {
  printf("%ld", i);
  printf("\n");
 }
  getch();
   return(0);
}

  #include<stdio.h>
  #include<conio.h>
  #define PI 3.1415

  int main()
 {
  float a, r;
   clrscr();
printf ("Input the Radius : ");
 scanf ("%f", &r);
  a = (PI * r * r);
printf ("Area of a Circle : %f", a);
 getch();
  return (0);
}
```

/* Single Inheritance */

```
#include<iostream.h>
#include<conio.h>
 class B
{
  int a;
  public:
    int b;
   void get_ab();
   int get_a(void);
   void show_a(void);
};
 class D: public B
{
  int c;
  public:
    void mul(void);
    void display(void);
};
  void B:: get_ab(void)
{
    a= 5;
    b= 10;
}
int B:: get_a()
{
    return a;
}
void B:: show_a()
{
    cout<<"a= "<<a <<endl;
}
void D:: mul()
{
    c = b* get_a();
}
void D:: display()
{
    cout<<"a= "<<get_a()<<endl;
    cout<<"b= "<<~~get~~ b << endl;
    cout<<"c= "<<c << endl;
}

int main()
{
    D d;
    d. get_ab();
    d. mul();
    d. show_a();
    d. display();
    d. b = 20;
```

```
d.mul();
 d.display ();
 getch();
  return (0);
}
```

/* Multilevel Inheritance */

```
#include<iostream.h>
#include<conio.h>
 class Student
{
 protected:
  int roll-number;
 public:
  void  get-number (int);
  void   put-number ( void );
 };
 void Student :: get-number (int a)
{
 roll-number = a;
}
 void Student :: put-number ()
{
 Cout << "Roll No. :" << roll-number <<endl;
}
 class test : public student
{
 protected:
  float sub1;
float sub2;
 public:
 void get_marks(float, float);
 void put_marks(void);
};
void test:: get_marks(float x, float y)
{
 sub1 = x;
 sub2 = y;
}
void test:: put_marks()
{
 cout<<"Marks in sub1 : "<<sub1<<endl;
 cout<<"Marks in sub2 : "<<sub2<<endl;
}
class result :: display(void)
{
 total = sub1+ sub2;
  put_number();
  put_marks();
 cout<<"\n Total: "<<total<<endl;
}
int main()
{
  result student1;
  student1. get_number(111);
  student1. get_marks(75.0, 59.5);
  student1. display();
  getch();
   return (0); }
```

/* Multiple Inheritance */

```
#include <iostream.h>
#include <conio.h>
class M
{
  protected:
    int m;
  public:
    void get_m(int);
};
class N
{
  protected:
    int n;
  public:
    void get_n(int);
};
class P : public M, public N
{
  public:
      void display(void);
};
void M :: get_m(int x)
{
  m=x;
}
void N :: get_n(int y)
{
  n=y; }
```

```
void P:: display(void)
{
    cout<< "M=  "<<m<< endl;
    cout<< "N=  "<<n << endl;
    cout<< "M*N :  "<<m*n << endl;
}
 int main()
{
   P p;
   p.get_m(10);
   p.get_n(20);
   p. display();

   getch();
    return(0);
}

/* Hierarichal Inheritance */

#include<iostream.h>
#include< conio.h>
  class student
 {
   protected:
     int roll_number;
   public:
     void get_number(int a);
     {
       roll_number = a;
     }
    void put-number (void)
  {
      cout << "Roll No: " << roll-number;
  }};
  class test : public student
  {
     protected:
           float part1, part2;
     public:
       void get-marks (float x, float y)
     {
       part1 = x; part2 = y;
     }
      void put-marks(void)
   {
     cout << "Marks Obtained: " << endl;
     Cout << " Part1: " << part1 << endl;
     cout << "Part2 : " << part 2 << endl;
  }};
     class result : public Student
     {
       float total ;
       public :
         void display(void)
       {
          get-number ();
             put-number ();
     };
       int main()
     {
```

Page

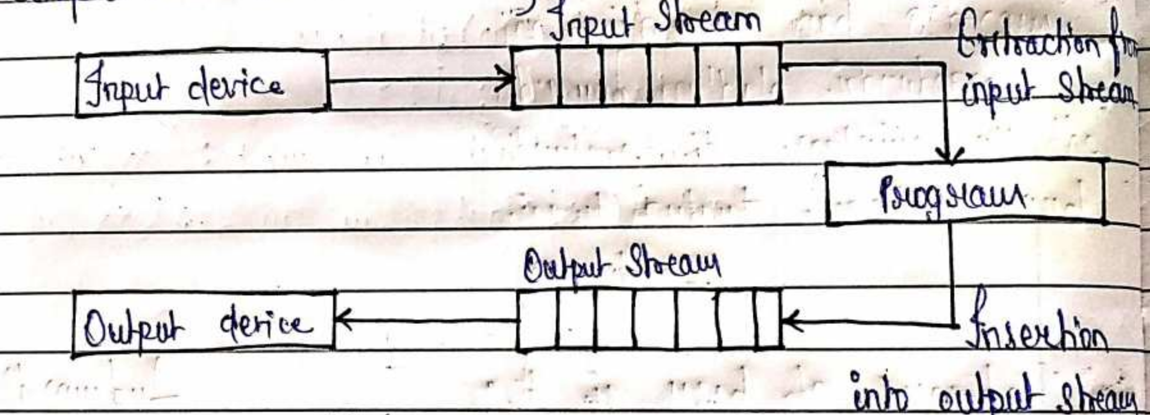
```
test t;
t. get-number();
t. put-number();
t. get-marks();

result r;
r. display();
getch();
 return(0);
}
```

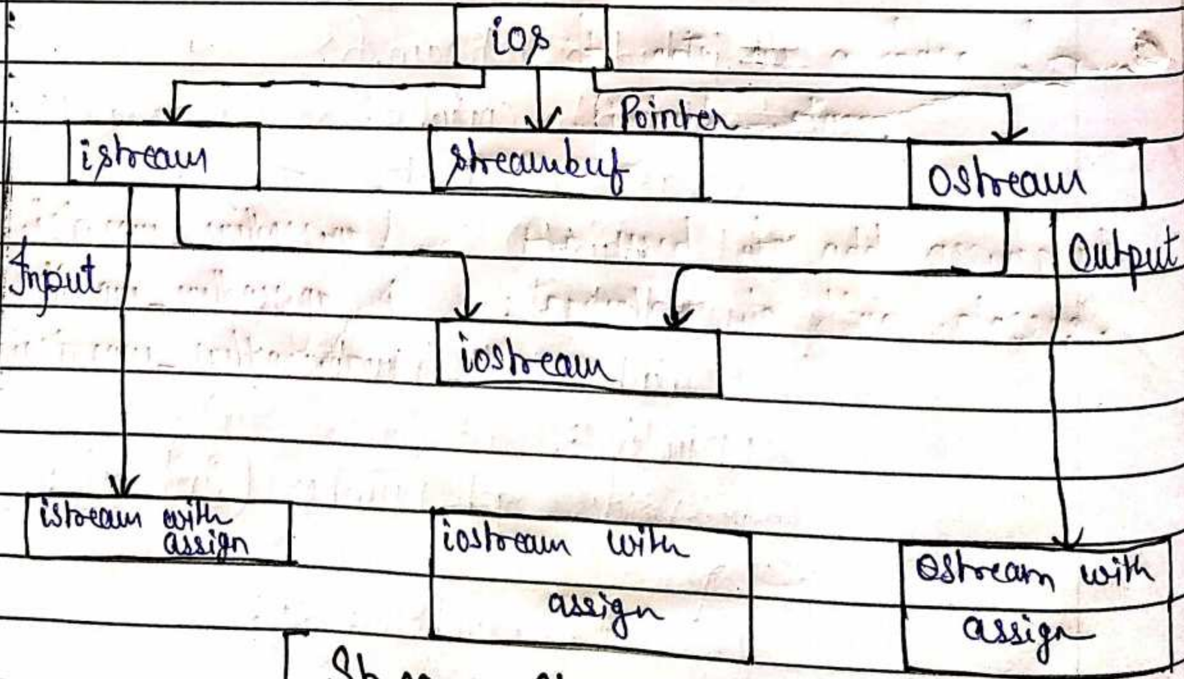
UNIT - 7 Streams

Stream / Stream Class - Hierarchy

A Stream is a sequence of bytes. It can be input stream or, output stream. The input stream accepts data from the source. Most commonly keyboard whereas the output stream is obtained from the destination. It is mostly computer screen or, printer.



[Fig: Data Streams]



[Stream Class - Hierarchy]

Figure shows Stream class - Hierarchy used for Input and Output operations. These classes are declared in the header file iostream. This file should included in all the programs that communicate with console unit.

Class Name	Description
i) ios (General Input/Output Stream Class)	• It contains basic functions and facilities that are used by all other I/O classes.
	• It is used to declare constants and functions that are necessary for handling formatted I/O operation.
	• It contains pointer to stream buffer object.
ii) istream (input stream)	• It inherits the properties of ios class.
	• It contains functions such as get(), getline(), read(), ..., etc.
	• It contains overloaded extraction (>>) operators

iii)	ostream (Output Stream)	• It inherits the properties of ios class. • It contains the output functions such as put() and write(), ...etc • It contains overloaded insertion (<<) operators.
iv)	iostream	• It inherits the properties of ios, istream and ostream through multiple inheritance. Therefore, it contains all the input/output functions and operations.
v)	streambuf	• It is used to provide interface to the physical devices. • It acts as a base for filebuf class.
vi)	istream_withassign ostream_withassign iostream_withassign	Are used to add assignment operators to these classes.