Name :- Sabita Dangi
Level :- Bachelor 2nd year, "3rd semester"
Course :- The Object Oriented Programming
Course no:- CSC-202

Unit: 1.1 :- Introduction to Programming Concept

Computer le natural language bujhdaina so there is a need of language to write program.

Programming Languages:-

A programming language is a tool used by a programmer to give the computer very specific instructions in order to serve some purpose for the user.

Examples: C, C++, java, basic, php, C#

Classification of programming languages:-

1. Unstructured programming
2. Procedural programming
3. Modular programming
4. Object-oriented programming

1. Unstructured programming:-

Usually people start learning programming by writing small and simple problems programs consisting only of one main program. Here, main program stands for a sequence of commands or statements which modify data which is global throughout the whole program.

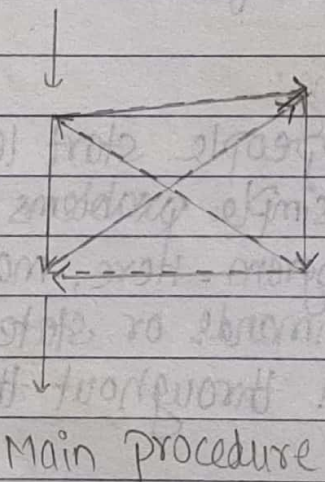
Disadvantages:

- Programming is difficult if the system for development is large.

- Copying of the same code in different part is required because there is no procedure.
- Programming takes time. So, cost of development is high.
- Mistakes on global data are difficult to trace.

2. Procedural Programming :-

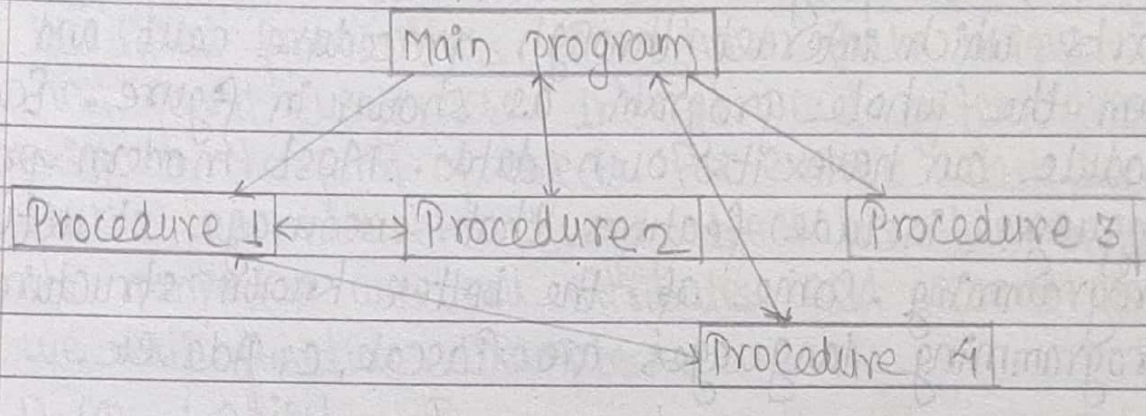
With procedural programming we are able to combine recurring sequences of statements into ~~one~~ one single place. A procedure call is used to invoke the procedure. After the sequence is processed, flow of control proceeds right after the position where the call was made.



Features:

- Emphasis is on algorithms
- Large program are divided into smaller programs known as functions or procedures

- Most of the functions share global data.
- Data move openly around the system from function to function.

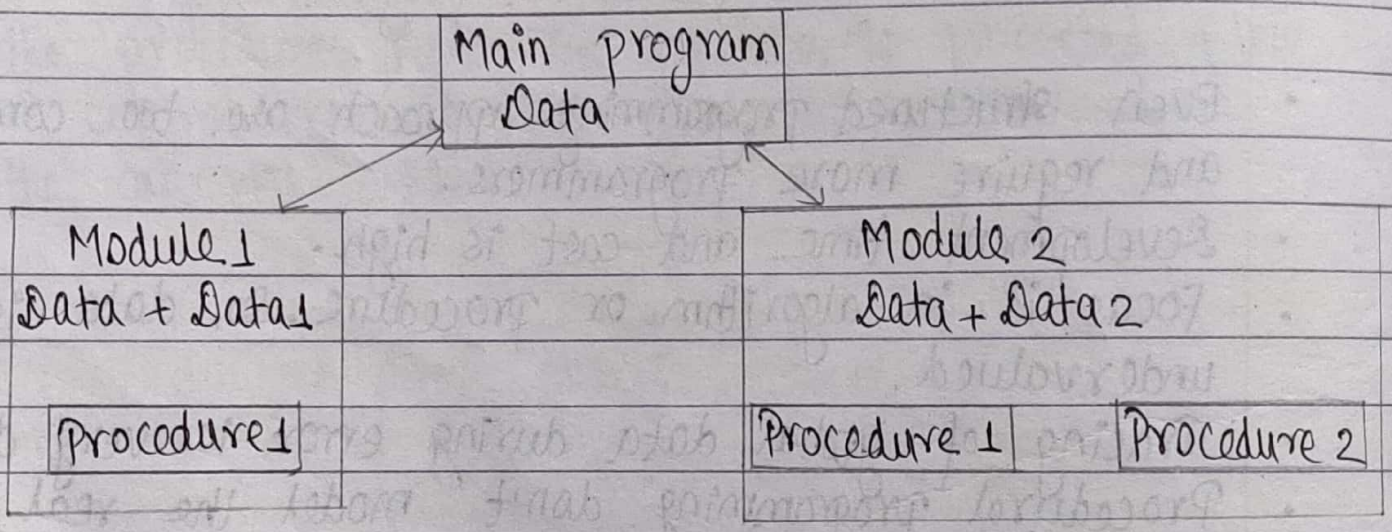


Some issues with procedural programming:

- Even structured programming approach are too complex and require more programmers.
- Development time and cost is high.
- Focus is in algorithm or procedure. So, data is undervalued.
- Tracing of global data during error is very difficult.
- Procedural programming don't model the real world very well.
- It is difficult to create new data type with procedural languages.

3. Modular Programming (structured Programming) :-

With modular programming procedures of a common functionality are grouped together into separate modules. A program is now divided into several smaller parts which interact through procedure calls and which form the whole program as shown in figure. Each module can have its own data. Most modern procedural languages include features that encourage structured programming. Some of the better known structured programming languages are Pascal, C, Ada, etc.



A structured program proceeds from the original problem at the top level down to its detailed sub-problems at the bottom level. This is called a top-down approach.

4. Object - Oriented Programming :-

Object - oriented Programming was started to overcome the limitations of earlier programming approaches. Objects are the entities that can be uniquely identified from others. They have their unique identity and found everywhere. In real world system, everything exists in the form of objects. For example: desk, bench, student, teacher, car, tree, etc. Every object has two things ; first its properties we call attributes and second its behaviour we call function. Example :-

i) Car : object

Attributes : colour, no_of_seats, chasis_number, engine_number, manufacturer, etc.

Behaviour : Move, stop, accelerate, turn, etc.

ii) Dog : object

Attributes (properties) : color, breed, size, height, name, weight, etc.

(Funct°) Behaviour : bark, bite, eat, run, chase, etc.

iii) Student : object

(properties) Attributes : color, height, weight, size, age, name, address, etc.

(function) Behaviour : read, write, learn, eat, play, talk, etc.

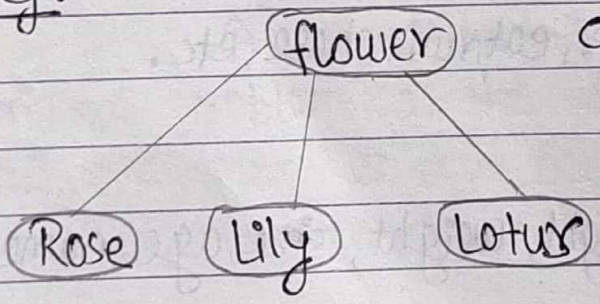
Striking features of Object-Oriented Programming (OOP):-

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Functions that operate on the data of an object are tied together in data-structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through functions.
- New data and functions can be added easily whenever necessary.
- Follow bottom of approach in programming development.

Characteristics / Features of OOP :-

1. **Object:** An object is a real world entity that has certain attributes (character) and behaviour (methods or operations). In OOP, an object is defined as an instance of a class.

Eg:-



Class is blueprint of object from which objects are created.

2. Class: A class is a group of objects that has similar attributes and behaviour. For example, we can say that the bird is a class and parrot, Pigeon & crow are the objects of that class.

Student (class)	
Data:	attributes
Name	
DOB	
Marks	
Functions:	Functions (operations)
total()	
average()	
display()	

Account
Data:
Name
Address
Tel. No
Acc. No
Functions:
deposit()
withdraw()
enquire

Car
Data:
Name
Color
Chasis no.
Function:
move()
stop()
turn()
accelerate()

3. Encapsulation:

Encapsulation is the process of combining data and functions together into a single unit called class. The data inside the class is accessible only by the member function. No function outside the class can access the data inside the class. This ensures the data security.

4. Data hiding:

The data and function can be hidden from the outside world by making them private members of the class. This ensures the data security and integrity because private members are accessible only to the class within which they are defined.

5. Inheritance:

It is the process of deriving a new class from an existing class. The class whose properties are inherited is called the base class or super class and the class that inherits this properties is called the derived class or sub-class. The sub-class can have additional properties of its own.

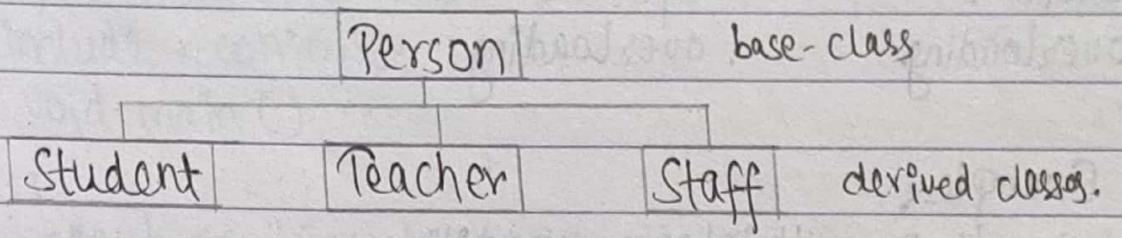


Fig: inheritance

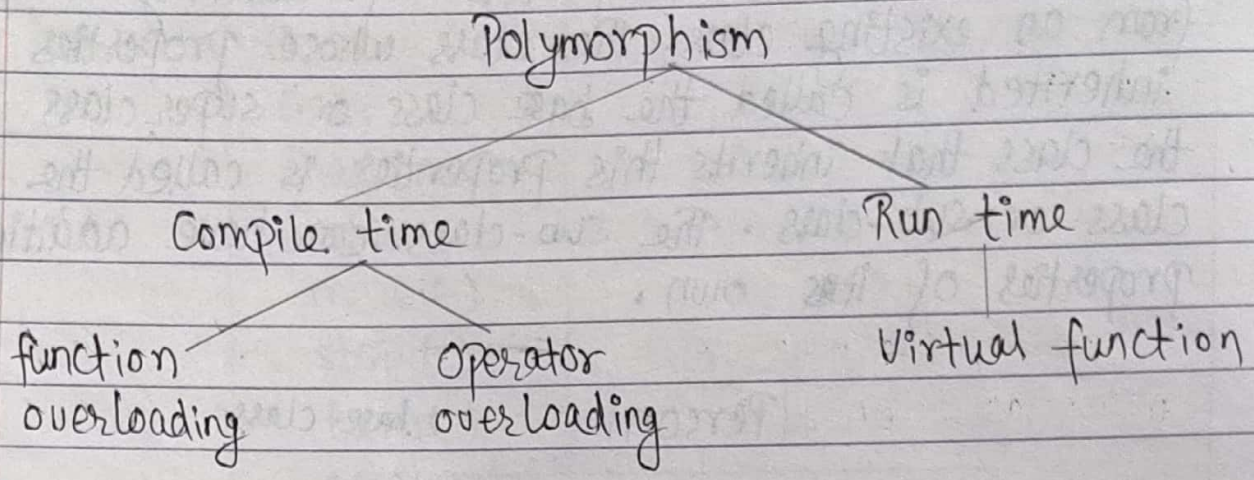
6. Reusability:

Reusability is reuse of structure without changing the existing one but adding new features or characteristics to it. It is one of the major advantage of inheritance. Reusability helps in reducing the size of application and saving the time.

7. Polymorphism:

The word polymorphism literally means having many forms. Polymorphism is a concept by which a function or an operator can be programmed to act in more than one form. In C++, there are two types of polymorphism; they are - compile time Polymorphism

and Run-Time Polymorphism.



Example:
i) 4+5 // integer addition
ii) 3.4 + 1.52 // float addition
iii) S1 + "bar" // string concatenation

8. Abstraction:

The abstraction is the process of getting detailed information according to the level of deep sight to the problem. If we want to get more information about the topic, we should go deeper to the problem.

Illustration: Map of the world in Atlas show few details of mountains, oceans, and large features on the earth. If we see map of country, it might include main city, rivers, villages, town, road, etc. If we see the map of a town, we find more details.

The highest level of abstraction can be viewed

as a community of associated objects. If we need more detailed information about the problem, we should give detailed study of interaction between objects.

First program:

```
# include <iostream.h>
# include <conio.h>
   void main()
   {
    cout << "C++ is easier than C";
    getch();
   }
```

→ Function can also exist independently of classes
eg. main()

→ cout tells the computer to display the quoted phrase.
" C++ is easier than C"

→ cout uses insertion operator <<.

→ #include <iostream.h> is an instruction to the compiler
A part of the compiler called the preprocessor deals with these directives before it begins the real compilatio

Comment:-

Comment is the written text which can be hint to the programmer or other in future. It is not executed by the compiler.

Types of comment:

1. Single line comment: // order commenting style
2. Multi line comment: /* this is preety good commenting style */.

Tokens:-

The smallest individual units in a Program are known as tokens. Some tokens are:-

- keywoords
- Identifiers
- Constants
- Strings
- Operators

Key words:-

They are explicitly reserved identifiers and cannot be used as names for the variables and user-defined elements.

Example:
- if
- char
- do
- int
- struct
- union
- try, etc.

Identifiers :-

It refers to the name of variables, functions, arrays, classes, etc. created by programmer.

Rules :

- Only alphabets, digits and underscore are permitted.
- Name cannot start with a digit.
- Uppercase and lowercase are distinct.
- keyword cannot be used as identifier.

Constants :-

Constants refers to fixed values that do not change during execution of a program.

Example :- 123
- 12.34
- "C++"
- 'c'

Data types :-

Both C and C++ compilers support all the built-in basic or fundamental datatype.

P.T.O

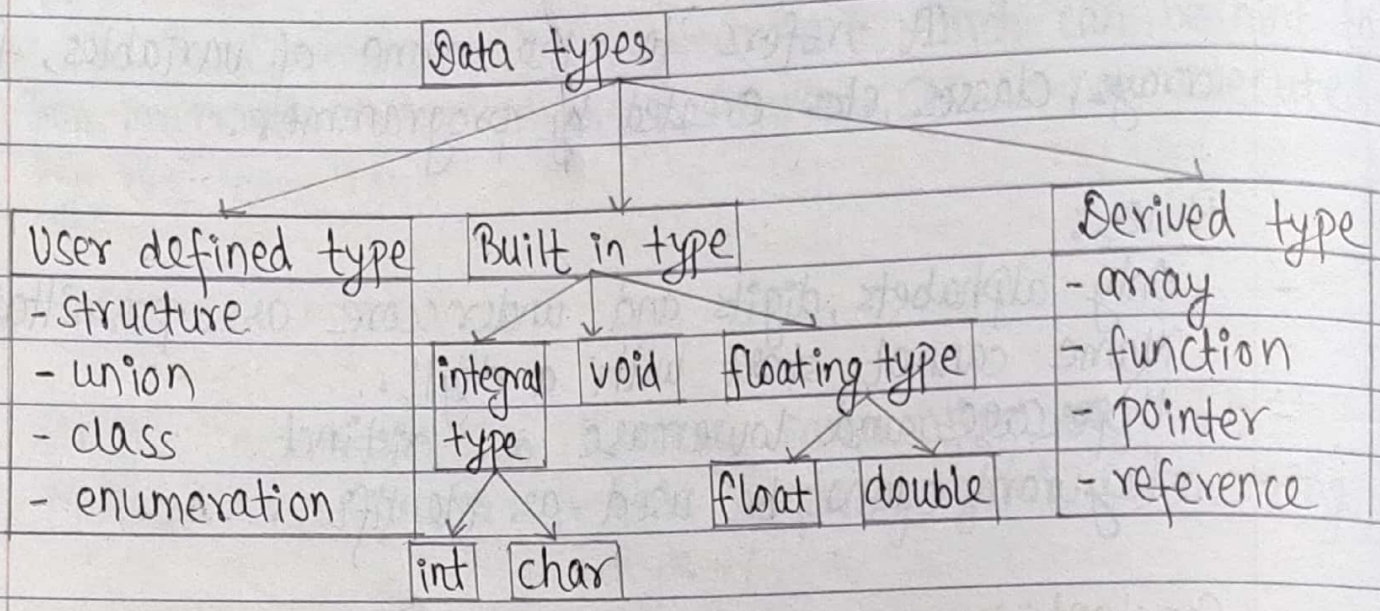


Fig:- hierarchy of C++ data types

Type	Byte	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
long int	4	
float	4	
double	8	
long double	10	

// calculate simple interest

```
#include <iostream.h>
#include <conio.h>
  void main()
  {
    float p,t,r,si;
    cout << "Enter the value of p,t,r:";
     cin >>p;
     cin >>t >>r;
     si = p*t*r/100;
    cout << "simple Interest =" <<si;
     getch();
  }
```

WAP to calculate area and circumfere of circle having radius r.

```
#include <iostream.h>
#include <conio.h>
  void main()
  {
    float r,a,c;
    cout<< "Enter the radius:";
     cin>>r;
      a= 3.14*r*r;
      c= 2*3.14*r;
      cout<<"Area of circle=" <<a;
      cout <<" circumference of circle=" <<c;
      getch();
  }
```

Ask the user temp^r in fahrenheit and then convert it into celsius.

```
# include <iostream.h>
#include <conio.h>
  void main()
  {
    float f,c;
    cout << "Enter the value of temperature in fahrenheit:";
     cin>> f;
     c= 5/9.0 *(F-32);
     cout << "The temperature in celsius = "<<c;
     getch();
  }
```

WAP that asks 3 coefficients of quadratic equation and calculate its root.

```
#include <iostream.h>
#include <conio.h>
# include <math.h>
  void main()
  {
    float a,b,c,x1,x2,d;
    cout <<"Enter the value of coefficients:";
    cin>> a>>b>>c;
    d = sqrt (b*b - 4*a*c);
    x1 = (-b-d) / (2*a);
    x2 = (-b+d) / (2*a);
    cout << "The roots are "; <<x1 <<x2;
  } getch();
```

Output using cout:

```
cout << " C++ is better than C";
```

The statement causes the string in quotation marks to be displayed on the screen. The identifier cout is a predefined object that represents the standard output stream in C++. The operator << is called the insertion or put to the operator. It inserts the contents of the variable on its right to the object on its left.

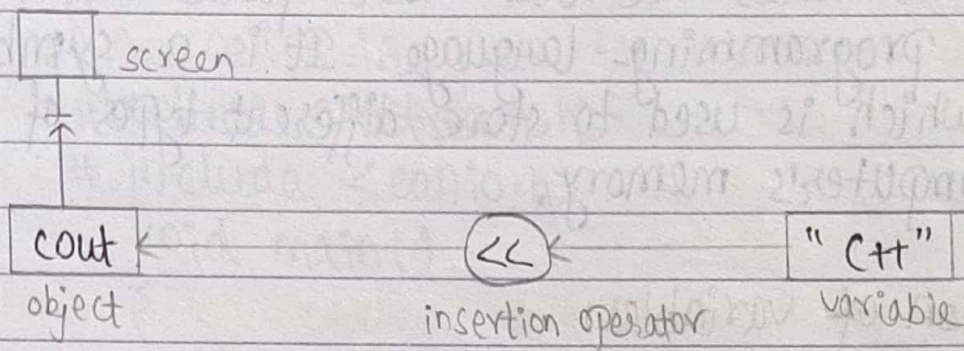


Fig: Output using insertion operator.

Input operator cin:

The statement

```
cin >> num;
```

is an input statement and causes the program to wait for the user to type in a number. The identifier cin is a predefined object in C++ that corresponds to the standard input stream. The operator >> is known as extraction or get from operator. It extracts the value from keyboard and assigns it to the variable on its right.

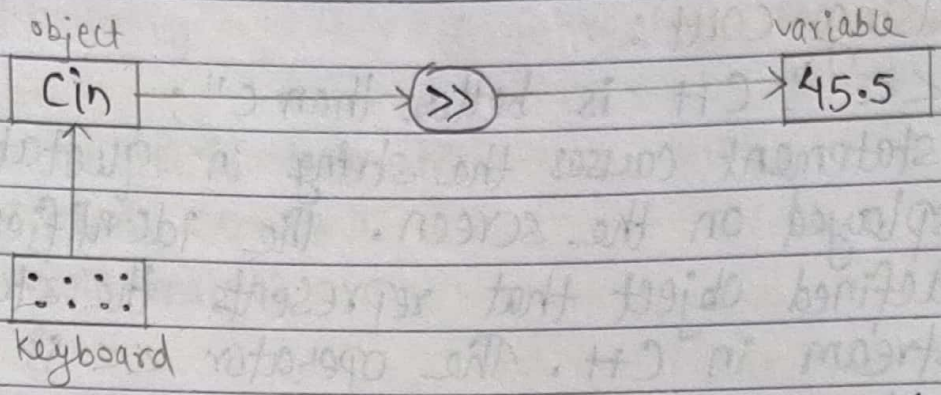


Fig: input using extraction operator.

Variables:

Variables are the most fundamental part of any programming language. It is a symbolic name which is used to store different types of data in the computer's memory.

Declaration of variables:

C++ allows the declaration of a variable anywhere in the scope.

Eg:

i) float x;
ii) int sum;
iii) char name[30];
iv) for (int i=1; i<5; i++)
{
}
v) float sum = 0

Dynamic initialization of variables:

C++ permits initialization of the variable at run time. This is referred to as dynamic initialization. A variable can be initialized at run time using expressions at the place of declaration.

Eg:

i) float area = 3.14*r*r;

ii) float avg = sum/i;

```
// Program to find area and perimeter of circle
// when radius is given by user.
#include <iostream.h>
#include <conio.h>
void main()
{
 float r,a,p;
 cout << "Enter the radius of circle:";
 cin >>r;
 a. = 3.14 *r *r;
 P = 2* 3.14 *r;
 cout<< "Area of circle = "<<a;
 cout << "Perimeter of circle = " <<p;
 getch();
}
```

Operators in C++ :-

All C operators are valid in C++ also. In addition, C++ introduces some new operators.

Operators	Name
<<	insertion operator
>>	extraction operator
::	scope resolution operator
::*	pointer to member declaration
->*	pointer to member operator
.*	pointer to member operator
delete	memory release operator
new	memory allocation operator
endl	line feed operator
setw	field width operator

Preprocessor directive:

We have used

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

This directive causes the preprocessor to add the contents of the iostream.h header file to the program. It contains declaration for the identifier cout, operator<< etc. The header file iostream should be included at the beginning of all programs that use input/output statement.

Header file:-

The different header files contain different statements. So, we have to include the header files inorder to use those statements. Some of the header files and their purpose are given below:

Header files	Purpose
<math.h>	contain function prototypes for math library functions.
<stdlib.h>	contains function prototypes for random numbers, memory allocation, etc.
<string.h>	contains function prototypes for string processing.
<time.h>	contains function prototypes for manipulating time and date

Using directives:

```
using namespace std;
```

Namespace is a new concept introduced by the ANSI C++. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the namespace scope we must include the using directive.

<stdio.h>:- It is a header file that includes function declaration (structure of function)

Library function:

Library function are the functions which can be directly used by the user and their prototype are defined in the header files. Some commonly used library functions are:

Function	Purpose
sqrt(x)	square root of x
pow(x,y)	x raised to power y
sin(x)	sine of x
strcpy(x,y)	copy contents of string variable y to x.
strupr(x)	convert all characters into uppercase.

etc.

UNIT :2

CONTROL STRUCTURES

Control Structure :-

In programming, we have to use different sequence of execution according to the program. There are 3 different types of control structures:

1. Sequential structure (Straight Line)
2. Selection structure (Branching)
3. Loop structure (iteration or repetition)

1. Sequential structure (Straight Line) :-

In sequential structure, the program codes or statements are executed one after another serially. There will be no condition or looping.

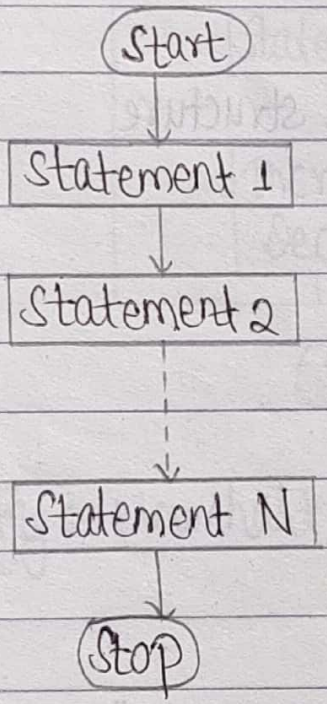


Fig :- Flowchart of sequential structure

Eg :-

```
#include <iostream.h>
#include <conio.h>
void main()
{
cout << "Hello World"
getch();
}
```

2. Selection or decision control structure :-

Different sections of program are executed on the basis of given conditions in terms of true or false, then decision is made how to program flow or control will jump.

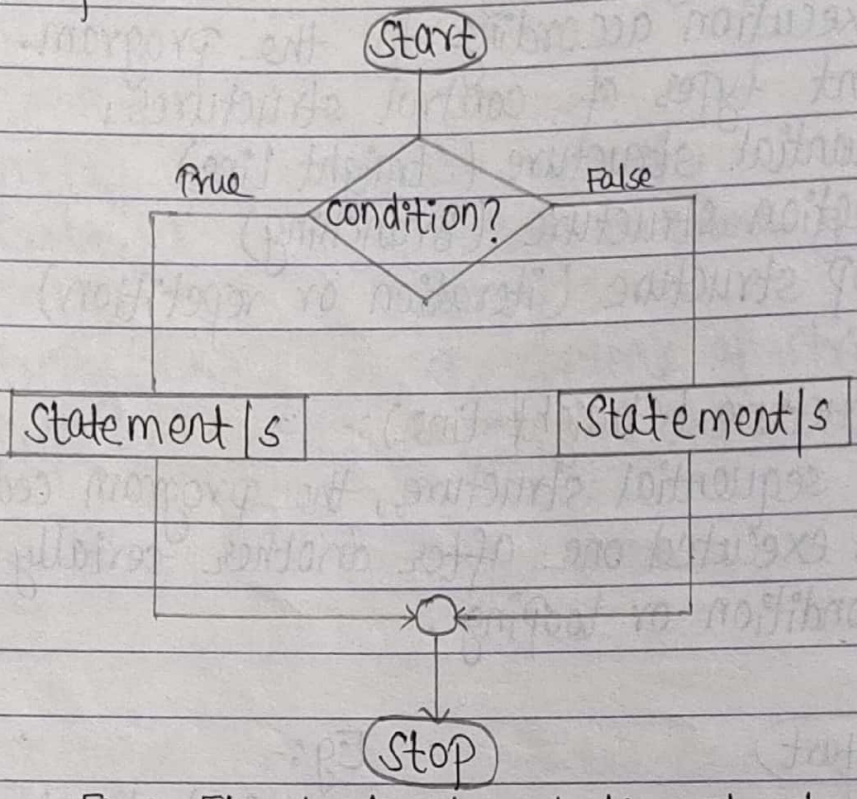


Fig :- Flowchart of selection structure

Example:

```
#include <iostream.h>
# include <conio.h>
  void main()
  {
    int x=100;
    if (x==100)
    {
      cout<<" The number is 100";
    }else
    {
  } }cout<<" The number is not 100";
```

3. Looping structure (iteration or repetition):-

Block of program statements are executed upto a fixed number of times or until a condition is satisfied. Following are the looping statements that are generally used:

- For loop
- While loop
- do while loop

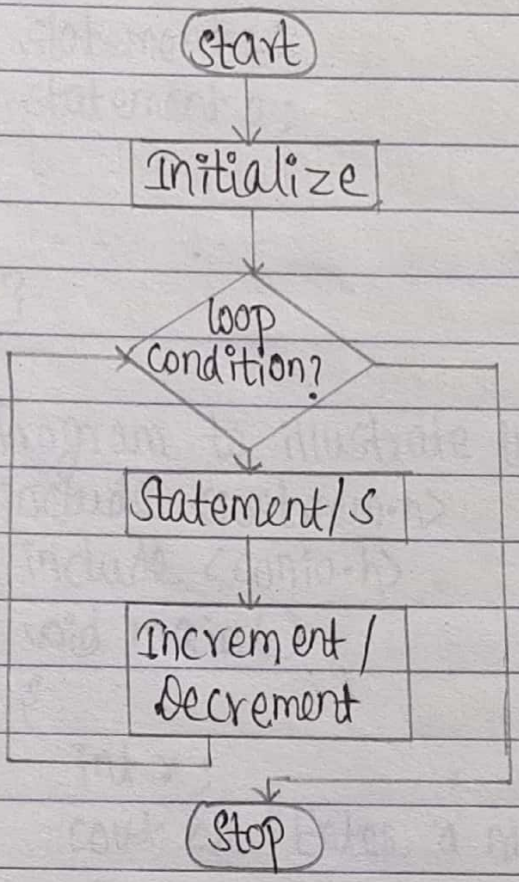


Fig:- Flowchart of looping structure.

Selection structure :-

→ If - statement :-

It is used to execute an instruction or block of instruction only if a condition is fulfilled.

Syntax:- for single statement

```
if (condition)
  {
    statement 1;
    statement 2;
    ------
    ------
  }
```

Example:- Program to illustrate the if statement

```
#include <iostream.h>
# include <conio.h>
  void main()
  {
    int x;
    cout << "Enter a number:";
    cin >> x;
    if (x==100)
     cout << "The number is equal to 100";
    getch();
  }
```

→ If... else statement :-

This statement is bi-directional statement because when the condition is true, then the control is transferred to one part of the program and when the condition is false, the control is transferred to another part of the program.

Syntax :-

For single statement

```
if (condition)
   statement;
else
   statement;
```

For multiple statement

```
if (condition)
{
   statement 1;
   statement 2;
   ---------
}
else
{
   statement 1;
   statement 2;
   ---------
}
```

Example :- Program to illustrate the if....else statement.

```
#include<iostream.h>
#include<conio.h>
void main()
{
   int x;
   cout<<"Enter a number:";
```

```
cin >> x;
if (x = 100)
 cout << " The number is equal to 100";
else
   cout << " The number is not equal to 100";
   getch();
}
```

→ Nested if...else statement:-

If we use another if...else statement in the block of if or the else, then this is known as nested if...else statement.

Syntax:-

```
if (condition)
 {
   if (condition)
     statement;
   else
     statement;
  }
   else
    {
     if (condition)
      statement;
     else
      statement;
    }
```

Example:-

WAP to find largest no. from 3 numbers.

```
#include <iostream.h>
#include <conio.h>
void main()
{
 float a,b,c;
 cout << "Enter three numbers :";
 cin >> a >> b >> c;
 if (a>=b)
  {
   if(a>=c)
   cout<< "The greatest is :" << a;
   else
    cout << "The greatest is :" <<c;
  }
 else
   {
    if (b>=c)
    cout << "The greatest is :" <<b;
    else
     cout << "The greatest is :" <<c;
   }
}
```

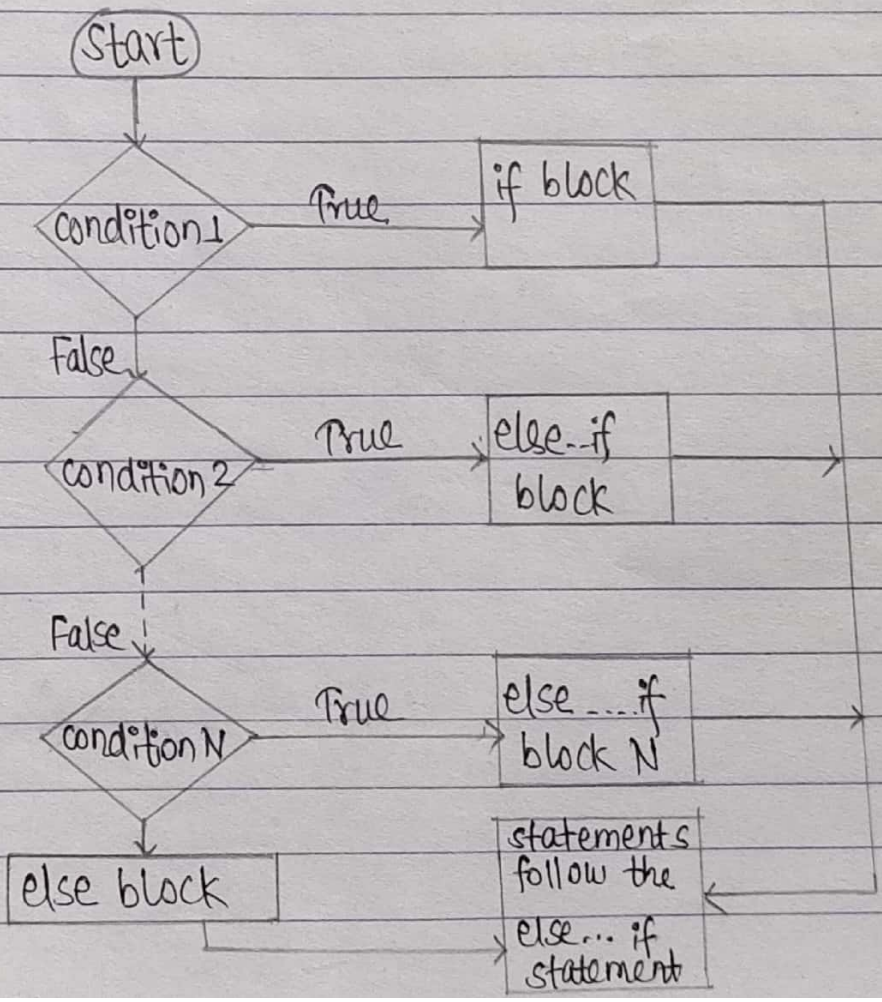
Imp:- → The else.... if construction (else..... if ladder)

There is an if... else statement in every statement (else statement) except the last else part.

Syntax:-

```
if (condition1)
  statement 1;
else if (condition 2)
  statement 2;
else if (condition 3)
  statement 3;
  ---------
else
  statement;
```

Flowchart:-



Example:-

WAP to find the largest number from the given 3 numbers.

```
#include<iostream.h>
#include<conio.h>
  void main()
  {
    float a,b,c;
    cout<<"Enter the three numbers:";
    cin>>a>>b>>c;
    if (a>=b)
```

The Switch statement :-

It is a multi-way decision statement. It allows a variable to be tested for equality against a list of ~~variables~~ values. Each value is called a case and a variable being switched is checked for each case.

Syntax :-

```
switch (expression)
{
  case constant-expression1:
     statement (s);
     break;
  case constant-expression2:
     statement (s);
     break;
  - - - - - - - -
  - - - - - - - -
  default:
     statement (s);
}
```

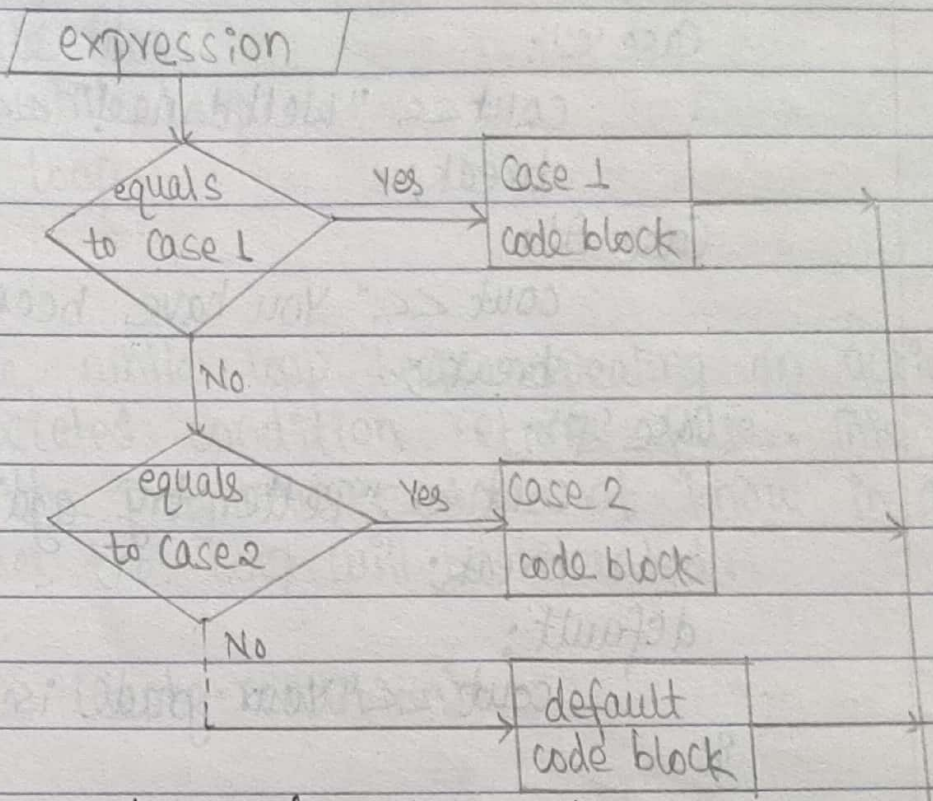


Fig:- Flowchart of switch statement.

Example:-

WAP to identify the grade and publish the result

```
# include <iostream.h>
# include <conio.h>
  void main()
  {
    char grade = 'B';
    switch (grade)
     {
       case 'A':
          cout << "Excellent" << endl;
          break;
```

```
Case 'B':
Case 'C':
        cout<< "Well done!" <<endl;
        break;
Case 'D':
        cout <<" You have been passed" <<endl;
        break;
Case 'F':
        cout <<"Better try again" << endl;
        break;
default:
        cout <<"Your grade is not valid" << endl;
    }
    cout<<"Your grade is :" << grade <<endl;
}
```

Output:-

You have been passed
Your grade is D.

Looping Structure:-

Looping means executing the same section of code more than once if the given condition is true. The repetition continues while a condition is true. When the condition becomes false, the loop ends and control passes to the statement following the loop.

Types of Looping Structure (statements)

- While loop
- do while loop
- for loop

1) While loop :-

The while loop keeps repeating an action until an associated condition returns false. This is useful where the programmer does not know in advance how many times the loop will be iterated.

Syntax :-

```
while (test-condition)
{
  statement (s);
}
```

Flowchart :-

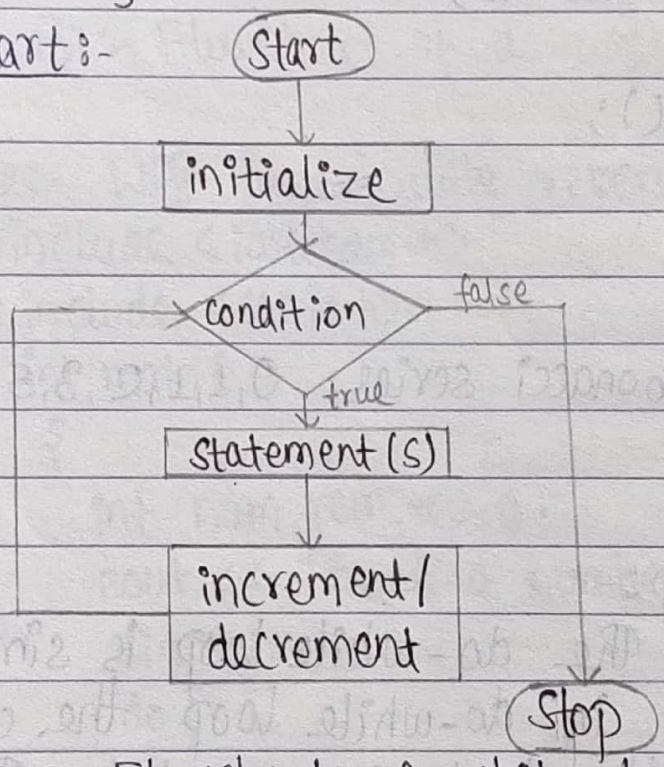


Fig :- Flowchart of while loop

Example:- WAP to generate fibonacci number between 1 to 100.

```
#include <iostream.h>
#include <conio.h>
void main()
{
  int t1=0, t2=1, t3=0;
  cout<<"The fibonacci series:"<<t1<<","<<t2<<",";
  t3 = t1+t2;
  while (t3<=100)
  {
    cout<<t3<<", ";
    t1=t2;
    t2=t3;
    t3=t1+t2;
  }
  getch();
}
```

Output:-

The fibonacci series: 0,1,1,2,3,5,8,13,21,34,55,89.

11) do-while loop:-

The do-while loop is similar to while loop but in do-while loop the condition is checked after the body is executed. This ensures that the loop body runs at least once if the condt

is false. It is exit control or post-test loop structure.

Syntax:-

```
do
{
  statement(s);
} while (test-condition);
```

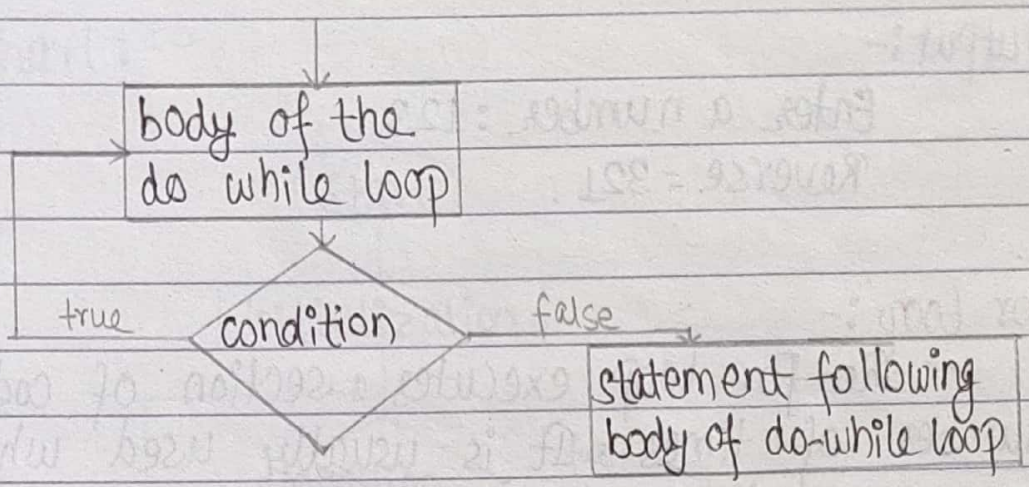


Fig:- Flowchart of do-while loop.

Example:- WAP to calculate reverse of a number.

```
#include <iostream.h>
#include <conio.h>
int main()
{
  int num, rem, rev=0;
  cout << "Enter a number:";
  cin >> num;
  do {
      rem = num % 10;
      rev = rev * 10 + rem;
```

```
        num = num/10;
    } while (num>0);
    cout << "Reverse = " << rev << endl;
    getch();
    return 0;
}
```

Output:-

Enter a number :123
Reverse = 321.

iii) For Loop :-

The for loop executes a section of code in a fix number of times. It is usually used when we know how many times loop will execute before entering to the loop.

Syntax :-

```
for (initialization ; test condition ; incr/decr)
    {
      statement (s);
    }
```

- initialization is a initial value setting for a counter variable.
- test condition is checked, if it is true the loop continue otherwise loop finishes.

- statement(s) can be single statement or block of statement.
- increment/decrement is value of increment/decrement for counter.

Flowchart:-

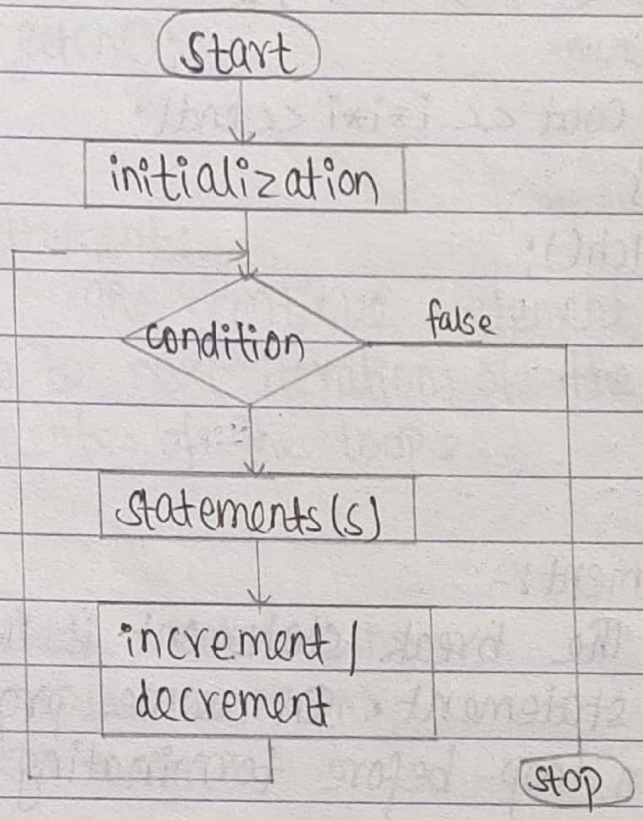


Fig:- Flowchart of for loop

Example:-
WAP to print cube of numbers from 1-10.

```
#include <iostream.h>
# include <conio.h>
  void main()
  {
    int i;
    for (i=1; i<=10; i++)
    {
      cout << i*i*i <<endl;
    }
    getch();
  }
```

Break statement:-

The break statement is used inside loops and switch statement. In some program, if we want to exit from loop before terminating loop, we can use this statement.

Example:-

```
#include<iostream.h>
 # include<conio.h>
   void main()
   {
     int n;
```

```
for (n=1; n<=10; n++)
{
  if (n==5)
  {
    cout << loop breaked at n=5" <<endl;
    break;
  }
  cout <<"number=" <<n <<endl;
}
getch();
}
```

Output:-
number=1
number=2
number=3
number=4
loop breaked at n=5

Continue statement:-

The continue statement is used when we want to go to next iteration of the loop after skipping some statements of the loop.

Example:-

```
#include <iostream.h>
#include <conio.h>
void main()
{
  int n;
  for (n=1; n<=10; n++)
  {
    if (n==3)
    {
      cout << number 3 is skipped" <<endl;
      con
```

```
        continue;
      }
      cout<<" number ="<<n<<endl;
   }
   getch();
}
```

Output:-

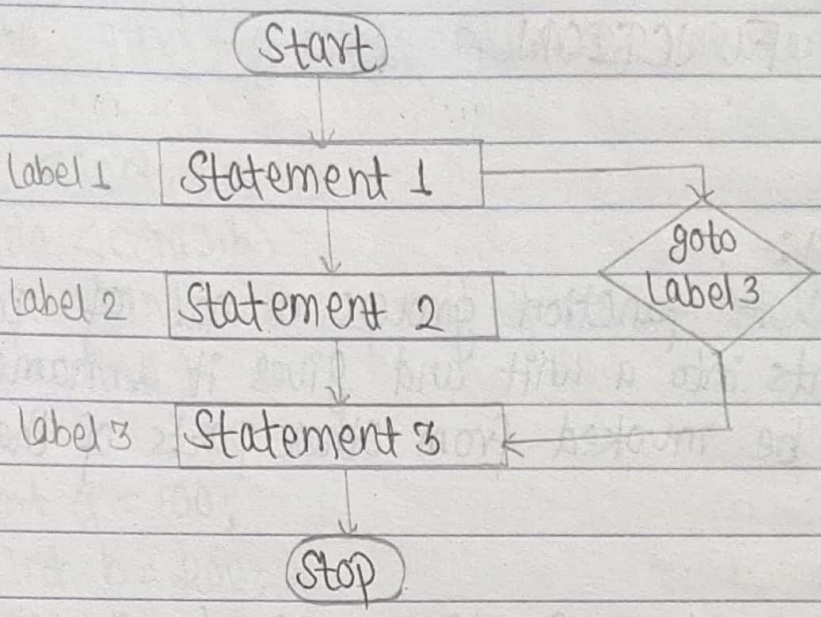
number = 1
number = 2
number 3 is skipped.
number = 4
number = 5
number = 6
number = 7
number = 8
number = 9
number = 10

goto statement:-

A goto statement provides an unconditional jump from goto to a labelled statement in the same function.

Syntax:

```
goto label;
--------
--------
label : statement;
```



Example:-

```
#include <iostream.h>
#include <conio.h>
 int main()
 {
   int i=10;
   goto jump;
   cout<<i<<","<<endl;
   i--;
   if (i>=0)
   {
    goto gotojump;
   }
   cout<<"finished"<<endl;
   getch();
   return;
 }
```

FUNCTION

Function:-

A function groups a no. of program statements into a unit and gives it a name. The unit can be invoked from other parts of the program.

Syntax:

```
return_type function_name (parameter list);
                    //function declaration
void main()
{
  ---------
  ---------
  function_name (parameter list); //function calling
  ---------
  ---------
}
return_type function_name (parameter list)
{                                  //function definition
  ---------
  ---------
  ---------

}
```

- A function declaration tells the compiler about a function's name, return type and parameters.
- Function call statement causes the function to execute.
- A function definition provides the actual body of the func

Example:-
WAP to find greatest number between two numbers.

```
#include <iostream.h>
#include <conio.h>
   int max(int, int);   // function declaration
   void main()
    {
       int a=100;
       int b=200;
       int rel;
       rel = max (a,b);    // function call
       cout << " The greatest number =" << rel;
       getch();
    }
    int max (int num1, int num2)
     {
       int result;
       if (num1>num2)
           result = num1;
       else
           result = num2;
       return result;
     }
```

Function prototyping / Declaration:

Prototype of a function is the function without its body. Function prototyping is one of the major improvement added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments, the function name and the type of return value.

Syntax:-

return_type function_name (argument list);

Example:-

float volume (float a, float b, int c);

In a function declaration, the names of the arguments are the dummy variable and therefore they are optional.

eg:- float volume (float, float, int);

Macros:-

Function call is a costly operation. During the function call and its execution, our system takes overheads like saving the values of registers, saving the return addresses, pushing arguments into the stack, jumping to the called function, loading registers with new values, returning to the calling function and loading the registers with previously stored values. For

a large function, this overhead is negligible but for small function, taking such a large overhead is not justifiable.

Macros are defined to reduce the function call overhead incase of small functions. Macros are defined by using hash(#) sign because they are processed by macro processors not by compilers.

Example:- Program to explain the use of macros.

```
#include <iostream.h>
#include <conio.h>
# define mul (a,b) a*b    // macro definition
# define div (x,y) x/y
void main()
{
  int a=5, b=7;
  int x = 10, y=5;
  cout <<"mul =" <<mul (a,b) << endl;  // macrocall
  cout <<"div =" << div (x,y) <<endl;
  getch();
}
```

jaha nera function call hunxa tei nera body {function declaration} lyera rakhxa

Inline functions:-

When the compiler sees a function call, it automatically generates a jump to the function. At the end of the function, it jumps back to the instruction following the call. This sequence of events may save memory space but it takes some extra time. The solution of

this problem is inline function.

A function which is expanded in line by the compiler each time its call is appeared instead of jumping to the called function as usual is called inline function.

Example:- Program to calculate sum of two numbers using inline function.

```
#include <iostream.h>
#include <conio.h>
inline int sum (int a, int b)
{
  int s;
  s=a+b;
  return s;
}
void main()
{
  int x,y;
  cout<<"Enter the value of x & y"<<endl;
  cin>>x>>y;
  cout<<"Sum"<<sum(x,y);
  getch();
}
```

O/P:-

Enter the value of x & y

6

1

sum 7

Advantage:
- No function call overhead therefore execution of the program becomes fast.
- Better error checking is performed as compared to macros.
- It can access members of a class which cannot be done with macros.

Disadvantage:
- If the function call is made repeatedly, object code size increases and thus requires more memory at the time of execution.
- The compiler cannot perform inlining if the function is too complicated. For example; functions cannot be expanded inline if it contains recursive call.
- The compiler also cannot perform inlining if the address of the function is taken implicitly or explicitly.

same function name lai different work garaune → function overloading.

Overloaded function:-

Function overloading is a mechanism that allows a single function name to be used for different functions. The compiler does the rest of the jobs to choose the particular function. It matches the argument numbers and types to determine which functions are being called. The function overloading can be done in two ways:-

i) Function overloading with different types of argumen
ii) Function overloading with different number of arguments.

Example 1:

// Function overloading with different types of arguments.

```
#include <iostream.h>
#include <conio.h>
  int mul (int, int);
  float mul (float, float);
  void main()
  {
    int a=5, b=7;
    float x=5.0, y=6.0;
    cout<<"Multiplication of integer = "<<mul(a,b)<<
    cout<<"Multiplication of float = "<<mul(x,y)<<endl;
    getch();
  }
    int mul (int m, int n)
    {
      return m*n;
    }
    float mul (float m, float n)
    {
      return m*n;
    }
```

O/P

Multiplication of integer = 35
Multiplication of float = 30.00

Example 2:

// Function overloading with different number of arguments

```
#include<iostream.h>
#include<conio.h>
  float area (float,float);
  float area (float);
  void main()
   {
    float sa,ra,l,b;
    cout<<"Enter length and breadth of rectangle:"<<endl;
    cin>>l>>b;
     ra = area(l,b);
    cout<<"Enter the length of square:"<<endl;
    cin>>l;
     sa = area(l);
    cout<<" The area of rectangle ="<<ra;
    cout<<" The area of square ="<<sa;
    getch();
   }
   float area (float x, float y)
    {
      return x*y;
    }
    float area (float x)
    {
      return x*x;
    }
```

Recursion:-

A function that calls itself is known as recursive function and this technique is known as recursion.

Example:- Program to calculate factorial of given number

```
#include <iostream.h>
#include <conio.h>
  long int factfun (long int);
  int main ()
  {
    long int n;
    long int fact;
    fact = factfun (n);
    cout << "The factorial of given number is:" << fact;
    getch();
    return 0;
  }
  long int factfun (long int n)
  {
    if(n=0)
      return 1;
    else
      return (factfun(n) * factfun(n-1));
  }
```

Passing argument to the functions:

We can pass arguments to a function in three ways:

1) Pass by value
2) Pass by reference
3) Pass by pointer.

1) Pass by value:-

In case of pass by value, copies of the arguments are passed to the function not the variable themselves.

Example:-

```
// Pass by value
#include <iostream.h>
#include <conio.h>
void exchange (int, int);
void main()
{
  int x=3, y=5;
  cout<<"Before the function call"<<endl;
  cout<<"x="<<x<<endl<<"y="<<y;
  exchange(x,y);
  cout<<"After function call"<<endl;
  cout<<"x="<<x<<endl<<"y="<<y;
  getch();
}
void exchange (int a, int b)
{
  int temp;
```

```
    temp = a;
    a = b;
    b = temp;
}
```

Output:-

Before function call
x=3
y=5
After function call
x=3
y=5

II) Pass by reference:-

In case of pass by reference, address of the variable (variable itself) not copies of the arguments are passed to the function.

Example:-

```
// Pass by reference.
#include <iostream.h>
# include <conio.h>
  void exchange (int &, int&);
  void main
  {
    int x=3, y=5;
    cout<<"Before the function call"<<endl;
    cout<<"x ="<<x<<endl<<"y = "<<y;
    exchange (x, y);
```

```
    cout<< "After the function call" <<endl;
    cout<< "x=" <<x<<endl<< "y=" <<y;
    getch();
}
void exchange (int &a, int &b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}
```

Output:-

Before the function call
x=3
y=5
After the function call
x=5
y=3

iii) Pass by pointer:-

Working principle of pass by pointer is same as the pass by reference. Only the difference is that instead of using reference variable we use pointer variable in function definition and address of the variable from the function call statement.

Example:-

```
//Pass by pointer
#include <iostream.h>
#include <conio.h>
  void exchange (int *, int*);
  void main()
  {
    int x=3, y=5;
    cout<< "Before the function call "<<endl;
    cout<<" x="<<x <<endl <<" y="<<y;
    exchange (&x, &y);
    cout<< "After the function call" <<endl;
    cout<<"x=" <<x <<endl <<"y="<<y;
    getch();
  }
    void exchange (int *a, int *b)
    {
       int temp;
       temp = *a;
      *a = *b;
      *b= temp;
  }
```

Default argument:-

When declaring a function we can specify a default value for each of the last parameters which are called default arguments. If a value for that parameters is not passed when the function is called then the default value is used but if a value is specified this default value is ignored and the passed value is used instead.

Example:-

```
// Default argument
#include <iostream.h>
#include <conio.h>
void divide (int a, int b=2)
{
int result;
  result= a/b;
  cout<<"The result is =" <<result;
}
void main()
 {
    divide (12,4);
     cout<< endl;
     divide (12);
     getch();
   }
```

O/P

```
The result is = 3
The result is = 6
```

storage class → life time

visibility

Scope & storage class of a variable :-

The scope of a variable determines which parts of the program can access it and its storage class determines how long it stays in existence.

Two different kinds of scope are :

1) local scope - variables with local scope are visible only within a block.

2) file scope - variables with file scope are visible throughout a file.

There are two storage classes :

1) automatic storage class - variables with automatic storage class exists during lifetime of the function in which they are defined.

2) static storage class - variables with static storage class exists for the life time of the program.

Local variables :-

The variables that is defined within a function.

1) Storage class - a local variable is not created until the function in which it is defined is called.

2) Scope - a variables scope or visibility describes the location within a program from which it can be accessed.

Global variables:-

i) storage class - global variables have storage class static i.e. they exist for the life of the program.

ii) scope - global variables are visible in the file in which they are defined starting at the point where they are defined.

Static local variable:-

Static local variables are used when it is necessary for a function to remember a value when it is not being executed.

i) storage class - Its lifetime is same as that of global variable except that it does not come into existence until the first call to the function containing it.

ii) scope - a static local variable has the visibility of an automatic local variable

2.3. Arrays

An array is a consecutive group of memory locations that all have the same type. To refer to a particular location or element in the array, we specify the name of the array and the position no. of the particular element in the array.

Declaration of array:

Syntax :-

```
data_type array_name [size];
```

eg:-

```
int number [10];
float salary [5];
char name [20];
```

Initialization of array:

Syntax :-

```
data_type array_name [size] = {val1, val2, ... valN}
```

eg:-

```
int number [10] = {1,2,3,4,5,6,7,8,9,10};
float salary [5] = { 3000.5, 4000.75, 5000.25, 1000.2, 3000.4};
```

Reading array element:

- Reading for an array num [5] is done as follows:

```
for (int i=1; i<5; i++)
{
  cin>> num [i];
}
```

Displaying values from an array:

- Let array num[5].

```
for (int i=1; i< 5; i++)
{
  cout << num[i];
}
```

Example 1:- WAP to find the average of an array elements.

```
#include <iostream.h>
#include <conio.h>
  int main()
{
  constant int SIZE =5;
  int i;
 float total =0, average, arr[SIZE];
 cout <<" Enter the sales of 5 days:" <<endl;
 for (i=0; i<=5; i++)
 {
   cin >> arr[i];
 }
  for (i=0; i<5; i++)
  {
    total += arr[i];
  }
  average = total / SIZE;
  cout<<" The average =" << average <<endl;
  return 0;
  getch();
}
```

Example 2:- WAP to find the largest and smallest number from the given array.

```
#include <iostream.h>
#include <conio.h>
int main()
{
  int SIZE=4;
  int i, arr[4]={6,10,4,2};
  int min, max;
  min = max = arr[0];
  for (i=0; i<4; i++)
  {
    if (arr[i] < min)
    {
      min = arr[i];
    }
    if (arr[i] > max)
    {
      max = arr[i];
    }
  }
  cout<<"The largest number = "<<max<<endl;
  cout<<"The smallest number = "<<min<<endl;
  getch();
}
```

Types of Array :-

There are two types of arrays:

1. One dimensional array
2. Two (multi) dimensional array.

1. One dimensional array :-

This type of array represents and store data in linear form. It is also called single dimensional array. The general syntax to declare one dimensional array is:

```
data_type array_name[size];
```

eg:
```
int number[5];
char name[20];
```

2. Two dimensional array :-

This type of array represents and store data in matrix form. The general syntax to declare two dimensional array is:

```
data_type array_name[Rows][Columns];
```

where :→ array_name is a named constant which cannot be changed during the execution of program.

→ [] is subscript operator to mentions the rows and columns of two dimensional array.

Eg :-

```
int arr[4][5];
```

Initialization of two dimensional array:

Along with declaration, a two-dimensional array is initialized as follows:

Syntax:-

data-type array_name[row][column] = { {val11, val12, ...val
{val21, val22, ...val

---
val n1, val n2, ... val n

Example:

```
int num[3][4] = { {1,2,3,4},
                  {4,5,86,7},
                  {8,9,10,11} };
```

Example 1:- WAP to read and find the sum of even elem of a matrix.

```
#include <iostream.h>
#include <conio.h>
 void main()
  {
   int i, j, mat[3][4], sum=0;
   cout<<"Enter the elements of matrix:\n";
   for (i=0; i<3; i++)
    {
     for (j=0; j<4; j++)
      {
       cin>> mat[i][j];
    } }
```

```
for (i=0; i<3 ; i++)
  {
    for (j=0; j<4; j++)
      {
        if (mat[i][j]%2 ==0)
          {
            sum += mat [i][j];
          }
      }
  }
  cout<<" The sum of even element is:"<<sum;
  getch();
}
```

Example 2:- WAP to find the sum of diagonals of a matrix.

```
#include <iostream.h>
#include <conio.h>
 void main()
  {
   int i,j, mat[3][4], sum=0;
   cout<<"Enter the elements of matrix :\n";
   for (i=0; i<3; i++)
    {
      for (j=0; j<4; j++)
        {
          cin>> mat [i][j];
        }
    }
```

```
for (i=0; i<3; i++)
  {
    for (j=0; j<4; j++)
     {
       if (mat[i] == mat[j])
        {
          sum += mat[i][j];
        }
     }
  }
cout<<" The sum of diagonals of a matrix is:" <<sum;
getch();
}
```

Passing arrays to functions:

- Arrays can be used as arguments to function.

Function declaration with array arguments:

In a function declaration, array arguments are represented by data type and size of an array.

For example:- function prototype for the function modifyArray might be written as;

```
void modifyArray [(int[], sizeofarray).
```

This prototype can also be written as;

```
void modifyArray (int arrayName[], sizeofarray);
```

Function call with array arguments:

To pass an array arguments to a function, specify the name of the array without any brackets.
For example:- if an array hourlyTemperature has been declared as;

```
int hourlyTemperature[24];
```

then the function call is;

```
modifyArray (hourlyTemperature, sizeofarray);
```

Function definition with array arguments:

In the function definition, the declarator look like;

```
void modifyArray (int hourlyTemperature[], sizeofarray)
{
  - - - - - -
  - - - - - -
  - - - - - - -
}
```

Example:- WAP to calculate sum of two matrices.

```
#include <iostream.h>
#include <conio.h>
int sum1(int[][], int[][], int row, int column);
int display( int[][], int row, int column);
void main()
{
  int a[3][3] = { {1,2,3},
                  {4,5,6},
                  {7,8,9} };
  int b[3][3] = { {7,8,9},
                  {7,7,6},
                  {2,4,7}};
  int sum[3][3],row,column;
  sum1(a,b,3,3);
  int sum[3][3] = sum1(a,b,3,3);
    display (sum,3,3);
    getch();
}
int sum (int a[3][3], int b[3][3], int row,
  int column)
{
 int sum[3][3], i, j;
 for(i=0; i<3; i++)
{
   for (j=0; j<3; j++);
    {
      sum[i][j] = a[i][j] + b[i][j];
    }
}
```

```
    return sum;
}
int display (int sum[3][3] , int row,int column);
{ for (i=0; i<3; i++)
    {
      for (j=0; j<3; j++)
        {
          cout <<"sum of two matricess :"<<sum[i][j];
        }
    }
}
```

2.4 Pointers

A pointer is a variable whose value is the address of another variable. Like any variable or constant it must be declared before using it.

Pointer variable declaration and initialization:

The general form of pointer variable declaration is;

```
type *var_name;
```

Here, type is the pointer's base type. It must be a valid C++ type and variable name is the name of the pointer variable. The asterik (*) is used to declare the pointer variable.

Pointer should be initialized when they are declared or in an assignment. A pointer may be initialized to 0, NULL or an address. A pointer with the value 0 or NULL points to nothing and is known as a NULL pointer. Symbolic constant NULL is defined in header file <iostream.h> to represent the value 0.
Initializing a pointer to NULL is equivalent to initializing a pointer to 0. When 0 is assigned, it is converted to a pointer of appropriate type.
Example;

```
int x
int *ptr
ptr = NULL;
ptr = &x;
ptr = 0.
```

Pointer operators:

The address operator (&) is a unary operator that returns the memory address of its operand.
For example;

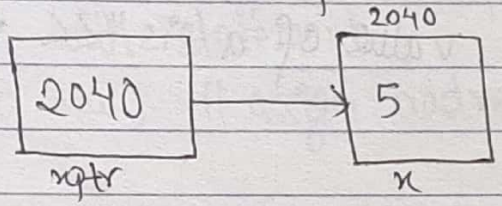
assuming the declaration

```
int x = 5;      // declare variable x
int *xptr;      // declare pointer variable xptr
```

Then the statement

```
xptr = &x;  // assign the address of x to xptr
```

assigns the address of variable x to pointer variable xptr. The variable xptr is said to "point to" x.



The asterik operator commonly referred to as indirection operator or dereferencing operator and returns a synonym for the object to which its pointer operand points. For example;
The statement,

```
cout << *xptr;
```

prints the value of variable x just as statement

```
cout << x;
```

Example:-

Program to illustrate pointer variable & pointer operator

```
#include <iostream.h>
#include <conio.h>
  int main()
  {
   int a;  // a is an integer variable
   int *aptr;  //aptr is a pointer variable of type integer
    a=7;    // assigned 7 to a
   aptr = &a;  // assigned address of a to aptr
   cout<<" The value of a is "<<a;
   cout<<"\n The value of aptr is "<<aptr;
   cout<<"\n The address of a is "<<&a;
   cout<<"\n The value of a is "<< *aptr;
    getch();
    return 0;
  }
```

Relationship between arrays and pointers:

Arrays and pointers are intimately related in C++ and may be used almost interchangeably. An array name can be thought of as a constant pointer. Pointers can be used to do any operations involving array subscripting. Assume the following declaration;

```
int a[5];   // create an array a with 5 elements
int *ptr;   // create int pointer ptr
```

Because the array name (without a subscript) is a pointer to the first element of the array. We can set ptr to the address of the first element in array a with statement;

```
ptr = a;  // assign address of a to ptr.
```

This is equivalent to taking the address of the first element of the array as follows:

```
ptr = &a[0];  // also assigns the address of a to ptr
```

Array element a[3] can alternatively be referred with the pointer expression *(ptr + 3)

Example:-
Program to illustrate the subscripting and pointer notation with arrays.

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

Arrays of pointer :-

Arrays may contain pointers. A common use of such a data structure is to form an array of pointer based strings, referred to simply as a string array.

```
#include<iostream.h>
#include<conio.h>
  int main()
  {
    int a[] = {10,20,30};
    int *aptr[3];
    cout<<"Array printed with array subscript\n";
    for(int i=0; i<3; i++)
     {
       cout<<a[i]<<"\n";
       aptr[i] = &a[i];
     }
    cout<<"Array printed with pointer\n";
    for(int i=0; i<3; i++)
     {
       cout<<*aptr[i]<<"\n";
     }
    getch();
    return 0;
  }
```

Function pointers:-

A pointer to a function contains the address of the function in memory. The name of a function is actually the starting address of memory of the code that performs the function's task.

Pointers to a function can be passed to a function, return from function, stored in arrays and assigned to other function pointers.

Example:

```
// Program to illustrate function pointer
#include <iostream.h>
#include <conio.h>
int add (int a, int b)
{
  return a+b;
}
int main()
{
  int sum;
  int (*p)(int, int);
  p= add;          OR //P = &add;
  sum = (*p)(2,4); OR // p(2,4);
  cout << "Sum =" << sum;
  getch();
  return 0;
}
```

Unit-3

3.1. Class and Object

Structure :-

A structure is an user-defined data type which contains the collection of different type of data under the same name.

Syntax:

```
struct structure_name
{
    data_type var_name;
    data_type var_name;
    - - - - - - - - -
    - - - - - - - - -
};
```

Example:

```
struct currency
{
    int rs;
    float ps;
};
```

Declaration of structure variable:

struct_name var_name1, var_name2, ...

Example:

currency c1, c2;

Initialization of structure variable:

currency c1 = {150, 86.5}; /* initialization at the time of declaration */

currency c1;
c1 = {150, 86.5}; // error

c1.rs = 150; /* initialization of variable separately */
c1.ps = 86.5;

Example:

```
#include<iostream.h>
#include<conio.h>
 struct currency
 {
    int rs;
    float ps;
 };
 int main()
 {
    currency c1, c3;
    currency c2 = {150, 86.5};
```

```
cout<<"Enter Rupees:"; cin>>c1.rs;
cout<<"Enter Paisa :"; cin>>c1.ps;
c3.ps = c1.ps + c2.ps;
if (c3.ps>=100)
{
   c3.ps=c3.ps-100;
   c3.rs++;
}
c3.rs = c3.rs+c2.rs+c1.rs;
cout<<"Rupees:"<<c3.rs;
cout<<"\n Paisa :"<<c3.ps;
getch();
return 0;
}
```

Improvement in C++ structure:

C++ has made following 3 extensions to C structure which makes the C++ structures more powerful than C structure.

a) C++ structure allows us to define functions as a member of structures.

Syntax:

```
struct employee
{
  - - - - -
  - - - - - -
  void getdata()
  {
  // function body
  } }
```

b) C++ structure allows us to use the access specifiers private and public inside the structure

~~Syntax:~~ Example:

```
struct employee
{
  private:
    int eid;
    float salary;
  public:
    void getdata()
    {
      //function body
    }
```

c) C++ structure allows us to use structures similar to that of primitive data types while defining variables.

```
struct employee e1 // C style
employee e1 // C++ style
```

Class & Objects:

Class is a collection of logically related data items and the associated function which operate and manipulate those data. This entire collection can be called new data type. So classes are user-defined data types and behaves like built-in types of a programming language.

Classes allow the data and functions to be hidden from the external use. Class being a user-defined data type, we can create any no. of variables for that class. These variables are called objects.

Specification of class: (declaration or definition of class)

The general form of class declaration is:

Syntax:

```
class class_name
{
    private:
        variable declarations
        function declarations
    public:
        variable declarations
        function declarations
};
```

- The class specification starts with a keyword class followed by a class name. The class name is an identifier.

- The body of a class is enclosed within brackets braces and terminated by a semicolon.
- The functions and variables are collectively called class members. The variables are called data members whereas the functions are called member functions.
- The two new keyword inside the specification above are private and public. These keywords are termed as access specifiers and they are also called as visibility labels.

Example:

```
class rectangle
{
  private:
     int length;
     int breadth;
  public:
     void setdata (int l, int b)
     {
       length = l;
       breadth = b;
     }
     void displaydata ()
     {
      cout << "length = " << length << endl << "Breadth = "
           << breadth << endl;
     }
```

```
int findarea()
{
  return length*breadth;
}
int findperimeter()
{
  return 2*(length+breadth);
}
};
```

Creating Objects

Once a class has been declared, we can create variables of that type using the class name.

Example:- rectangle r;

The above statement creates a variable r of type rectangle. The class variables are known as objects so r is called object of class rectangle. We can create any no. of objects from the same class.

For example:-

```
rectangle r1,r2,r3;
```

Objects can also be created when a class is defined by placing their names immediately after the closing brass. Eg:-

```
class rectangle
{
  - - - - - -
  - - - - - -
} r1,r2,r3;
```

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

Accessing class members:

The class members are accessed using a dot operator but this work only for public members. The dot operator is called member access operator. The general format is;

```
class_object.class_member;
```

For public data members, we can use following syntax to access them.

```
class_object.data_member;
```

Eg:-

```
obj1.data1 = obj1.data2 + obj1.data3;
```

In the above example, obj1 is an object of class and data1, data2 and data3 are its public members.

For the public member functions we can use the following syntax to access them.

Syntax:

```
class_object.member_function (argument list);
```

Eg:-

```
r.setdata(4,2);
r.displaydata();
```

```
#include<iostream.h>
#include<conio.h>
 class rectangle
 {
  private:
      int length;
       int breadth;
  public:
      void setdata (int l, int b)
      {
        length=l;
        breadth=b;
      }
       void displaydata ()
       {
        cout<<"length ="<<length << endl<<"Breadth ="
            << breadth<<endl;
       }
       int findarea()
       {
        return length * breadth;
       }
      int findperimeter()
       {
         return 2*(length +breadth);
       }
 };
```

```
int main()
{
  rectangle r;
  r.setdata(4,2);
  r.displaydata();
  cout<<"Area="<<r.findarea();
  cout<<"\nPerimeter="<<r.findperimeter();
  getch();
  return 0;
}
```

Defining the member function of the class:

Member function can be defined in two ways:-

i) Outside the class
ii) Inside the class

The code for the function body would be identical in both the cases. Irrespective of the place of definition the function perform the same task.

i) Definition of member function outside the class:

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

Syntax:

```
return_type class_name :: function_name (argument_list)
{
  // function body
}
```

The function is generally defined immediately after the class specifier. The function name is preceeded by;

a, return type of the function
b, class name to which the function belongs
c, symbol with double colon (::). This symbol is called scope resolution operator. This operator tells the compiler that the function belongs to the ~~class~~ class_name.

Example:

```
class rectangle
{
  - - - - -
  - - - - -
  public:
    void setdata (int, int);
  - - - - -
  - - - - -
};
void rectangle :: setdata (int l, int b)
{
  length = l;
  breadth = b;
}
```

In this example, the return type is void which means the function setdata does not return any value. The scope resolution operator tells that the function setdata is a member of the class rectangle. The arguments are l and b.

ii) Definition of member function inside the class:

Function body can be defined in the class itself by replacing declaration by function definition. All the member functions defined inside the class are inline by default. Hence, all the restrictions that apply to inline function will also apply here.

Example:-

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

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

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

Example:-

```
Class A
{   private:
        int a, b;
    public:
        void getdata();
};
inline void A :: getdata()
 {
   cin>> a>>b;
 }
```

Nested member functions:

A member function can be called by using its name inside another member function of same class. This is known as nesting of member functions.

Example:-

```
#include <iostream.h>
#include <conio.h>
 class Addition
 {
  private:
      int a,b,sum;
  public:
      void read();
      void show();
       int add();
 };
 void Addition :: read()
  {
    cout<<"Enter a & b"<<endl;
   cin>>a>>b;
  }
 void Addition :: show()
 {
   sum =add();
   cout<<"Addition = "<<sum;
  }
  int Addition :: add()
  {
    return (a+b);
  }
  int main()
  {
```

```
    Addition a;
    a.read();
    a.show();
    getch();
    return 0;
}
```

Private member functions:

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

Example:

```
class A
{
private:
    int a,b,sum;
    int add(); // private member function
public:
    void read();
    void show();
};
void Addition::read()
{
    cout<<"Enter a & b"<<endl;
    cin>>a>>b;
}
```

```
void Addition :: show()
{
 sum = add(); /* calling private member function */
 cout << "Addition =" << sum;
}
int Addition::add()
 {
  return (a+b);
 }
 int main()
 {
   Addition a;
   a.read();
   a.show();
    getch();
    return 0;
 }
```

If a is an object of A then the following statement a.add() is not valid. This is because add() is a private member function. Hence, cannot be called using object and dot operator.

Memory allocation for objects:

Object 1	Object 2	Object 3
data member 1	data member 1	data member 1
data member 2	data member 2	data member 2
- - - - - -	- - - - - -	- - - - - -
data member n	data member n	data member n

Method 1
Method 2
- - - -
Method k.

Fig:- Object in memory.

For each object, the memory space for data members is allocated separately because the data members will hold different data values for different objects. However, all the objects in a given class use the same member functions. Hence, the member functions are created and placed in memory only once when they are defined as a part of a class specification and no separate space is allocated for member functions when objects are created.

Pointers to Objects:

It is perfectly valid to create pointers pointing to classes. To declare a pointer pointing to any class, we use the class name as the type for the pointer.

Example:-

```
class rectangle
{
- - - - -
public:
    void getdata()
    {
      //function body
    }
};
```

Then the pointer to class rectange is declared as;

```
rectangle *rect;
```

The above statement allocates memory to keep the address of an objects of class rectangle. We should use new operator to allocate memory for the objects of rectangle class as below:

```
*rect = new rectangle();
```

While creating pointer objects, we can also combine above two statements in single as below;

```
rectangle *rect = new rectangle();
```

To refer directly to a member of an object pointed by a pointer we should use array operator.
Example;

```
rect -> getdata();
```

Arrays of Objects:

Array can be created of any datatype since a class is also a user defined data type, we can create array of objects. Such an array is called an array of objects. Accessing member data in an array of objects is a two step process. At first, identify the member of the array by using the index operation ([]) and then add the member operator (.) to access the particular member variable.

Example:-

//Program to display id and salary of different department of different no. of employees.

```
#include <iostream.h>
#include <conio.h>
 const int MAX = 100;
 class Employee
 {
     private:
        int id;
        int salary;
     public:
      void getdata()
```

```
{
 cout << " Enter the ID :";
 cin >> id;
 cout << " Enter the salary :";
 cin >> salary;
}
void putdata()
 {
  cout << " ID :" << id
  cout << "\n Salary :" << salary;
 }
};
int main()
 {
   Employee e[MAX];
    int n=0;
    char ans;
    do
    {
     cout << "Enter the Employee Number ::" << n+1 << endl;
     e[n++].getdata();
     cout << "Enter another (y/n)? :";
      cin >> ans;
   } while (ans != 'n');
   cout << endl << "***** Employee Details ***** " << endl;
   for (int i=0; i<n; i++)
   {
     cout << "\n Employee Number is ::" << i;
```

```
        e[i].putdata();
    }
    getch();
    return 0;
}
```

Output:

```
Enter the Employee Number::1
Enter the ID : 001
Enter the salary :50,000
Enter another (y/n)? y
Enter the Employee Number :: 2
Enter the ID : 002
Enter the salary : 25,000
Enter another (y/n)? n
***** Employee Details *****
Employee Number is::1  ID:001  Salary : 50,000
Employee Number is::2  ID:002  Salary: 30,000
```

Objects as function arguments:-

Like any other data type, an object may be used as a function argument in three ways:

i) Pass by value
ii) Pass by reference
iii) Pass by pointer

1) Pass by value:

If arguments are passed by value to the method, a copy of object is passed to the function. Any changes made to the object inside the function do not affect the object used in a function call.

Example:
Program to add two distance objects each having two private data members feet and inches.

```
#include<iostream.h>
#include<conio.h>
  class Distance
  {
     private:
          int feet;
          int inches;
     public:
          void getdata(int f, int i)
           {
              feet = f;
              inches = i;
           }
     void display()
     {
      cout<<"["<<feet<<","<<inches<<"]"<<endl;
     }
     void add distance (Distance d1, Distance d2)
     {
```

```
        feet = d1.feet + d2.feet;
        inches = d1.inches + d2.inches;
        feet = feet + inches/12;
        inches = inches % 12;
    }
};
int main()
{
    Distance d1,d2,d3;
    d1.getdata(5,6);
    d2.getdata(7,4);
    d3.adddistance(d1,d2);
    cout<<"d1=";
    d1.display();
    cout<<"d2=";
    d2.display();
    cout<<"d3=";
    d3.display();
    getch();
    return 0;
}
```

Output:

```
d1 = (5,6)
d2 = (7,4)
d3 = (12,10)
```

11) Pass by reference:

If arguments are passed by reference an address of the object is passed to the function. The function works directly on the actual object used in the function call. This means any changes made to the object inside the function will reflect in the actual object.

Example:
Changing the above example to pass by reference.

```
#include<iostream.h>
#include<conio.h>
  void add distance (Distance &d1, Distance &d2)
  {
    feet = d1.feet + d2.feet;
    inches = d1.inches + d2.inches;
     feet = feet + inches/12;
    inches = inches %12
  }
This function must be called as follows:-
    d3.add distance (d1, d2);
```

iii) Pass by pointer:

Like pass by reference, pass by pointer method can also be used to work directly on the actual object used in the function call.

Example:

```
void adddistance (Distance *d1, Distance *d2)
{
    feet = d1->feet + d2->feet;
   inches = d1->inches + d2->inches;
   feet = feet + inches/12;
    inches = inches % 12;
}
d3.adddistance (&d1, &d2);
```

Returning objects:-

A function not only receive objects as argument but also can return them. A function can return objects in 3 ways:

1) Return-by-value:

In this method, a copy of the object is returned to the function call.

Example:-
Program to find the sum of complex numbers and returns the objects by value.

```
#include <iostream.h>
#include <conio.h>
 class complex
 {
   private:
       int real, img;
   public:
       void getdata()
        {
          cout<<"Enter the real & imaginary value:";
          cin>>real>>img;
        }
       void displaydata()
        {
         cout<<"("<<real<<"+i"<<img<<")";
        }
       complex addcomplexnumber(complex C1)
       {
       complex sum;
       sum.real=real+C1.real;
        sum.img=img+C1.img;
         return sum;
       }
};
```

```
int main()
{
  complex c1,c2,c3;
  c1.getdata();
  c2.getdata();
  c3 = c2.addcomplexnumber(c1);
  c3.displaydata();
  getch();
  return 0;
}
```

Output:-

```
Enter the real & imaginary value:
2
4
Enter the real & imaginary value:
6
2
(8+i6)
```

ii) Return-by-reference:

In this method, an address of the object is returned to the function call.

Example;

```
complex &add complexnumber (complex &c1)
{
  complex sum;
  sum.real = (real + c.real);
  sum.img = (img + c.img);
  return sum;
```

The function must be called as:

```
C3 = C2.addcomplexnumber (C1);
```

iii) Return-by-pointer:

Like return-by-reference method, return-by-pointer also returns the address of the object to the function call.

Example:-

```
complex *addcomplexnumber (complex *C1)
{
    complex *sum;
    sum->real = (real + C1->real)
    sum->img = (img + C1->img)
    return *sum;
}
```

In the main function, C_3 must be declared as a pointer variable and the function call must be made as below:

```
complex C1, C2, *C3;
C3 = C2.addcomplexnumber (&C1);
C3->display();
```

Static Data members:

If a data member in a class is defined as static then only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created. Hence, these data members are normally used to maintain values common to the entire class and are also called class variables. Memory for static data members is allocated at the time of declaration. A static variable is declared using the static keyword.

For example:-

```
class A
{
    static int i;
    public:
    ------
};
```

Then we must defined storage for the static data member in the definition file like this; int A:: i = 1;

The definition must occur outside the class and only one definition is allowed.

Example:

```
//Program to count the number of objects.
#include <iostream.h>
#include <conio.h>
  class Student
  {
```

```
int roll;
char name[20];
static int count;
public:
  void getdata()
  {
   cout<<"Enter name & roll no. of a student:"
   cin>>name>>roll;
   count++;
  }
  void displaydata()
  {
   cout<<"count="<<count<<endl;
  }
};
int student::count=0;
int main()
{
 student S1,S2,S3;
 S1.displaydata();
 S2.displaydata();
 S3.displaydata();
 S1.getdata();
 S2.getdata();
 S3.getdata();
 S1.displaydata();
 S2.displaydata();
 S3.displaydata();
 getch();
} return 0;
```

Output:

```
count=0
count=0
count=0
Enter name & roll no. of a stude
Ram 21
Enter name & roll no. of a studen
Hari 22
Enter name & roll no. of a studen
Shyam 23
count=3
count=3
count=3
```

In this program, the static variable count is initialized to 0 when the objects are created. The count is incremented whenever data is supplied to an object.

static data member lai access garna sakxa.

Static Member Function:

In a class, functions can also be declared as static. Properties of static member functions are:

i) They can access only static members (functions and variables) declared in the same class.
ii) They can be called in the ordinary way with dot or arrow in association with an object.
iii) They can also be called using class name.

```
class x
{
  public:
     static void fun()
     {
       // function body
     }
};
int main()
{
   x:: fun(); // calling static member function
}
```

:: → resolution operator.

Example:-

```
#include <iostream.h>
#include <conio.h>
  class A
  {
     int x;
     static int count;
     public:
        void getdata()
        {
          cout<<"Enter the value of X="<<endl;
          cin>>x;
          count++;
        }
        static void display()
        {
          cout<<"Count="<<count;
        }
  };
  int A::count=0
  int main()
  {
     A a1,a2,a3;
     a1.display();
     a2.display();
     a3.display();
     a1.getdata();
     a2.getdata();
```

```
    a3.getdata();
    a1.display();
    a2.display();
    a3.display();
}   getch();
    return 0;
}.
```

Nested class :-

A nested class is a class which is declared in another enclosing class. A nested class is a member and has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class i.e. the usual access rules shall be obeyed.

Example:

```
#include <iostream.h>
#include <conio.h>
 class Employee
 {
   int eid, salary, age;
   char name[20];
   class DOB
   {
     int y, m, d;
```

```
public:
    void getDOB()
    {
     cout<<"Enter year:";
     cin>>y;
     cout<<"Enter month:";
     cin>>m;
     cout<<"Enter day:";
      cin>>d;
    }
    void displayDOB()
    {
     cout<<y<<"-"<<m<<"-"<<d;
    }
};
 DOB db;
Public:
  void getdata()
  { cout<<"Enter eid, salary, age & name:";
    cin>>eid>>salary>>age>>name;
    cout<<"Enter DOB:";
    db.getDOB();
  }
   void displaydata()
      cout<<"Employee information:";
      cout<<"Eid="<<eid<<"salary="<<salary
          <<"Age="<<age<<"name="<<name;
      cout<<"DOB=";
```

```
    db.display_DOB();
  }
};
int main()
{
  Employee e;
  e.getdata();
  e.display_data1();
  getch();
  return 0;
}
```

kunai function vitra kunai class declare kunxa vane tesko class lai local class vanincha.

Local class:-

A class that is declared within a function is called local class. In practice, we do not use local class. Declarations in a local class can only use type names, enumerations, static variables from the enclosing scope as well as external variables and functions.

Syntax:

```
void fun()
{
  class A
  {
  };
}
```

Example:-

```
#include <iostream.h>
#include <conio.h>
 void studentList();
 int main()
  {
     studentList();
     getch();
     return 0;
  }
  void studentList()
  {
     class student
      {
        int roll;
        char name[20];
        public:
          void getdata()
           {
           cout<<"Enter roll & name:";
            cin>>roll>>name;
           }
          void displaydata()
          {
           cout<<"Roll no ="<<roll<<endl;
           cout<<"Name ="<<name<<endl;
          }
  };
```

```
    student s1;
    s1.getdata();
    s1.displaydata();
}
```

Constant objects and constant member function:

Some objects need to be modified and some do not. We use the keyword const to specify that an object is not modifiable and that any attempt to modify the object should result compiler error. A const object can only invoke a const member function.

If a member function does not alter any data in the class then we may declare and define it as a constant member function.

Example:-

```
// Program to illustrate constant object and constant
member function.
#include <iostream.h>
#include <conio.h>
class Test
{
  public:
      void normalfun();
      void constantfun() const;
};
void Test :: normalfun()
```

```
{
 cout<<"I am a normal function"<<endl;
}
void Test::constantfun() const
{
  cout<<"I am a constant function"<<endl;
}
int main()
{
  Test normalobj;
  normalobj.normalfun();
const Test constantobj;
   constantobj.constantfun();
   getch();
   return 0;
}
```

Friend function:-

A function is said to be a friend function of a class if it can access the members (including private members) of the class even if it is not member function of this class. In otherwords, a friend function is a non-member function that has access to the private members of the class. Some special characteristics of friend functions are:

i) Since it is not in the scope of the class to which it has been declared as a friend, it cannot be called using the

object of that class.

ii) It can be invoked like a normal function without the help of any objects.

iii) Like member functions, it cannot access the member name directly. So it has to use an object name and dot operator with each member name.

iv) It can be declared either in the public, protected or in the private part of a class without affecting its meaning.

v) Usually it takes objects as argument.

Example:-

```
// Program to illustrate friend function.
#include<iostream.h>
#include<conio.h>
   class Beta;
   class Alpha
   {
    private:
        int data;
    public:
        void setdata (int d)
        {
          data =d;
        }
        friend int sum (Alpha, Beta);
   };
   class Beta
   {
```

```
private:
    int data;
public:
    void setdata (int d)
    {
      data=d;
    }
    friend int sum (Alpha ,Beta);
};
int sum (Alpha a, Beta b)
{
   return (a.data + b.data);
}
int main()
{
   Alpha a;
   a.setdata (7);
   Beta b;
   b.setdata (3);
   cout << "The sum=" << sum (a,b);
   getch();
   return 0;
}
```

Output:

The sum =10

Friend class :-

A class can also be declared to be the friend of some other class. When we create a friend class then all the member functions of the friend class also become the member of the other class. This requires the condition that the friend becoming class must be first declared or defined.

Example :-

```
#include <iostream.h>
#include <conio.h>
 class Alpha
  {
   private:
      int data;
   public:
      void setdata(int d)
       {
        data=d;
       }
   friend class Beta;
};
class Beta
{
  public:
     void displaydata(Alpha a)
     {
       cout<<"Data="<<a.data<<endl;
     }
```

```
int main()
 {
   Alpha a;
   a.setdata(100);
   Beta b;
   b.displaydata(a);
   getch();
   return 0;
 }.
```

object to memory address, name conflict hatauna

It gives the address of the object.

This pointer:-

Every object in C++ has access to its own address through a pointer called This pointer. The pointer 'This' acts as an implicit argument to all the member functions. Therefore, inside a member function This may be used to refer to the invoking object. We can use This pointer for following two purposes:

i) To resolve the name conflict between member variable and local variables in a method.

ii) To return the invoking object.

Example:

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
  class Person
   {
```

```
string name;
int age;
public:
    void getdata (string name, int age)
    {
      this -> name = name;
      this -> age = age;
    }
    void displaydata()
    {
     cout << "Name:" << name; {or, this -> name;}
     cout << "Age:" << age; {or, << this -> age;}
    }
    person isElder (person p)
    {
        if(age < p.age)
          {
            return *this;
          }
        else
            {
              return p;
            }
};      }
        int main()
        {
          person P1, P2, P3;
          P1.getdata ("Ram", 25);
```

```
P2.getdata ("Hari", 27);
P3 = P2.isElder (P1);
cout<<"Elder is:"<<endl;
P3.displaydata();
getch();
return 0;
}
```

object create ^ automatic call hune function
- defined in public area
- yesko name class name sanga equal hunxa

V.V.V.V. Imp.

Constructors:-

A constructor is a special member function that is executed automatically whenever an object is created. It is used for automatic initialization. Automatic initialization is the process of initializing objects data member when it is created, without making a separate call to a member function. The name of the constructor is same as the class name.

Example:-

```
class rectangle
{
  private:
     int length;
     int breadth;
  public:
     rectangle()      // constructor.
     {
       length=0;
```

```
    breadth=0;
  }
  ------
  ------
  ------
};
```

Constructors have some special characteristics, these are:-

1. Constructor should be defined or declared in the public section.
2. They do not have return type.
3. They cannot be inherited but a derived class can call the base class constructor.
4. Like function, they can have default arguments.
5. Constructors can not be virtual.
6. We cannot refer to their addresses.

Types of constructors :-

There are basically three types of constructors.

1. Default constructor :- (-no parameter

A constructor that doesnot take any parameter is called default constructor. There are 3 possible situations for this.

a, If we do not provide any constructor with a class, the compiler provided one would be the default constructor. and it does not do anything other than allocating memory for the class object.

Example:-

```
class A
{
  // no constructor
}
```

b) If we provide constructor without any arguments then that is called default constructor.

Example:-

```
class A
{
  public
    A()
    {//body
    }
  OR
  or
    A(void)
    {
      //body
    }
};
```

c) If we provide constructor with all default arguments then that can also be considered as the default constructor.

Example:-

```
Class A
  {
```

```
public:
    A(int X=10)
    {
      // body
    }
};
```

2. Parameterized constructor :-

A constructor that takes arguments is called a parameterized constructor. Arguments are passed when the objects are created. This can be done in two ways:

a) By calling the constructor explicitly
b) By calling the constructor implicitly

Example:-

```
class A
{
  int val;
  public:
    A(int x)
    {
      val = x;
    }
};
void main()
{
  A a1(100);    // implicit call
  A a2 = A(200); // explicit call
}
```

3. Copy constructor:-

A copy constructor allows an object to be initialized with another object of the same class. It implies that the values stored in data members of an existing object can be copied into the data variables of the object being constructed, provided the objects belong to the same class. A copy constructor has a single parameter of reference type that refer to the class itself as shown below:

```
Class Test
{
    int a, b;
    Public:
        Test ( Test &t)
        {
            a = t.a;
            b = t.b;
        }
        - - - - -
        - - - - -
};
```

Once a copy constructor is defined, we can use copy constructor as below:

```
Test t2(t1);
```

It creates new t_2 object and performs member

by member copy of t_1 into t_2. Another form of this statement is;

Test $t_2 = t_1$;

Constructor overloading:-

The process of sharing the same name by two or more functions is referred to as function overloading. similarly, when more than one constructor is defined in a class then it is called contructor overloading. We can define more than one constructor in a class either with different number of arguments or with different types of argument. It is used to achieve polymorphism.

Example:

```
// Program to illustrate constructor overloading.
#include <iostream.h>
#include <conio.h>
  class Test
  {
      int a,b;
     public:
        Test()
        {
          a=0;
          b=0;
        }
        Test (int x)
        {
          a=b=x;
        }
```

```
Test (int x, int y)
{
  a=x;
  b=y;
}
void display()
{
  cout<<"a="<<a<<endl;
  cout<<"b="<<b<<endl;
}
};
int main()
{
 Test t1;
 cout<<"The value of object one:"<<endl;
 t1.display();
 Test t2(100);
 cout<<"The value of object two:"<<endl;
 t2.display();
 Test t3(100,200);
 cout<<"The value of object three:"<<endl;
 t3.display();
 getch();
 return 0;
}
```

Output:

```
The value of object one:
a=0
b=0
The value of object two:
a=100
b=100
The value of object three:
a=100
b=200
```

Constructor with default argument :-

Like function, constructors can also have default argument.

Example:-

Consider a program that calculates simple interest. It declares a class interest, representing principal, rate and year. The constructor function initializes the object with principal and no. of year. If rate of interest is not passed as an argument to it, the simple interest is calculating taking the default value of rate of interest.

```
#include <iostream.h>
#include <conio.h>
  class SimpleInterest
  {
    int principal, time, rate;
    float amount;
    Public:
      SimpleInterest (int p, int t, int r=10)
      {
        principal = p;
        time = t;
        rate = r;
      }
      void calculateInterest()
      {
        cout << "Principal = " << Principal << endl;
```

```
    cout<<"Time="<<time<<endl;
    cout<<"Rate="<<rate<<endl;
    amount = amount + (principal * time * rate)/100;
    cout<<"Amount="<<amount<<endl;
}
int main()
{
 SimpleInterest S1(10000,2);
 S1.calculateInterest();
 SimpleInterest S2(10000,2,15);
 S2.calculateInterest();
 getch();
 return 0;
}
```

- dynamically memory allocate garxa.

Dynamic Constructor:-

Allocation of memory to object at the time of their construction is known as dynamic constructor of objects. In the dynamic constructor new operator is used for allocation of memory.

Example:-

```
#include <iostream.h>
#include <conio.h>
  class Dynamic
   {
     int a, *ptr;
     public:
```

```
Dynamic()
{
 a=0;
 ptr= new int;
 *ptr =a;
}
Dynamic (int x)
{
    a =x;
   ptr = new int ;
   *ptr = x;
  }
   void display()
   {
    cout<<" The value of a="<<a<<endl;
    cout<<" The value of *ptr="<<*ptr<<endl;
   }
};
int main()
{
 Dynamic d1, d2(100);
 cout<<"The value for object one:"<<endl;
  d1.display();
 cout<<" The value of object two:"<<endl;
  d2.display();
  getch();
  return 0;
}.
```

Destructors :-

A destructor is a special member function that is executed automatically just before lifetime of an object is finished. The most common use of destructors is to destroy the memory that was allocated for the object by the constructor. A destructor has the following characteristics:

- A destructor has the same name as the class but is preceeded by a tilde (~) sign.
- Like constructors, destructors do not have a return type.
- Destructors take no arguments. Hence, we can use only one destructor in a class.
- Destructor can be virtual.
- Destructor cannot be inherited.

Example :-

```
// Program to illustrate destructor.
#include <iostream.h>
#include <conio.h>
  class Test
  {
    static int count;
     Public:
       Test()
       {
         count ++;
         cout<< count << " Object is created " << endl;
       }
```

```
~Test()
  {
  cout << count << " Object is destroyed" << endl;
   count--;
  }
};
int Test:: count;
 int main()
  {
   cout << "Main Block" << endl;
    Test t1;
    {
      cout << "Block one" << endl;
      Test t2;
       cout << "Block one completed " << endl;
    }
    cout << "Main block completed" << endl;
    getch();
    return 0;
  }
```

Output:

```
Main Block
1 object is created
Block one
2 object is created
Block one completed
2 object is destroyed
Main block completed
1 object is destroyed.
```

object ko member lai operate garna sakne capability.

3.2) Operator overloading :-

Operator overloading is one of the most exciting features of C++. The term operator overloading refers to giving the normal C++ operators additional meaning and semantics so that we can use them with user defined data types.

Example:

```
int a, b, c;
c = a+b;    // add two integers a and b.
```

But if a and b are objects then c = a+b is wrong. To add two operands i.e. a+b we have to overload operator '+'.

Restrictions on operator overloading:

We can overload all the C++ operators except the following :-

1. Class member access or dot operator (.)
2. Scope resolution operator (::)
3. Size operator (sizeof)
4. Conditional operator (?:)

In case of friend function, we cannot overload these operators.

1) = assignment operator
2) () function call operator

WINNER
Page No.:
Date: / /

3) [] subscripting operator
4) → class member access/ arrow operator.

Operator functions:-

Operator overloading is done with the help of a special function called operator function. The general form of an operator function is:

```
return-type operator op (argument list)
{
  // function body
}
```

where;
→ return_type is the type of returned value by the operation
→ operator is the keyword and
→ op is the operator symbol.

Operator function must be either member function or friend function. A basic difference between them is that a friend function will have only one argument for unary fun operators and two for binary operators. While a member function has no argument for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend function.

Rules for operator overloading:-

1. Operators already predefined in the C++ compiler can be only overloaded.
2. Operator overload cannot change operator templetes i.e. for example; the increment operator '++' is used only as unary operator. It cannot be used as binary operator.
3. Overloading an operator does not change its basic meaning i.e. '+' operator should not be re-defined to subtract one value from another.
4. Unary operators overloaded by means of a member function take no explicit argument and return no explicit values but those overloaded by means of a friend function take one reference argument.
5. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function takes two explicit.
6. Overloaded operators must either be a non-static class member function or global function.
7. A global function that needs access to private or protected class members must be declared as a friend of that class. A global function must take at least one argument.

Overloading unary operators:-

Unary operators are those operators that act on a single operand such as ++, --.

Overloading pre-increment operator:

Let us consider a increment operator ++. This operator increases the value of operand by 1 when applied to a basic data item. We can overload this operator so that it can be used to increase the value of each data members when applied to an object.

Example:-

```
#include<iostream.h>
#include<conio.h>
  class rectangle
  {
    private:
        int length;
        int breadth;
    Public:
        rectang(int l, int b)
        {
          length=l;
          breadth=b;
        }
        void operator ++()
        {
```

```
        ++length;
        ++breadth;
      }
      void display()
      {
       cout<<"Length="<<length<<endl;
       cout<<"Breadth="<<breadth<<endl;
      }
};
int main()
{
  rectangle r(5,1);
  ++r;          // r.operator++();
  r.display();
  getch();
  return 0;
}
```

Output:

Length = 6

Breadth = 2.

Overloading post-increment operator:

To overload the post increment operator, we need to use int as dummy argument to the operator function. That int is not really an argument and it does not mean integer. It is simply a signal to the compiler to create the post-fix version of the operator.

Example:-

```
void operator ++(int)
{
    length ++;
    breadth ++;
}
```

We can call this operator function as follows:

r++.

Overloading unary operator using friend function:

When overloading unary operator by using friend function, it must take an object as argument.

Example:-

```
#include <iostream.h>
#include <conio.h>
class rectangle
{
  private:
```

```
    int length;
    int breadth;
public:
    rectangle (int l, int b)
    {
      length = l;
      breadth = b;
    }
    friend void operator ++ (rectangle &);
    void display ()
    {
     cout<<"Length ="<<length<<endl;
     cout<<"Breadth ="<<breadth<<endl;
    }
};
void operator ++ (rectangle &r)
 {
   ++r.length;
   ++r.breadth;
 }
  int main()
  {
   rectangle r1 (5,1);
   ++r1;          // operator ++ (r1)
   r1.display()
   getch();
    return 0;
  }
```

Overloading negation operator:

Minus operator acts as unary operator which makes value of any value negative when put behind the value. We can overload negation operator to make value of object negative

Example:-

```
#include<iostream.h>
#include<conio.h>
class Test
{
  int x,y;
  public:
  void getdata()
  {
  cout<<"Enter the value of x & y:"<<endl;
   cin>>x>>y;
  }
  void display()
  {
   cout<<"x="<<x<<endl;
   cout<<"y="<<y<<endl;
  }
  Test operator -()
  {
    Test t;
    t.x = -x;
    t.y = -y;
  } return t;
```

```
int main()
{
Test p,q;
p.getdata();
q=p;
q.display();
getch();
return 0;
}
```

Overloading binary operators :-

Binary operators are operators which require two operands to perform the operation. When they are overloaded by means of member function, the function takes one argument whereas it takes two arguments in case of friend function.

Overloading plus operator:-

This operator adds the values of two operands when applied to a basic data item. This operator can be overloaded to add the values of corresponding data members when applied to two data objects.

Example:-

```
// overloading plus operator
#include <iostream.h>
#include <conio.h>
```

```
class distance
{
  int feet, inches;
    public:
       distance (int f, int i)
       {
        feet = f;  inches = i;
       }
       void display ()
       {
       cout <<"(" << feet <<"}, " << inches << ")" <<endl;
       }
        distance operator +(distance d)
        {
          distance d3;
          d3.feet = feet + d.feet;
          d3.inches = inches + d.inches;
          d3.feet = d3.feet + d3.inches/12;
          d3.inches = d3.inches %.12;
           return d3;
          }
  };
   int main()
    {
     distance d1(5,11), d2(3,6), d3;
       d3 = d1 + d2;   // d1.operator+(d2);
       cout << " d1 =";
       d1.display();
```

```
    cout<<" d2 = ";
    d2.display();
    cout<<" d3 = ";
    d3.display();
    getch();
    return 0;
}
```

Output:-

$d_1 = (5, 11)$
$d_2 = (3, 6)$
$d_3 =$ [illegible]

Overloading plus operator using friend function:

Here, we need to use two arguments in operator function because function is called without using object and hence no object is passed as implicitly to the function.

Example:-

```
// overloading plus operator using friend function.
#include <iostream.h>
#include <conio.h>
class distance
{
    int feet, inches;
```

```
public:
    distance (int f, int i)
    {
      feet = f ; inches = i;
    }
     void display()
     {
     cout<< "(" << feet << "," << inches << ")" << endl;
     }
     friend distance operator +(distance, distance);
};
distance operator +(distance x, distance y)
 {
   distance d;
   d.feet = x.feet + y.feet;
   d.inches = x.inches + y.inches;
   d.feet = d.feet + d.inches /12;
   d.inches = d.inches %.12;
    return d;
  }
 int main()
  {
   distance d1(5,11), d2(4,6), d3;
     d3 = d1+d2;
     cout << "d1=";
     d1.display();
     cout << "d2=";
     d2.display();
```

```
    cout<<" d3=";
    d3.display();
    getch();
    return 0;
}
```

Overloading plus operator to concatenate two strings:

We can overload the plus operate to concatenate two strings.

Example:

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
  class string
  {
    char *s;
    int l;
    public:
        void getdata()
        {
          char str[20];
          cout<<"Enter your name:";
          cin>>str;
```

```
    l = strlen (str);
    s = new char (l+1);
    strcpy (s, str);
  }
  void display ()
  {
    cout << s << endl;
  }
    string operator + (string x)
    {
      string temp;
      temp.s = new char (l+x.l+1);
      strcpy (temp.s, s);
      strcat (temp.s, x.s);
      return temp;
    }
};
int main ()
{
  string s1, s2, s3;
  s1.getdata ();
  s2.getdata ();
  s3 = s1 + s2;
  cout << "s3 =";
  s3.display ();
  getch ();
  return 0;
}
```

Output:

```
Enter your name: Sabita
Enter your name: Dang;
s3 = Sabita Dang;
```

Overloading stream insertion and stream Extraction operator:-

C++ is able to input and output the built-in data types using the stream extraction operator (>>) and stream insertion operator(<<). The stream insertion and stream extraction operator also can be overloaded to perform input and output for user defined types like an object.

Here, it is important to make operator overloading function a friend of the class because it could be called without creating an object.

Example:

```
#include <iostream.h>
#include <conio.h>
 class distance
 {
  private:
      int feet;
      int inches;
  public:
      distance (int f , int i)
      {
        feet = f; inches = i;
      }
      friend ostream &operator <<(ostream &output,
        distance &d);
      friend istream &operator >>(istream &input,
                                  distance &d);
};
```

```
ostream &operator <<(ostream &output, distance &d)
{
 output << "F=" << d.feet <<endl<< "I=" << d.inches<<endl;
 return output;
}
istream &operator >>(istream &input, distance &d)
{
 input >> d.feet >> d.inches;
 return input;
}
int main()
{
 distance d1(11,10), d2(5,11), d3(0,0)
 cout << "Enter the value of object:" << endl;
 cin >> d3
 cout << "First distance =" << d1 <<endl;
 cout << "second distance= " << d2 << endl;
 cout << "Third distance =" << d3 <<endl;
 getch();
 return 0;
}
```

Type conversion:

It is the process of converting one type into another. In other words, converting expression of a given type into another is called type-casting (conversion). A type conversion may either be explicit or implicit.

Implicit type conversion is done by the compiler automatically. If the data types are user-defined, the compiler does not support automatic type conversion and therefore we must design the conversion routines by ourselves. Three types of situations might arise in the data conversion:

i) Conversion from basic type to class type
ii) Conversion from class type to basic type
iii) Conversion from one class type to another class type.

1) Conversion from basic type to class type:-

To convert basic type to object, it is necessary to use the constructor. The constructor in this case takes single argument whose type is to be converted.

Example:-

```
// Basic to object type
#include<iostream.h>
#include<conio.h>
  class Distance
```

```
{
private:
    int feet;
    int inches;
public:
    Distance (int f, int i)
    {
     feet = f;
     inches = i;
    }
    Distance (float f)
    {
     feet = (int) f;
     inches = (f - feet) * 12;
    }
    void display()
    {
     cout << " feet = " << feet << endl;
     cout << " inches = " << inches << endl;
    }
};
int main()
 {
   Distance d1;
   float f = 2.5;
Distance  d1 = f;        // d1.Distance(f)
   d1.display();
   getch();
   return 0;
```

11) Conversion from class type to basic type:-

To convert class type to basic type, it is necessary to overload casting operator. The cast operator does not have return type. Its implicit return type is the type to which object need to be converted.

Example:-

```
// from object to basic type
#include <iostream.h>
#include <conio.h>
  class distance
  {
   private:
      int feet;
      int inches;
   public:
      distance (int f, int i)
      {
       feet = f;
       inches = i;
      }
(keyword) ← operator float ()
      {
       float a = feet + inches / 12.0;
       return a;
      }
};
```

```
int main()
{
  distance d1(5,6);
  float x = d1;      // float x = float(d1);
  cout << "x=" << x;
  getch();
  return 0;
}
```

iii) Conversion from one class type to another class type:-

This type of conversion can be carried out either by a constructor or an operator function. It depends upon where we want the routine to be located - in the source class or in the destination class.

a) Function in the source class:

If we want to convert object of one class to object of another class, it is necessary that the operator function be placed in the source class.

Example:-

// operator function in the source class.

```
#include <iostream.h>
#include <conio.h>
class Item
{
```

```
private:
    int a, b;
public:
    void getdata (int x, int y)
    {
      a=x; b=y;
    }
    void displaydata()
    {
     cout<<"x="<<a<<endl;
     cout<<"y="<<b<<endl;
    }
};
class product
{
 private:
    int m, n;
 public:
    product (int x, int y)
    {
      m=x; n=y;
    }
    operator Item()
    {
      Item i;
      i.getdata(m,n);
      return i;
    }
};
```

source ma operator use hunxa
destination ma constructor use hunxa


```
int main()
{
 product P(4,5);
 Item i;
 i=P;
 i.displaydata();
 getch();
 return 0;
}
```

Output :-
a=4
b=5

b) Function in the destination class:

In this case, it is necessary that the constructor be placed in the destination class. This constructor is a single argument constructor and serves as an instruction for converting an argument type to the class type of which it is a member.

Example:-
// method in the destination class

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

```
{
 private:
     int m,n;
 public:
     product(int x, int y)
     {
      m=x; x=y;
     }
     int getM()
     {
      return(m);
     }
     int getN()
     {
       return(n);
     }
};
Class Item
{
 private:
     int a,b;
 public:
     Item() {}
     Item (Product P)
     {
      a=p.getM();
      b=p.getN();
     }
```

```
    void displaydata()
    {
      cout<<" a ="<<a<<endl;
      cout<<" b="<<b<<endl;
     }
};
int main()
 {
  Product P(45);
   Item i;
   i=P;
   i.displaydata();
    getch();
    return 0;
 }
```

O/P :-

a = 4

b = 5

INHERITANCE

3.3 Inheritance

Inheritance is the process of creating new classes called derived classes from existing classes called base classes. The derived class inherits all the properties from the base class and can add its own properties as well. The inherited properties may be hidden if private in the base class or visible if public or protected in the base class to the derived class. Inheritance uses the concept of code reusability which saves time, money and increases program's reliability. A programmer can use a class created by another person or company.

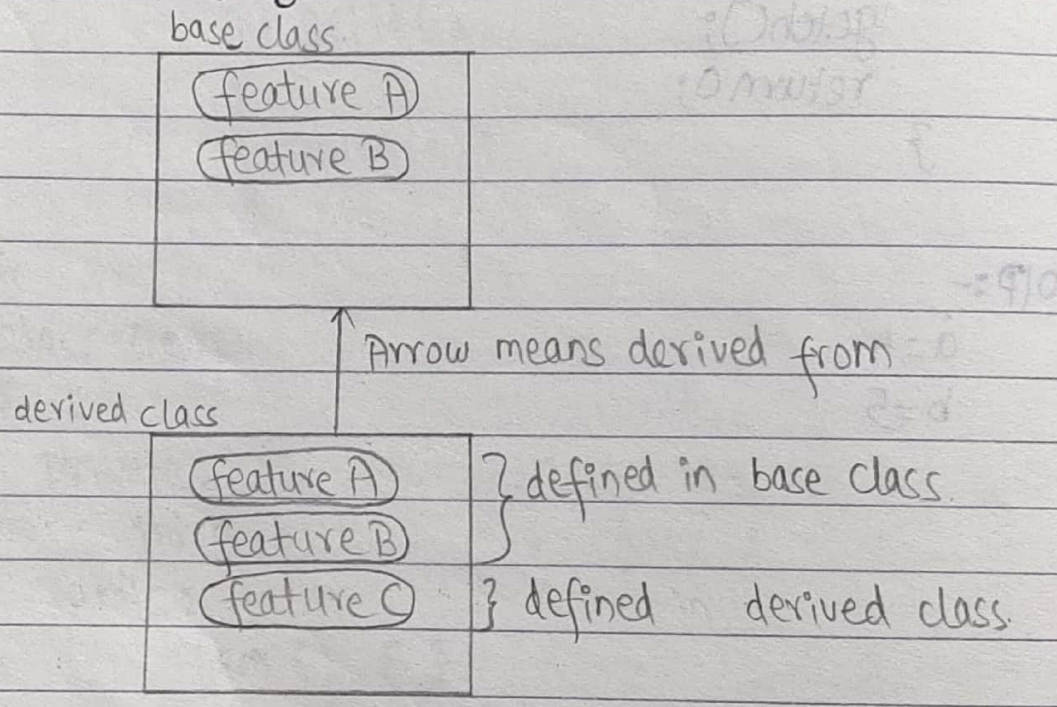


Fig:- Inheritance

Defining derived class :-

The general syntax of defining derived class is as follows:-

```
class derivedclassname : access_specifier baseclassname
{
  // members of derived class
};
```

Example:-

```
class A
{
  // member of class A
};
class B: public A
{
  // member of class B
};
```

Forms of inheritance (Types):-

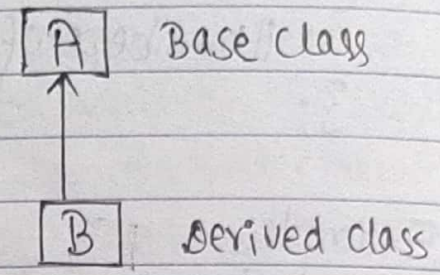
A class can inherit properties from one or more classes and from one or more labels. On the basis of this concept, there are 5 forms of inheritance:

1) Single inheritance
2) Multiple inheritance
3) Hierarchical inheritance
4) Multi-level inheritance
5) Hybrid inheritance

1) Single inheritance:-

In single inheritance, a class is derived from only one base class. The general form and figure show this inheritance.

```
class A
{
  // body of class A
};
class B: public A
{
  // body of class B
};
```



[Fig:- Single inheritance]

Example:-

```
// single inheritance
#include <iostream.h>
#include <conio.h>
class fruit
{
public:
    void displayfruit()
    {
      cout<<"This is fruit class"<<endl;
    }
};
class apple: public fruit
{
```

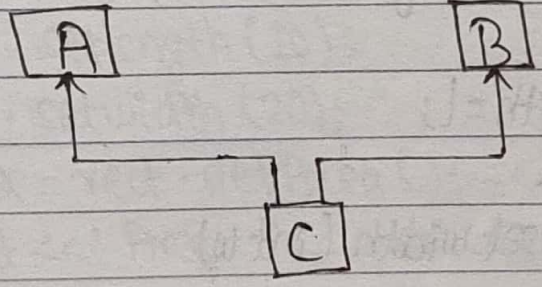
```
public:
    void display apple()
    {
     cout<< "This is an apple class"<<endl;
    }
};
 int main()
  {
    apple a;
    a.display fruit();
    a. display apple();
     getch();
     return 0;
  }
```

O/P:

This is fruit class
This is apple class

2) **Multiple inheritance :-**

In this inheritance, a class is derived from more than one base class. The general form and figure below show this inheritance.



[Fig:- Multiple inheritance]

```
class A
{
  // body of class A
};
 class B
{
  // body of class B
};
class C : public A, public B
{
  // body of class C
};
```

Example:-

```
// Multiple inheritance
#include <iostream.h>
#include <conio.h>
 class shape
 {
  protected:
      int length;
      int width;
  public:
      void setlength (int l)
      {
        length=l;
      }
      void setwidth (int w)
```

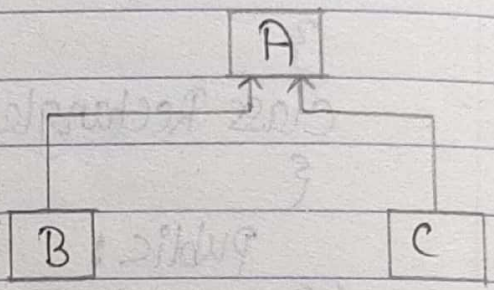
```
    {
      width = w;
    }
};
class percost
{
  public:
      int getcost( int area)
      {
        return (area *70);
      }
};
class Rectangle : public shape, public percost
{
  public:
      int getArea()
      {
        return (length * width);
      }
};
int main()
{
  Rectangle rect;
  int area;
  rect.setlength(10);
  rect.setwidth(20);
  area = rect.getArea();
  cout << "Area = " << area << endl;
```

```
    cout<< "Cost =" <<rect.getcost (area)<<endl;
    getch();
    return 0;
}
```

3) Hierarchical inheritance:-

In this type of inheritance, two or more classes inherits the properties of one base class. The general form and figure below show this inheritance.

```
Class A
{
  // body of class A
};
Class B: public A
{
  // body of class B
};
Class C: public A
{
  // body of class C
};
```



[Fig:- Hierarchical inheritance]

Example:

```
// Hierarchical inheritance
#include <iostream.h>
#include <conio.h>
  class person
```

```
{
 char name[20];
 long int phono;
public:
 void getPerson()
  {
   cout<<"\n Enter Name and Phone:";
   cin>>name>>phono;
  }
   void show_Person()
   {
    cout<<"\n Name:"<<name;
    cout<<"\n Phone:"<<phone;
   }
};
class student: public person
 {
  int roll no;
  char course[20];
  public:
    void getstd()
    {
     getPerson();
     cout<<"\n Enter Roll no:";
     cin>>roll no;
     cout<<"\n Enter course:";
      cin>>course;
    }
```

```
void showstd()
{
  showPerson();
  cout<<"\n Rollno:"<<rollno;
  cout<<"\n Course:<<course;
}
};
class teacher: public person
{
  char dep_name[20];
  char qua[10];
  public:
    void getTeacher()
    {
      getPerson();
      cout<<"\n Enter department:";
      cin>>dep_name;
      cout<<"\n Enter qualification:";
      cin>>qua;
    }
    void showTeacher()
    {
      showPerson();
      cout<<"\n Department:<<dep_name;
      cout<<"\n Qualification:<<qua;
    }
};
```

```
int main()
{
  student s;
  s.getstd();
  s.showstd();
  teacher t;
  t.getTeacher();
  t.showTeacher();
  getch();
  return 0;
}
```

or or defined type
objects - variable
class - type (user defined)
:: - resolution operator

local variable → parameter
variable ko value → argument.

return 0 → successful termination

4) Multilevel Inheritance

The mechanism of deriving a class from another derived class is known as multilevel inheritance. The process can be extended to an arbitary number of levels. The general form and figure below shows inheritance.

```
class A
{
  //body of class A
}
class B : public A
{
   //body of class B
}
class C : public B
{
   //body of class C
}
```

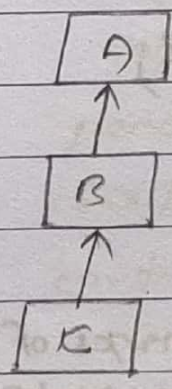


fig: multilevel inheritance

Example :

#

#

```
class student
{
    int rollno;
    char name[20];
    public:
        void getstd()
        {
            cout<<"\n Enter rollno. and Name :";
            cin>>rollno>>name;
        }
        void showstd()
        {
            cout<<"\n Roll no. : "<<rollno;
            cout<<"\n Name : "<<name;
        }
};
class marks : public student
{
    int sub1, sub2, sub3;
    public:
        void getmarks()
        {   getstd();
            cout<<"\n Enter marks of three subjects :";
            cin>>sub1>>sub2>>sub3;
        }
```

```
    void showmarks()
    {
        cout<<"\n subject1 : "<<sub1;
        cout<<"\n subject2 : "<<sub2;
        cout<<"\n subject3 : "<<sub3;
    }
    int Total_Marks()
    {
        return (sub1+sub2+sub3);
    }
};
class Result : public marks
{
    int total, percentage;
    public:
      void getdata()
      {
          getmarks();
      }
      void showResult()
      {   showmarks();
          total= TotalMarks();
          percentage = total/3;
          cout<<"\n Total marks = "<<total;
          cout<<"\n Percentage = "<<percentage;
      }
};
```

```
int main()
{
    Result r;
    r.getdata();
    r.showResult();
    getch();
    return 0;
}
```

O/P

```
Enter roll no. and Name
Prakriti 12
Enter marks of three subjects
72 73 77
subject1 = 72
subject2 = 73
subject3 = 77
Total marks = 222
Percentage = 74
```

5) Hybrid Inheritance

This type of inheritance includes more than one type of inheritance mentioned previously. The general form and figure below shows this inheritance.

```
class A
{
    //body of class A
};
class B: public A
{
    //body of class B
};
class C: public A
{
    //body of class C
};
```

```
class D : public B, public C
{
    //body of class D
};
```

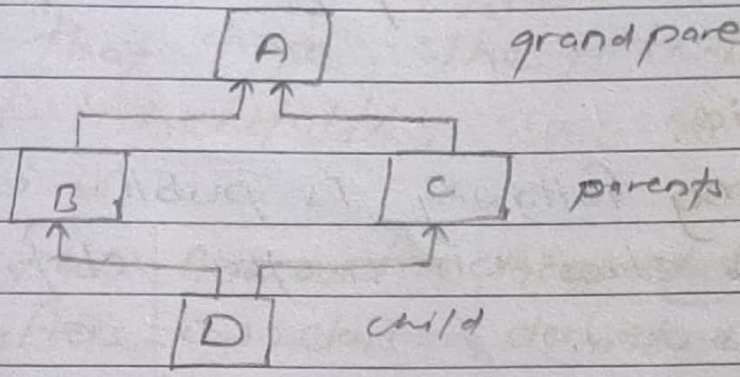


Fig: Hybrid inheritance

Access Specifiers

Data hiding is one of the important features of OOP which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the use of keywords public, protected and private. These keywords are called access specifiers.

1> Public:

The most open level of data hiding is public. public means all members declarations that follow are available to everyone. The syntax for public is.

Public:

Everything following is public until the end of the class on another data hiding keyword is used.

11> Protected:

A protected member can be accessed from within the class where it lies and from any class derived from this class. It cannot be accessed from outside of these classes. Thus a protected member achieves data hiding as well as allows data members to be access from derived class.

iii) Private :-

Private is the highest level of data hiding. The private keyword means that no one can access that member except the creator of the type inside function member of that type. The private members are not inheritable.

Access specifiers	Accessed from own class	Accessed from derived class	Accessed from objects outside class
public	Yes	Yes	Yes
Protected	Yes	Yes	No
private	Yes	No	No

Derived class constructor

constructors play an important role in initializing objects. As long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contents a constructor with one or more arguments then it is mandatory for the derived class to pass argument to the base class constructor. When

the both derived class and base classes contains constructor, the base class constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance the base classes are constructed in the order in which they appear in the derivation of the derived class. Similarly in multilevel inheritance the constructor will be executed in the order of inheritance.

Since the derived class takes the responsibility of supply initial values to its base classes, we supply the initial values that are required by all the classes together when a derived class object is declared. The constructor of derived class receives the entire list of values as its arguments and pass them on the base constructors in the order in which they are declared in the derived class.

Example:

```
// derived class constructor
#
#
class A
{
    int a;
    public:
        A(int x)
        {
            a=x;
            cout<<"\n Class A constructor initialized";
        }
        void display()
        {
            cout<<"\n a="<<a;
        }
};
class B
{
    int b;
    Public:
        B(int x)
        {
            b=x;
            cout<<"\n Class B constructor initialized";
        }
```

```
void dispb()
{
    cout<<"\n b = "<<b;
}
};
class c: public B, public A
{
    int c;
    public:
    c(int x, int y, int z): B(x), A(y)
    {
        c=z;
        cout<<"\n Class c constructor initialized";
    }
    void dispc()
    {
        cout<<"\n c = "<<c;
    }
};
int main()
{ c c(10,20,30);
  c.dispa();
  c.dispb();
  c.dispc();
  getch();
  return 0;
}
```

O/P.

Class B constructor initialized
" A " "
" C " "
a=20
b=10
c=30

object create constructor
end of object validity

Destructor under inheritance

Destructors for a base class are automatically called in the reverse order of constructor. The derived class destructor is executed first and then the destructor in the base class is executed. In the case of multiple inheritance, the derived class destructors are executed in the order in which they appear in the defination of the derived class. Similarly, in the multilevel inheritance, the destructor is executed in the order of inheritance.

Example:

```
#include<iostream.h>
#include<conio.h>
class A
{
   public:
      ~A()
      {
         cout<<" class A destructor"<<endl;
      }
};
class B : public A
{
```

```
    public:
        ~B()
        {
            cout<<" class B destructor "<<endl;
        }
};
class C : public B
{
    public:
        ~C()
        {
            cout<<"Class C destructor "<<endl;
        }
};
int main()
{
    {
        C c1;
    } //destructor is called at this point.
    getch();
    return 0;
}
```

Output

```
Class C destructor
class B destructor
class A destructor
```

base class and derived class

same name or same argument in parent and child

Overriding Member function

The process of defining the function in the derived class having same name and signature as that of base class is called function overriding. If we call the overridden function by using the object of derived class then method version define in derived class is invoked. We can call the version of method derived from base class as below:

```
obj.basename::methodname
```

Example

```
//method overriding
#
#
class A
{
    public:
        void show()
        {
            cout<<"\n This is class A";
        }
};
class B : public A
{
```

```
public:
    void show()
    {
        cout<<"\n This is class B";
    }
};
int main()
{
    B b;
    b.show(); //call to member of B
    b.A::show(); //call to member of A
    getch();
    return 0;
}
```

Output

This is class B

This is class A

Ambiguities in Inheritance

Ambiguity in inheritance arise in two situations first one is in case of multiple inheritance and second one is hybrid multipath inheritance.

1) Ambiguity in multiple inheritance:-

Suppose two base classes have an exactly similar member and a class derived from both of these class has not this member. Then, if we try to access this member from the object of the derived class, it will be ambigeous. We can remove this ambiguity by using the syntax:

```
obj.classname::methodname
```

Example

```
// ambiguity removal in multiple inheritance
#
#
class A
{
    public:
        void show()
```

```
        {
            cout<<"\n THIS IS class A";
        }
};
class B
{
    public:
        void show()
        {
            cout<<"\n THIS IS class B";
        }
};
class C : public A, public B
{

};
int main()
{
        C C1;
        // C1.show;
        C1.A::show();
        C1.B::show();
        getch();
        return 0;
}
```

O/P

THIS IS class A

THIS IS class B

Alternative way to solving it

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

2) Ambiguity in multipath hybrid inheritance

Another kind of ambiguity arises if we derive a class from two classes that are each derived from the same class as shown in figure below; which is called multipath hybrid inheritance

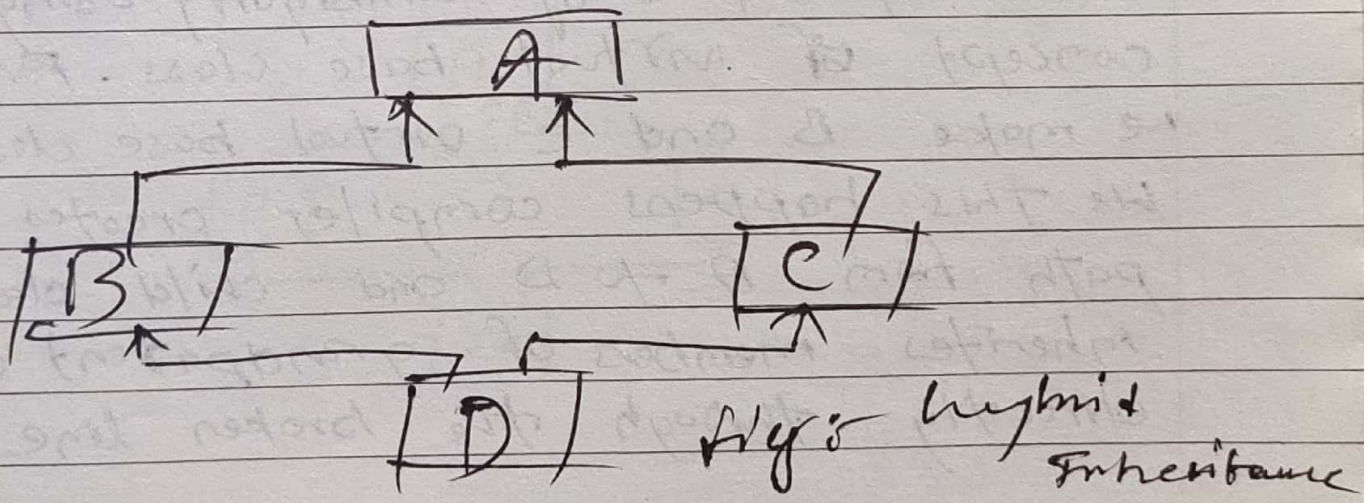


fig: hybrid Inheritance

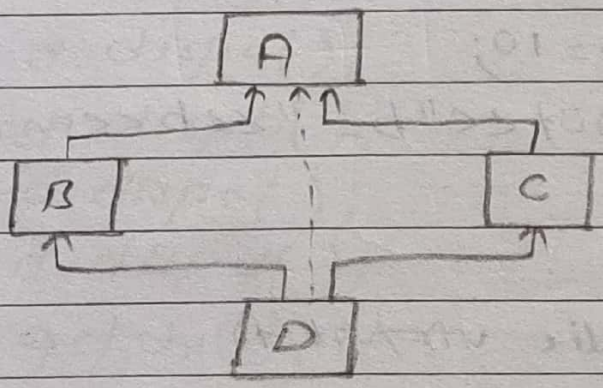
In this case public or protected member of grandparent is derived from the child class which creates confusion to the compiler. We can remove this kind of ambiguity by using the virtual base class concept of

Virtual Base Classes

Consider the following senario class D derived from two classes B and C that are each derived from the same class A. This is called hybrid multipath inheritance. And all the public and protected members from class A inherited into class D twice. Once through path A → B → D and again through the path A → C → D. This causes ambiguity and should be avoided. We can remove this kind of ambiguity using the concept of virtual base class. For this we make B and C virtual base classes. We this happens compiler creates direct path from A to D and child class inherites members of grandpatent class directly through the broken line.

The grandparent is sometime refered to as indirect base class.

```
class A
{
    //body of class A
};
class B : public virtual A
{
    //body of class B
};
class C : public virtual A
{
    //body of class C
};
class D : public B, public C
{
    //body of class D
};
```



Example

```
// virtual base class
#
#
class A
{
    int a;
    public:
        void disp a()
        {
            a=5;
            cout<<" a= "<<a<<endl;
        }
};
class B : virtual public A
{
    int b;
    public:
        void disp b()
        {
            b=10;
            cout<<"b= "<<b<<endl;
        }
};
class C : public virtual A
{
    int c;
```

```
    public:
        void disp c()
        {
            c = 15;
            cout << "c = " << c << endl;
        }
};
class D : public B, public C
{
    int d;
    public:
        void disp d()
        {
            d = 20;
            cout << "d = " << d << endl;
        }
};
int main().
{
        D d;
        d.disp a();
        d.disp b();
        d.disp c();
        d.disp d();
        getch();
        return 0;
}
```

Output

a = 5

b = 10

c = 15

d = 20

data class to object or to class

Containership (Aggregation)

When a class contains objects of another class as its member, this kind of relationship is called containership. And the class which contains objects of another class as its member is called container class.

Example:

```
//
#
#
Class Employee
 {
      int eid;
      float sal;
      public:
          void getdata()
          {
```

of employee:

```
        cout<<"\n Enter Id and salary";
        cin>>eid>>sal;
    }
    void displaydata()
    {
        cout<<"\n Employ ID :"<<endl;
        cout<<"\n Employee salary :"<<sal;
    }
};
class company
{
    int cid;
    char Cname[20];
    Employee e;
    public:
      void getcompany()
      {
        cout<<"\n Enter company Id and name:";
        cin>>cid>>name;
        e.getdata();
      }
      void displaycompany()
      {
        cout<<"\n Company Id :"<<cid;
        cout<<"\n company name:"<<name;
        e.displaydata();
      }
};
```

```
int main()
{
    Company C;
    c.getCompany();
    c.displayCompany();
   getch();
    return 0;
}
```

Output

```
Enter company ID and salary: 8 Prakriti
Enter ID and salary of employ: 9 8000
Company ID: 8
company name: Prakriti
Employee ID: 9
Employee salary: 8000
```

Unit-3

~~Dynamic Polymorphism~~

8.4 Virtual functions and Dynamic polymorphisms

Polymorphism:

Polymorphism means state of having many forms. C++ supports two types of polymorphism: static polymorphism and dynamic polymorphism.

The polymorphism implemented using the concept of overloading functions and operators then it is called static polymorphism or static binding or static linking. This is also called compile time polymorphism because the compiler knows the information needed to call the function at the compile time and therefore compiler is able to select the appropriate function for a particular call at the compile time.

There is another kind of polymorphism called run time polymorphism. In run time polymorphism selection of appropriate function is done dynamically at run time. So this is also called late binding or dynamic binding. C++ supports a mechanism known as virtual

functions to achieve run time polymorphism. Run time polymorphism also requires the use of pointers to object of base class and method overriding.

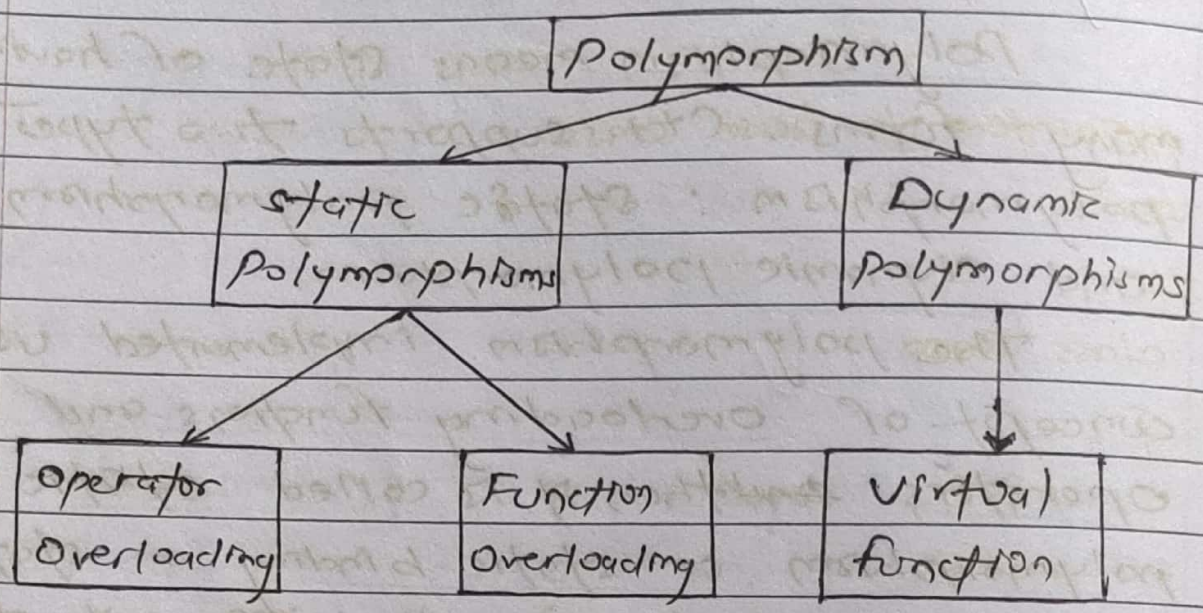


Fig: Polymorphism in C++

Pointer to base class

An object can be used as its own type or as an object of its base type so the derived class object can be manipulated through an address of the base type i.e the pointer to a derived class is a type compatible with a pointer to its base class but vice versa is not true. The limitation of base class pointer is that we can only

refer to the members that are defined of declared in base class and no to the members of derived class that are not inherited from base class.

Example:

```
//Pointers to base class
#
#
class Base
{
    int b;
    public:
        void getb (int i)
        {
            b=i;
        }
        void display()
        {
            cout<<"\n This is Base class:";
            cout<<"\n b="<<b;
        }
};
Class Derived : public Base
{
    int d;
```

```
public:
    void getd(int i)
    {
        d=i;
    }
    void display()
    {
        cout<<"\n This is derived class";
        cout<<"\n d = "<<d;
    }
};
int main()
{
    Base *bp;
    Derived d;
    bp = &d;
    bp -> getb(400);
    bp -> display();
/* bp -> getd(500); invalid because getd is not
                    inherited from Base class */
    getch();
    return 0;
}
```

```
((Derive *) bp) -> getd(500);
((Derive *) bp) -> display();
```

Virtual functions

Virtual means existing in appearance but not in reality. A virtual function is a member function that can be redefined in the derived while preserving its calling properties through reference. We can define the virtual function as below :-

```
class A
{
    virtual void display()
    {
        // function body
    }
};
```

Virtual member function accessed with pointer (Dynamic polymorphism)

We should use virtual functions and pointers to objects of base class to achieve run time polymorphism.

Example

```
// Dynamic polymorphism
#
#
```

```
class A
{
    public:
        virtual void show()
        {
            cout<<"\n This is class A";
        }
};
class B : public A
{
    public:
        void show()
        {
            cout<<"\n This is class B";
        }
};
class C : public A
{
    public:
        void show()
        {
            cout<<"\n This is class C";
        }
};
int main()
{
```

```
A *bp, a;
B b; C c;
bp = &b;
bp->show();
bp = &c;
bp->show();
bp = &a;
bp->show();
getch();
return 0;
}
```

Output

```
This is class B
This is class C
This is class A.
```

Pure virtual function

Pure virtual functions are virtual function with no definition. They start with virtual keyword and ends with equal to 0. We can define pure virtual function as below:

```
class Base
{
  public:
```

```
virtual void getdata()=0 //pure virtual function
virtual void displaydata()=0;
};
```

Abstract class

Abstract class is a class which contains at least one pure virtual function in it. Abstract classes are used to provide an interface for its subclass. Classes inheriting an abstract class must provide definition to the pure virtual function. Otherwise they will also become abstract class.

Characteristics

- An abstract class cannot be instantiated but pointers of abstract class type can be created.
- Abstract class can be normal functions and variables along with a pure virtual function
- Abstract class are mainly used for upcasting. so that its derived classes can use its interface.
- Class inheriting an abstract class must implementing all pure virtual function otherwise they will also become abstract class.

Example:

```
// Pure virtual function
#
#
Class Base //abstract class
{
  public:
    virtual void show()=0; //pure virtual function
};
Class Derived : public Base
{
  public:
    void show()
    {
      cout<<"Implementation of pure virtual
            function";
    }
};
int main()
{
    //Base b; error
    Base *bp;
    Derived d;
    bp = &d;
    bp->show();
    getch();
    return 0;
}
```

dynamic memory allocation

Virtual Destructors

Destructors in the base class can be virtual. Whenever upcasting is done destructors of the base class must be made virtual for proper destruction of the object when the program exit. Note that constructor cannot be virtual.

Example:

```
//Program to illustrate the need of virtual destructor
#
#
class Base
{
    public:
        Base()
        {
          cout<<"\n Base constructor called";
        }
(virtual)~Base()
        {
            cout<<"\n Base destructor called";
        }
};
class Derived : public Base
{
```

int *p = new int
delete p
memory location of p delete

```
    public:
        Derived ( ){ cout<<"\n Derived constructor called ";
        ~Derived ( ){ cout<<"\n Derived destructor called ";
};
int main ( )
{
    Base *bp;
    bp = new Derived ;
    delete bp ;
    getch ( );
    return 0;
}
```

Output

Base constructor called
Derived constructor called
Base destructor called

Here the derived class destructor doesn't execute because static binding of pointer bp is performed. So, bp is bind with the base class destructure. Therefore the statement delete bp deletes the memory of the base class object. To resolve this problem we have to make base class destructors as virtual.

Templates

~~Template and G~~

Template and Generic programming

Template enable us to define generic classes and functions and thus provides support for generic programming. Generic programming is an approach where generic types are used as parameters in program so that they work for a variety of suitable data types and data structures. A template can be used to create a family of class or functions. Templates offer several advantages:-

Advantages

1) Templates are easier to write. We can create generic vision of class or function instead of manually creating specialization.
2) Templates are type safe because the type that templates act upon are known at compile time. Therefore the compiler can perform type checking before error occurs.
3) Templates can be easier to understand since they can provide a straight forward way of abstracting type information.

Types of templates:-

Templates comes in C++ in two variations.

a) Function templates
b) Class templates

a) Function templates

A templates function is defined as an unbounded function. This allow us to create a function template whose functionality can be adopted to more than one type or class without repeating the entire code for each type.

In C++, this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as arguments just like a regular function parameter. These function templates can use these parameters as if they are any other regular type.

```
template <class type1, type2, ...)
ret-type function_name < type1, part1, type2, part2,
                                                ...>
{
    // function body
}
```

Example

```
template <class T>
T max (T x, T y)
{
    return (x > y) ? x : y;
}
```

Example

```
// function template
#
#
template < class Type>
Type max ( Type x, Type y)
{
    return ((x > y) ? x : y);
}
int main ( )
{
    int i1, i2;
    float f1, f2;
    char c1, c2;
    cout << "Enter two integer values : " << endl;
    cin >> i1 >> i2;
    cout << max (i1, i2) << "is larger " << endl;
    cout << "Enter two floats values " << endl;
    cin >> f1 >> f2;
```

```
cout<< max(f1, f2) << " is larger "<<endl;
cout <<"Enter two character :"<<endl;
cin>> c1>>c2;
cout<< max(c1, c2)<<" is larger "<<endl;
  getch();
  return 0;
}
```

Class Templates:

C++ class templates are used whenever we have multiple copies of code for different data type with the same logic. If a set of function or classes have the same functionality for different data types they becomes good candidates for being return as class templat.

Example:

```
// Declaration of templates for stack implementation
#
#
template <class T>
class stack
{
    int top;
```

```
    T st[100];
    public:
      stack()
      {
         top = -1;
      }
      void push(T i)
      {
           st[++top] = i;
      }
      T pop().
      {
         return(st[top--]);
      }
};
int main()
{
      stack<int> ii;
      stack<char> cc;
      ii.push(25);
      cc.push('a');
      cout<<"Integer value = "<<ii.pop()<<endl;
      cout<<"Character value = "<<cc.pop()<<endl;
      getch();
      return 0;
}
```

Output
Integer value = 25
character value = a

Templates Specializations:

a. Class Specialization

Class template specialization allows us to specialize a template class for a particular data type. This is used when we want to define a different implementation for a template when a specific type is passed as template parameter.

Example:

Let us consider we have a class called Mycontainer that can store one element of any type and that it has just one member function called increase which increases its value but we find that when it stores an element of type char it would be more convient to have a completely different implementation with a function member uppercase, so we decide to declare a class template specialization for that type.

```
// class template specialization
#
#
template <class T>
```

```
class Mycontainer
{
    T element;
    public:
     Mycontainer (T arg)
     {
        element =arg;
     }
     T increase ()
     {
        return (++element);
     }
};
//class template specialization
 template <>
class Mycontainer <char>
 {
    char element;
    public:
      Mycontainer (char arg)
      {
         element = arg;
      }
      char uppercase ()
      {
          if((element >='a' && element <='z'))
```

```
    {
        return(element = element + ('A' - 'a'));
    }
  }
};
int main()
{
    Mycontainer<int> myint(5)
    Mycontainer<char> mychar('d');
    cout<<" After increment = "<<myint.increase()<<endl;
    cout<<" Uppercase value = "<<mychar.uppercase();
    getch();
    return 0;
}
```

Output

After increment = 6

Uppercase value = D

b. Function template specialization

Function templates specialization is use to create a specialized version of a function for a type. To define the specialized function replace the template type with the specific type.

Example

Let us consider we have a program that compares two arguments and find out the larger value. If the function template is instantiated with a template argument of type string, the generated instance fails to compile because the string class doesnot support either the less than or greater than operator. The code given below shows how we can specialize a function template. Remember that the general function template must have been declared before providing function template specialization.

```
//Function template specialization
#
#
#include<string.h>
template<class T>
T max (T a, T b)
```

```
{
    return (a>b)? a:b;
}
char * max (char *a, char*b)
{
    return strcmp(a,b)>0?a:b;
}
int main()
{
  cout<<" max(4,6) = "<<max(4,6)<<endl;
  cout<<" max(4.5, 3.2) = "<<max(4.5, 3.2)<<endl;
  cout<<" max(\"Ram\", \"Hari\")"<< max("Ram",
        "Hari");
  getch();
  return 0;
}
```

Output

max(4,6) = 6

max(4.5, 3.2) = 4.5

max("Ram", "Hari") = Ram

Exception Handling

Exceptions are run time anomalies or unusual condition that a program may encounter while executing. Exceptions might include conditions such as division by zero, access to an array out of its bound, running out of memory or disk space, not being able to open a file, trying to initialize an object to an impossible value etc.

When a program encounter an exceptional condition, it is important that it is identified and dealt effectively. C++ provides built in language features to detect and handle exception which are basically runtime errors.

The purpose of exception handling mechanism is to provide means to detect and report an exceptional circumstance so that appropriate action can be taken. The mechanism suggest a separate error handling code that performs the following task.

- Find the problem (Hit the exception
- Inform that an error has occured (Throw the exception)
- Receive the error information (catch the exception)
- Take corrective action (Handle the exception.

The error handling code basically consists of two segments, one to detect and to throw exception, and the other to catch the exceptions and to take appropriate actions.

Exception handling mechanism in C++ is basically built upon three keywords: try, throw and catch.

i) **Try**: A try block indicates a block of code for which particular exception will be activated. It is followed by one or more catch blocks.

ii) **Throw**: A program throws an exception when a problem occurs. This is done using throw keyword.

iii) **Catch**: A program catches an exception with an exception handler at the place in a program where we want to handle the problem. The catch keyword indicates the catching of an exception.

The figure below shows the exception handling mechanism if try block throws an exception.

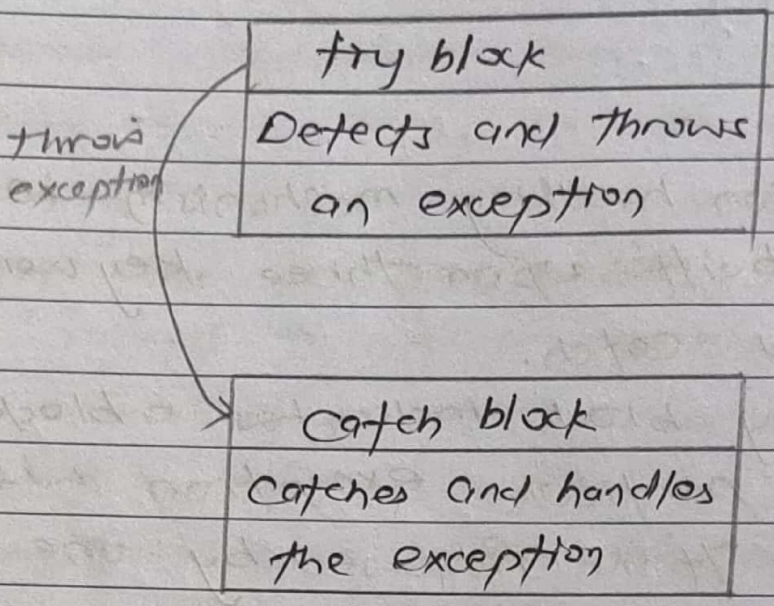


Fig: Exception handling mechanism

The general form of try catch statement is as follows:

```
try
{
    //code to be tried
    throw exception
}
catch
{
    //code to be executed in case of exception
}
```

Example1:

```
// try block throwing an exception
#
#
int main()
{
    int a, b;
    cout<<"Enter the value of a and b:";
    cin>>a>>b;
    try
    {
        if (b==0)
            throw b;
        else
            cout<<"\n Result="<<(float) a/b;
    }
    catch (int i)
    {
        cout<<"\n Divide by zero error:b="<<i;
    }
    cout<<"\n End of the exception code";
    getch();
    return 0;
}
```

Output
Enter the value of a and b: 10, 5
Result = 2.0
End of the exception code

Example 2:

```
//Function invoked by try block for throwing excepti
#
#
void divide(int a, int b)
{
    if(b==0)
        throw b;
    else
        cout<<"Result="<<(float)a/b;
}
int main()
{
    int a,b;
    cout<<"Enter the values of a and b:";
    cin>>a>>b;
    try
    {
        divide(a,b);
    }
```

```
    catch (int i)
    {
        cout<<"\n Divide by zero exception: b="<<i;
    }
    getch();
    return 0;
}
```

Output

First Run

Enter the value of a and b: 10 0
Divide by zero exception: b=0

Throwing mechanism

When an exception is detected, it is thrown using throw statement in one of the following form

```
throw (exception);
throw ; //used for rethrowing exception
```

The operand exception may be of any type (built-in and user-defined), including constants. When an exception is thrown the catch statement associated with the try

block will catch it. That is, the control exits the current try block, and is transferred to the catch block after the try block. Throw point can be in a deeply nested scope within a try block or in a deeply nested function call. In any case, control is transferred to the catch statement.

Rethrowing an Exception:

A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke throw without any arguments as shown below:

```
throw;
```

This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

Example:

```
// using throw statement
#
#
```

Example:

```
//Rethrowing an exception
#
#
void divide (int a, int b)
{
    try
    {
        if (b==0)
            throw b;
        else
            cout<<"Result = "<<(float) a/b;
    }
    catch(int)
    {
        throw;
    }
}
int main()
{
    int a,b;
    cout<<"Enter the value of a and b: ";
    cin>>a>>b;
    try
    {
        divide(a, b);
    }
```

```
catch (int)
{
    cout<<"\n Divide by zero exception: b="
}
getch();
return 0;
}
```

O/P

Enter the value of a and b: 5 0
Divide by zero exception : b=0

Catch Mechanism

The catch block following the try block catches any exception. Code for handling exception is included in catch blocks. The general form of catch block is

```
catch (type arg)
{
    //body of catch block
}
```

The type indicates the type of exception that catch block handles. The parameter arg is optional. If it is named, it can be used in the exception handling code. The catch statement catches an exception whose type matches with the type of catch argument. When it is caught, the code in the catch block is executed. After its execution, the control goes to the statement immediately following the catch block. If an exception is not caught, abnormal program termination will occur. The catch block is simply skipped if the catch statement doesnot catch an exception.

Multiple catch statement:

It is possible that a program segment has more than one condition to throw an exception. In such cases, we can associated more than one catch statement with a try as shown below:

```
try
{
    // Try block
}
```

```
catch (type1 arg)
{
    //catch block1
}
catch (type2 arg)
{
    //catch block2
}
...
catch(typeN arg)
{
    //catch block N'
}
```

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed. After that, the control goes to the first statement after the last catch block for that try skipping other exception handlers. When no match is found, the program is terminated.

Example:

```
/*Program using multiple catch blocks for a single
try block */
#
#
int test (int x)
{
    try
    {
        if(x==0)
            throw x;
        if(x==1)
            throw 1.0;
    }
    catch (int i)
    {
        cout<<"\n caught an integer";
    }
    catch (double d)
    {
        cout<<"\n caught a double";
    }
}
int main()
{
    test(0);
    test(1);
```

```
    getch();
    return 0;
}
```

O/P

```
Caught an integer
Caught a double.
```

Catching all Exception

If we want to capture all possible types of exceptions in a single catch block, for that we have to write three points instead of the parameter type and name separated by catch.

```
try
{
    //try block
}
catch(...)
{
    // all exception handling code goes here
}
```

Example:

```
// Catching all exceptions
#
#
void test (int x)
{
    try
    {
        if (x==0)
            throw x;
        if (x==1)
            throw 1.0;
    }
    catch (...)
    {
      cout << "Exception occured " << endl;
    }
}
int main()
{
    test(0);
    test(1);
    getch();
    return 0;
}
```

Nested try-catch :

When a try-catch block is written inside another try-catch block then it is called Nested try-catch. C++ allows ~~except~~ to next exceptions, using the same techniques as conditional statements. The general form of Nested try-catch is

```
try
{
    try
    {
        //try code
    }
    catch( )
    {
        //Exception handler
    }
}
catch ( )
{
    //Exception handler
}
```

Example

```
//nested try-catch statement
#
#
int main()
{
    int a, b;
    cout<< "Enter the value of a and b:";
    cin>>a>>b;
    try
    {
        try
        {
            if(b==0)
                throw b;
            else
                cout<<"Result = "<<(float)a/b;
        }
        catch(int)
        {
            throw;
        }
    }
    catch(int i)
    {
        cout<<"\n Divide by zero exception";
    }
```

```
    getch();
    return 0;
}
```

O/P
Enter the value of a and b : 5 0
Divide by zero exception.

Specifying Exceptions:-

It is also possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to the function definition. The general form is as follows:-

```
type function (arg-list) throw (type-list)
{
    //function body
}
```

The type list specifies the type of exception that may be thrown. Throwing any other type of exception will cause abnormal program termination.
If we wish to prevent a function from

throwing any exception, we may do so by making the type-list empty. Hence the specification in this case will be

```
type function (arg-list) throw ()
{
    //function body
}
```

Example:-

```
//Specifying Exception
#
#
void test (int x) throw (int, double, char)
{
    if (x==0)
        throw x;
    if (x==1)
        throw 1.0;
    if (x==2)
        throw 'a';
}
int main()
{
    int n;
```

```
cout<<" Enter the value of n=";
cin>>n;
try
{
    test(n);
}
catch (int i)
{
 cout<<" caught an integer";
}
catch (double d)
{
 cout<<" caught an double";
}
catch (char c)
{
 cout<<" caught an character";
}
getch();
return 0;
}
```

o/p

Enter the value of n = 0
Caught an integer.