INTRODUCTION

POP

* Procedure/structure oriented
- top down approach

* Data is insecure

* Global data is shared among the functions in program

OOP

object-oriented
- bottom up approach

* Data is highly secured with access specifiers

* Data is shared among objects through member fn.

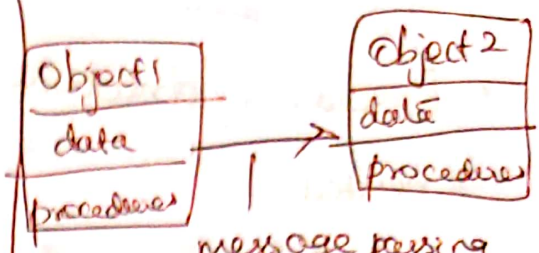
POP

Procedure operates on set of data sequentially

procedure 1 → data

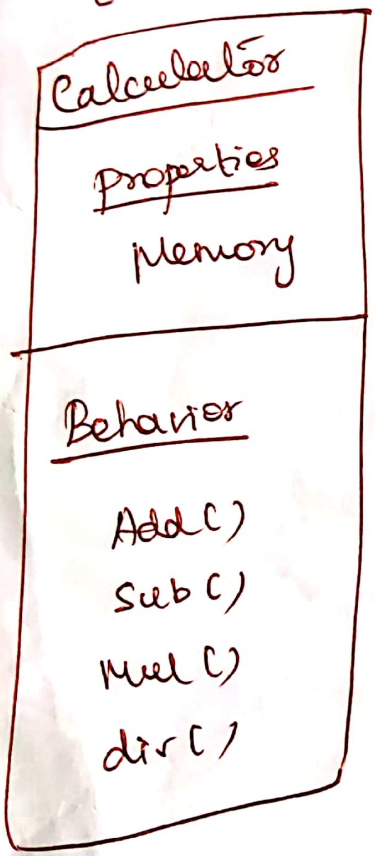
procedure 2 → data

OOP



Multiple objects communicate to create a sln thru msg passing

eg.

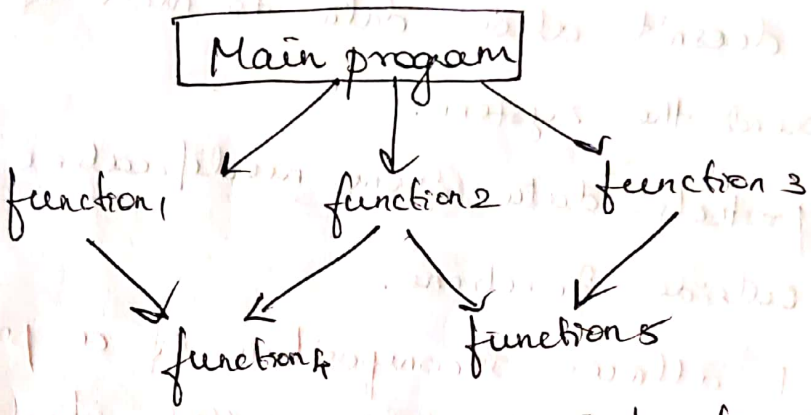


1.1 Procedure Oriented Programming (POP)

In POP, the problem is viewed as a sequence of things to be done such as

- reading
- calculating
- printing

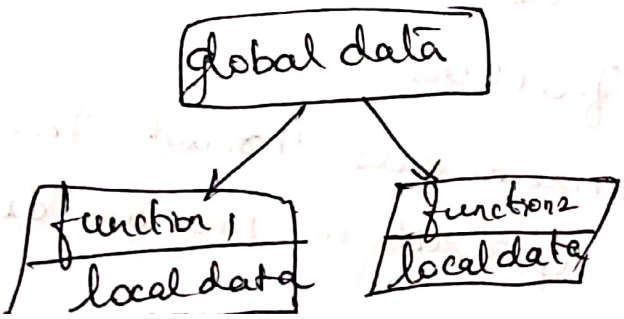
- A no. of functions are written to accomplish this tasks.



* POP - Write a list of instructions for the computer to follow
  - Organize these instructions into groups known as functions.

Disadvantages

* Little attention given to data that are being used by various functions.
  - Many data items are placed as global & accessed by all functions



* Global data are vulnerable to changes & create chances for bugs to occur.
* Pop does not model real world problems very well.
* Data moves openly around the system from function to function.

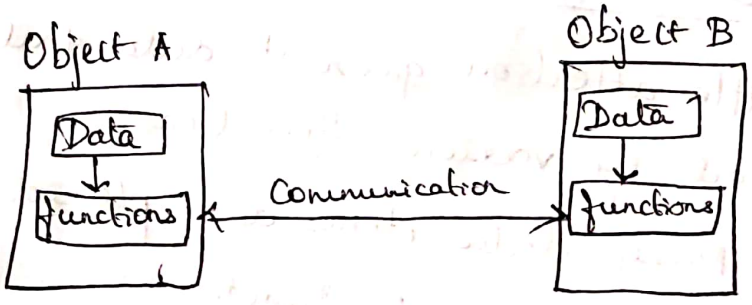
1.2. Object Oriented Paradigm (OOP)

OOp doesn't allow data to move freely around the system.

- protects data from modification from outside functions.

"OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects".

* Data of an object can be accessed only by the functions associated with that object



* Data is hidden & cannot be accessed by external functions.
* Objects communicate through functions.
* Bottom up approach in program design.

Print even number or not. POP

```
#include<stdio.h>
void main()
{ int a, i;
  for(int i = 0; i<5; i++)
  {
     printf("enter the number %d", i+1);
     scanf("%d", &a);
  }
  for( i=0; i<5; i++)
  {
     if (a %2==0)
       printf("%d number is even", a);
     else
       printf("%d number is odd", a);
  }
  getch();
}
```

Problem is devided into different steps.

1. identify variables, data
2. decide how many nos/.
3. Logic to find even no/.
4. Read i/p, calculate / manipulate and display o/p.

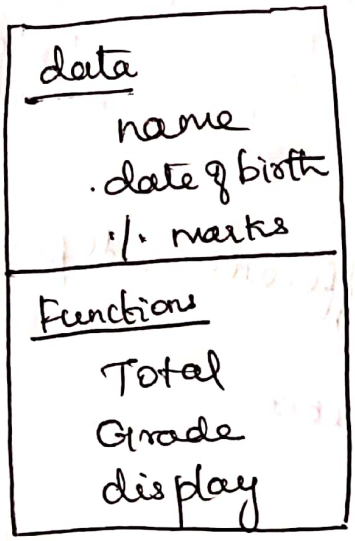
Concepts / Principles of OOP

* Objects
* classes
* Abstraction
* Encapsulation
* Inheritance
* Polymorphism
* Dynamic binding
* Message Passing

Objects

- a real world entity with its own features or characteristics. It has data and code to manipulate.

student - object



Classes

Objects are variable of type class.

Class is a collection of objects of similar type

Class is an user defined data type.

Abstraction

"Act of representing essential features without including background details".

e.g. dvd player - remote buttons - ⏩ press u can view video forwarded few seconds but user dont know technically how it moves | works inside

- Identify information that should be hidden.

eg: ATM machine - slw - withdraw, deposit & methods

hiding how it works.

Encapsulation

Wrapping up of data and functions into a single unit is known as encapsulation.

i.e. feature of a class.

"Data is not accessible to outside world and only those functions which are wrapped in class can access it".

Inheritance

Objects of 1 class acquires properties of objects of another class. - hierarchial classification

Parent class - base class

Derived class - child class

- use is 'reusability'

- Add additional features to existing class without modifying it by deriving a new class from existing one.

- New class has combined features of both the classes.

Polymorphism

- ability to take more than one form.

- an operation may exhibit different behaviors in different instances. The behavior depends on types of data used in operation.

eg. operation of addition. — 2 number - generate sum
                          — 2 strings - string concatenation

Types

1. Operator overloading
2. Function overloading

* Process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

* Single function name to perform different types of tasks is known as function overloading.

Dynamic Binding

Binding refers to the linking of a procedure call to code to be executed in response to the call.

Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time.

Message Passing

Objects communicate with each other. A message for an object is a request for execution of a procedure

- It consists of name of object, function & information to be sent.

eg. m.salary(name);
↑ object ↑ msg ↑ infor..

Benefits of OOP

* Redundant code can be eliminated
* Reusability achieved & extend use of existing classes.
* Saves development time.
* Build secure programs
* Easily can upgrade from small to large systems
* Software complexity can be easily managed.

Applications of OOP

* User interface design
* Real time system

* Simulation & Modeling
* AI, expert s/m
* Neural Networks
* CAM/CAD system

C++ Fundamentals

* C++ is an object-oriented programming language.
* Developed by Bjarne Stroustrup at AT&T Bell lab, USA - 1980
* C++ is an extension of C (Increment operator ++ - augmented version of C)
* C++ solves real world problem easily & effectively

Structure of C++

Include files
Class declaration
Member function definitions
Main function

Tokens

The smallest individual unit in a program is known as token.

C++ tokens — keywords
- identifier
- constants
- strings
- operators

Keywords

break, if, else, for, int, long, do while, public etc.

Identifiers

- names of variables, functions, arrays, classes created by programmer.

Rules for identifiers

* Only alphabetic characters, digits, underscores are permitted
* Name can't start with digit
* Keyword can't be used as a variable name
* No limit on length of name.

Constants

- fixed values - do not change during execution of program.

eg. x = 123 - decimal integer
y = 'A' - character constant
z = 12.34 - floating point

Data types

1. Built-in / Primitive types
   - Integral type → int, char
   - floating type → float, double
   - void.

2. User defined type
   - structure
   - union
   - class
   - enumeration

3. Derived type:
   - array
   - function
   - pointers
   - reference.

User defined datatypes

In order to group logically related data items together we have arrays but it is used to group similar type of elements.

Structures are used for grouping elements of dissimilar types.

Syntax:

```
struct name
{
    datatype member1;
    datatype member2;
    ...
};
```

eg.

```
struct book
{
  char title[25];
  char author[25];
  int pages;
  float price;
};
struct book b1, b2, b3;
```

- here b1, b2, b3 are declared as variables of user defined type 'book'

- To access member elements of a structure use (.) dot operator.

```
b1.pages = 500;
b2.price = 250.5;
```

Unions

Union is similar to structures. Difference is in their implementation.

* Size of structure type is equal to the sum of the sizes of individual member types.

* Size of union = size of its largest member element

eg. union result
{
int marks;
char grade;
float percent;
};

→ largest size member element is float.
∴ 4 bytes is size of union

if it is structure, size would be int + char + float
= 2 + 1 + 4 = 7 bytes.

∴ Union is memory efficient. It occupies less memory ie. share common memory space.

Enumerated data types

It provides a way for attaching names to numbers.

enum keyword is used. It automatically enumerates a list of words by assigning them values as 0, 1, 2....

eg. enum colour { red, green, blue, orange };
colour background;
↳ variable of type colour

By default, enumerators are assigned integer values starting with 0, 1 .... We can override the default by explicitly assigning integer values to enumerators.

eg. enum colors { red, blue = 4, green = 8 };
↓
0 by default.

enum colors { red = 5, blue, green };
↓ ↓ ↓
5 6 7.
↑
automatically initialized.

Storage classes

The storage class of a variable specifies the lifetime & visibility of a variable within the program.

Lifetime — duration till which variable remains active during program execution.

Visibility - scope of variable.

Types :

1) Automatic / auto
   - default storage class of any type of variable
   - visible to function in which it is declared.
   - destroyed as soon as pgm control leaves its contained function block
2) External - global variable
   - declared outside function
   - Visibility all across program

3) static

- declared inside a function block mean, it does not get destroyed after function is executed but retains its value so that it can be used by future function calls.

4) Register

- stored in CPU registers.
- Increase its access speed
- Program run faster.

	Auto	External	static	Register
Lifetime	function block	Entire program	Entire program	function blok
Visibility	local	global	local,	local
Initial value	garbage	0	0	garbage
Storage	stack segment	data segment	data	CPU register
Purpose	Local variables used by a single function	global variables used throughout program	local variables retaining their values throughout pg	variables using CPU registers fo storage purpo
keyword	auto	extern	static	register

Derived data types

Arrays - collection of data items of similar type.

int marks [4] = {20, 40, 60, 80};

C

char string [3] = "xyz";

C++

char string [4] = "xyz"

size 1 > no. of characters in string

Functions

- Building block of a program.
- Objective is to save memory space when function is likely to be called many times.

Calculate factorial of a number

Recursive function example

```
#include<iostream.h>
int fact (int n)
{
    if(n==0)
    return 1;
    return(n * fact(n-1)); //recursive
                           function call
}
int main()
{
    int num;
    cout<<"enter number";
    cin>>num;
    cout<<"factorial of "<<num<<"is"
                <<fact(num);

    return 0;
}
```

Initialize an array

```
int marks[5] = { 50, 70, 90, 100, 85 };
```

Find largest element in an array

```
int main()
{
   int numbers[5], i, large;
   for(i=0; i<5; i++)
   {
     cout<<"enter number"<<(i+1)<<" :";
     cin>>numbers[i];
   }
   large = numbers[0];
   for(i=1; i<5; i++)
   {
      if(large < numbers[i])
        large = numbers[i];
   }
   cout<<"Largest number is "<<largest;
   getch();
}
```

```
Enter number 1 : 34
      "      2 : 45
      "      3 : 56
      "      4 : 12
      "      5 : 3
Largest number is : 56
```

Arrays

- Collection of data items of same type stored in sequential memory location.
- Elements in an array is accessed using an index

Declare

datatype arrayname[size];

eg:

int empid[100]; // declares an integer array with 100 elements

C++ program to get 10 numbers from user & display its sum

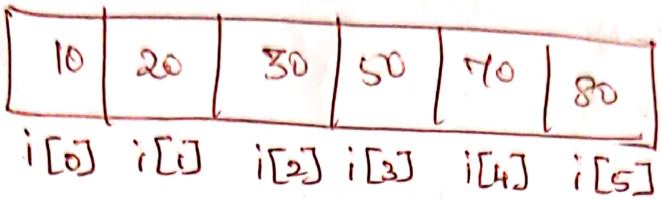
```
int main()
{
  int arr[10], sum=0, i;
  cout<<" enter 10 numbers ";
  for(i=0; i<10; i++)
  {
     cin>>arr[i];
     sum = sum + arr[i];
  }
  cout<<" Sum of array values are "<<sum;
  return 0;
}
```

o/p

Enter 10 numbers
10
20
30
.
.
Sum of array values are 1000

Arrays
- Single Dimensional Array
- Multiple Dimensional Array

1 D array



Multi-D-array

Syntax: datatype arrayname [d1] [d2] ... [dn];

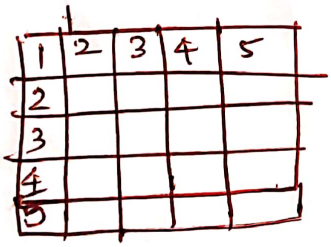
eg: int a[10][10];

- declares an integer array with 100 elements (10*10)

eg2 float f[5][10];

- declares a float array with 50 elements (5*10)

2D array



Enter elements of a matrix & display it:

```
#include<iostream.h>
int main()
{
    int arr[10][10], row, col, i, j;
    cout<<"enter size of row & column";
    cin>>row>>col;
    cout<<"enter elements of matrices"
```

```
for(i=0; i<row; i++)
  for(j=0; j<col; j++)
    cin >> arr[i][j];
cout<<"Display matrix";
for(i=0; i<row; i++)
  {  for(j=0; j<col; j++)
       cout<< arr[i][j]<<" ";
  }
  getch();
}
```

op

enter size 2 3

enter elements

12 131 51

19 13 24

Display

12 131 51

19 13 24

Initializing 2-dimensional arrays

```
int table[2][3] = { 0,0,0, 1,1,1 };
```

1st row, 2nd row

Initialization is done row by row.

(or) it can be written as:

```
int table[2][3] = { {0,0,0}, {1,1,1} };
```

(or)

```
int table[2][3] = {
                    {0,0,0},
                    {1,1,1},
                  };
```

Multidimension

int a[3][5][12] - 3D array of 180 elements
(3×5×12=180)

3D array is represented as a series of 2D -arrays

eg. survey[2][3][10] - denotes rainfall in the month of October during 2nd yr in city-3

Year 1

month / city	1	2	.....	12
1				
2				
3				
4				
5				

Year 2

Month / city	1	2	12
1			
2			
⋮			
5			

Strings

String is a sequence of characters that is treated as a single data item.

- Group of characters defined between double quotation is string constant
  "Students details".

Operations performed on character strings includes:

- Reading & writing strings
- Combining strings
- copying one string into another
- Comparing strings for equality
- Extracting a portion of string

declare

```
char string_name[size];
                 ↓
              no. of characters

    char city[20];
              ↳ greater than actual size
```

```
#include<iostream.h>
int main()
{
  char str[100];
  cout << "enter string";
  cin >> str;
  cout << "You entered:" << str << endl;
  return 0;
```

if suppose you type as "I like programming" it will display only I because space after I will not be considered because space is taken as terminating character

∴ To read & display entire line entered by user including blank space:

```
#include<iostream.h>
int main()
{
    char str[100];
    cout<<"enter a string";
    cin.get(str,100);
    cout<<"string is "<<str<<endl;
    return 0;
}
```

cin.get function - read text containing blankspace.

Built-in functions

1) strcpy(s1,s2); copies string s2 into string s1

2) strcat(s1,s2); concatenates string s2 into end of string s1

3) strlen(s1) - returns length of string s1

4) strcmp(s1,s2) - returns 0 - if s1 & s2 equal
returns <0 if s1 < s2
returns >0 if s1 > s2

5) strchr(s1, ch); - returns a pointer to 1st occurrence of character 'ch' in string s1

6) strstr(s1, s2); - returns a pointer to 1st occurrence of string s2 in string s1.

```
#include<iostream.h>
int main()
{
  char str1[10] = "hello";
  char str2[10] = "world";
  char str3[10];
  int len;
  strcpy(str3, str1);
  cout<<"Copied string is "<<str3;
  strcat(str1, str2);
  cout<<"Concatenated string is "<<str1;
  len = strlen(str1);
  cout<<"Length is "<<len;
  return 0;
}
```

o/p

Copied string is hello
Concatenated string is helloworld.
length is 10

```
#include <iostream.h>
void main()
{
  char s1[20], s2[20], s3[20];
  int x, l1;
  cout << "enter 2 strings";
  cin >> s1 >> s2;
  x = strcmp(s1, se);
  if( x != 0)
  {
    cout << "strings not equal";
    strcat(s1, s2);
  }
  else
    cout << "strings are equal";
  strcpy(s3, s1);
  l1 = strlen(s1);
  l2 = strlen(s2);
  l3 = strlen(s3);
  cout << "length of 3 strings are " << l1 << l2 << l3;
}
```

In C++, you can also create a string object for holding strings.

- it has no fixed length, can be extended as per requirements.
- instead of using cin >>, cin.get() / getline() is used

ANSI C++

```
#include<string.h>
int main()
{
string str; // declaring a string object
cout<<"enter string";
getline(cin, str);
cout<<"You entered"<<str;
return 0;
}
```

* String objects are used like any other built-in type data.

String class available in <string.h>.

Functions of string class

append() – appends a part of string to other string

at() – obtains character stored at specified location.

capacity() – gives total elements that can be stored.

empty() – returns true if string is empty

size() – gives no. of characters in string

swap() – swaps the given string with invoking string

String objects

```
String s1;
String s2("xyz");
s1 = s2;          // assigning string objects
s3 = "abc" + s2;  // concatenating strings
cin>>s1;          // reads one word from keyboard
```

getline(cin, s1); // read a line from keyboard

s1.insert(4, s2); // inserts character at a specified location

eg. s1 = 12345;
s2 = abcde;
s1.insert(4, s2);

↓ o/p

1234abcde5

0	1	2	3	4
1	2	3	4	5

↑ insert

1234abcde5

Modular Programming with Functions

"Set of actions to perform a specific task is known as functions".

- Certain type of ~~function~~ operations or calculations are repeated at many points throughout a program → function

"A function is a self-contained block of code that performs a particular task"

Functions
- Built-in - eg. main(), strlen(), strcmp() → library functions
- Userdefined

```
Main()
{
    fun1();
    fun2();
}

fun1()
{
    return;
}
```

call →

Elements of user defined function

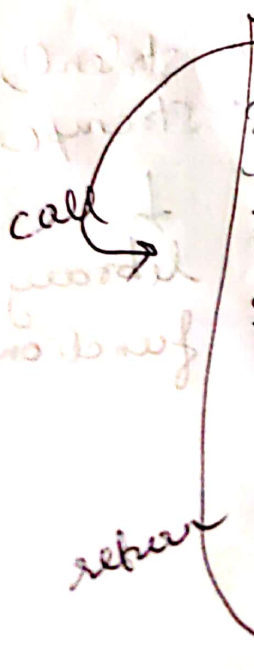
* Function declaration
* Function definition
* Function call

Syntax

```
function returntype  functionname (parameters)
{
    variables;
    statements;
    return;
}
```

eg.

```
int add(int, int);   // declaration
main()
{
    int a, b, c;
    c = add( a, b);
}
int add( int x, int y);
{  int z;
   cout << "enter 2 values";
   cin >> x >> y;
   z = x+y;
   return z;
}
```



Category of Functions

1) Functions with no arguments and no return values
2) " with " " " "
3) " with " and return value
4) " without " and " "

No args, No return values

```
main()
{
fun();
}
fun()
{
==
return;
}
```

No args, return value

```
main()
{
z=fun();
}
int fun()
{
==
return;
}
```

With return value, with arguments

```
main()
{
z=fun(x,y);
}
int fun(int a, int b)
{
==
return;
}
```

With arguments No return values

```
main()
{
fun(x,y);
}
void fun(a,b)
{
}
```

Programs for functions categories

1) Functions with arguments, no return values

```
void add(int, int);
void add(int x, int y)
{
    int result;
    result = x + y;
    cout<<"sum of "<<x<<" and "<<y<<result;
}
void main()
{
    add(30, 15);
    getch();
}
```

2) Functions without arguments, with return value

```
int send(void);
#include<iostream.h>
void main()
{
    int z;
    z = send();
    cout<<"you entered"<<z;
    getch();
}
int send()
{
    int num;
    cout<<"enter";
    cin>>num;
    return(num);
}
```

3) Functions with arguments, with return values

```
int add(int a, int);
#include<iostream.h>
int main()
{
    int a, b, c;
    cout<<"enter numbers";
    cin>>a>>b;
    c=add(a,b);
    cout<<"result is "<<c;
}
int add(int x, int y)
{
    int z;
    z=x+y;
    return z;
}
```

4) Functions without arguments without return values

```
void sum(void);
#include<iostream.h>
void main()
{
    sum();
    getch();
}
void sum()
{   int a,b,c;
    cout<<"enter numbers";
    cin>>a>>b;
    c=a+b;
```

```
cout<<"result is"<<c;
}
```

Program to return multiple values — Using Arrays / Using Pointers

```
void find(int, int, int);
#include<iostream.h>
void main()
{
  int x, y, arr[2];
  cout<<"enter 2 numbers";
  cin>>x>>y;
  find(x, y, arr); cout<<"greater number
                  smaller number is
                  inside"<<arr[0]<<
                  arr[1];
  getch();
}
void find(int a, int b, int arr[])
{
    if(a>b)
    {
      arr[0] = a;
      arr[1] = b;
    }
    else
      {
        arr[0] = b;
        arr[1] = a;
      }
}
```

Functions that return multiple values

```
void mathop( int x, int y, int *s, int *d);
main()
{
   int x = 20, y = 10, s, d;
   mathop(x, y, &s, &d);
   cout << "values are." << s << d;
}
void mathop( int a, int b, int *sum, int
                                  *diff)
{
   *sum = a+b;
   *diff = a-b;
}
```

value of x is passed to a

" y " to b

address of s " to sum

" d " to diff

Parameters Passing

- Call by value
- Call by reference

Call by value	Reference
* A copy of variable is passed	* A variable itself is passed
* Change in copy of variable doesn't modify original value outside function	* Change in variable affect value outside the function also

Program for call by value

```
#include<iostream.h>
void change(int);
int main()
{
  int data = 3;
  change(data); // function call
  cout << "Data value is " << data;
  return 0;
}
void change(int data)
{
  data = 5; → changed data
}
```

changes not modified outside fn/.

o/p

Data value is 3

so data = 3 is o/p

Call by reference

- actual & formal parameters share the same address space.
- value changed inside the function is reflected inside & outside function.

```
#include<iostream.h>
void swap(int*,int*);
int main()
{
  int x=5, y=6;
  swap(&x, &y);
  cout<<"Values are "<<x<<y;
  return 0;
}
void swap(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
```

o/p

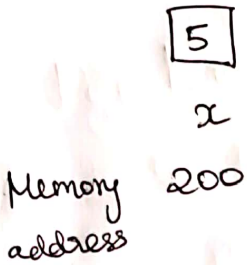
Values are 6 5

Pointers

- derived data type
- contains memory address as their value
- more efficient to handle arrays
- used to return multiple values from functions
- supports dynamic memory mgmt.

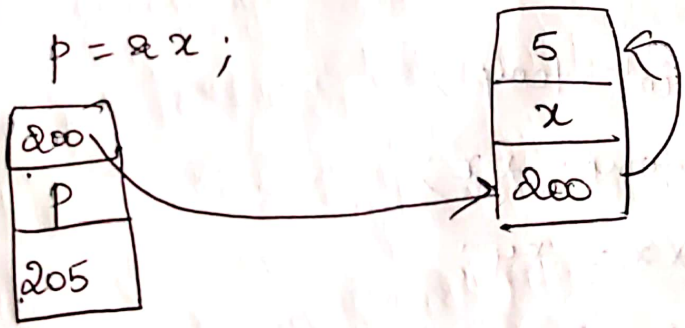
```
int x = 5;
```

→ Normal variable



Pointer variable - holds memory addresses of another variable.

```
int *p;
p = &x;
```

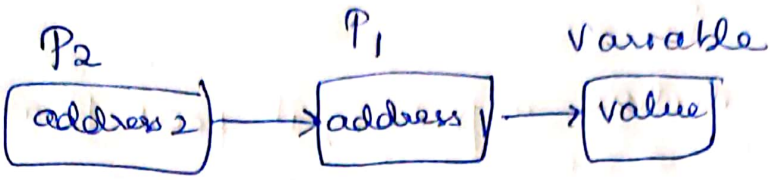


$\therefore *p = 5$

& - reference operator

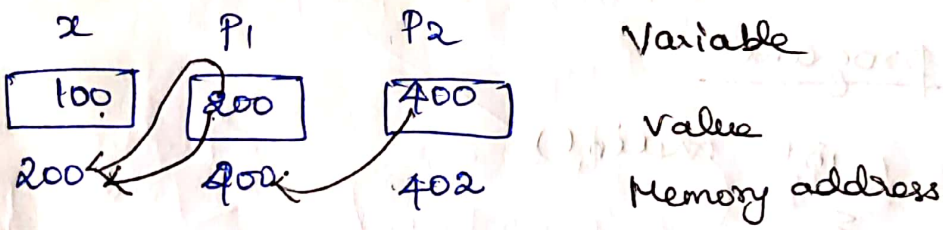
* - dereference / value at address operator.

Chain of pointers



→ Multiple indirections

```
main()
{
  int x, *p1, **p2;
    x=100;
    p1 = &x;
    p2 = &p1;
    cout<<" value is " << x;
    cout<<" Pointer value is " <<*p1;
    cout<<" Double pointer value is " << **p2;
}
```

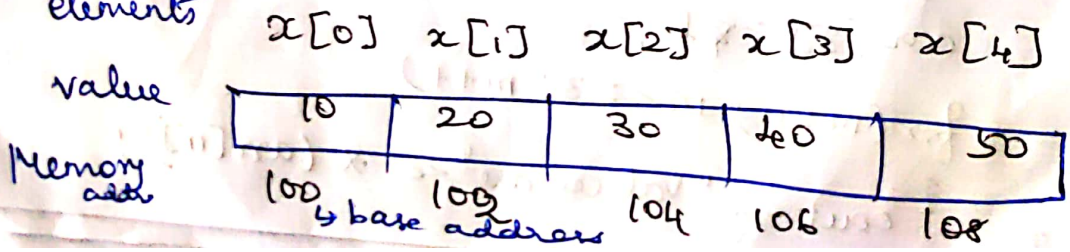


*p1 = 100

p2 = 100 ← multiple indirection (p2)

Arrays & Pointers

int x[5] = {10, 20, 30, 40, 50};



Base address is location of first element of the array.

If we declare a pointer, it point to array by

```
p = &x[0];
```

To access all values of x use p++ to move from one element to another.

p = &x[0] = 100

p+1 = &x[1] = 102

p+2 = &x[2] = 104

p+3 = &x[3] = 106

p+4 = &x[4] = 108

*(p+3) = 40

Program

```
int main()
{
    int *arr[4];
    int i=10, j=20, k=30, l=40, m;

    arr[0] = &i;
    arr[1] = &j;
    arr[2] = &k;
    arr[3] = &l;
    for(m=0; m<=3; m++)
    { cout << "Values are" << *(arr[m]);
```

Program using pointers to compute sum of all elements stored in an array

```
void main()
{
    int *p, sum, i;
    int x[5] = {5, 10, 15, 20, 25};
    i=0; sum=0;
    p = &x[0];
    while(i<5)
    {
        cout<<"x["<<i<<"]"<<*p;
        sum = sum + *p;
        i++, p++;
    }
    cout<<"Sum"<<sum;
    getch();
}
```

Note

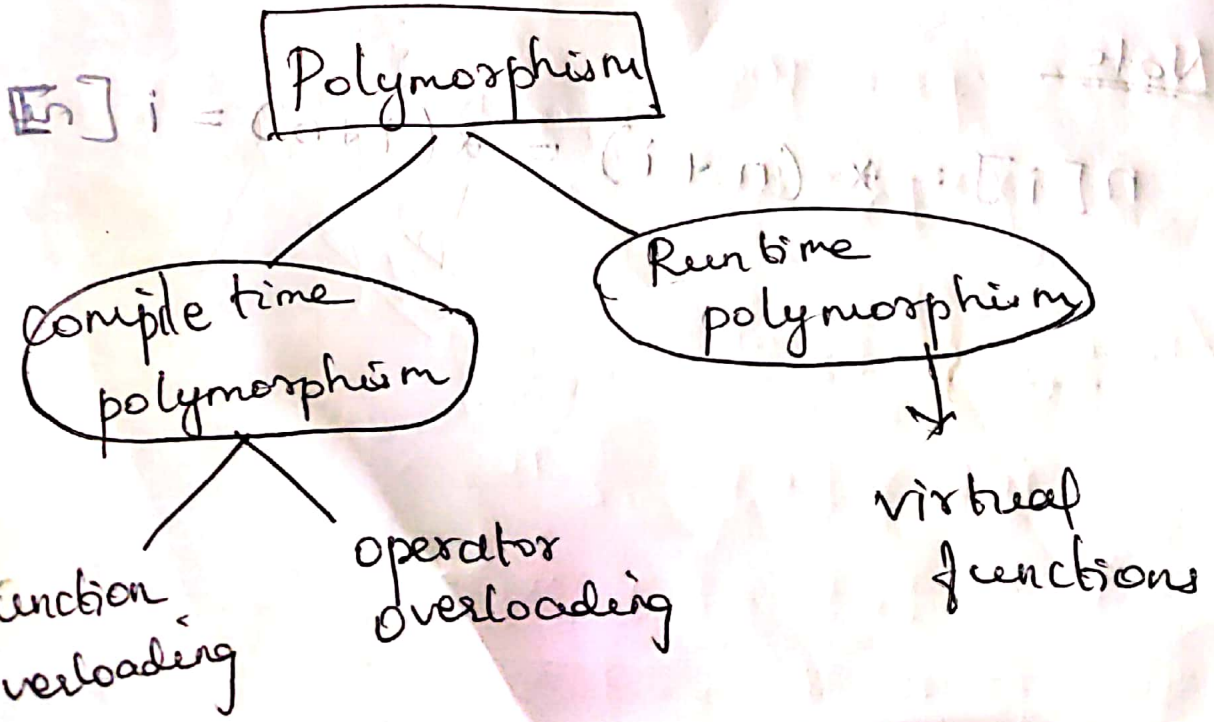
$n[i] = *(n+i) = *(i+n) = i[n]$

Runtime Binding

* Dynamic binding is one of the powerful features of C++.
  - it requires the use of pointers to objects.

* If the compiler is able to select the appropriate function for a particular call at compile time itself it is called as early binding / static binding / static linking
  - also called as compile time polymorphism.
  eg. function, operator overloading.

* The appropriate member function is selected when the program is running. It is late or dynamic binding or runtime polymorphism
  - it is supported by virtual functions.



Operators & Expressions

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.

* Operators are used to manipulate data and variables.

Categories

1. Arithmetic operators (+, -, *, /, %)
2. Relational " (<, >, <=, >=, ==, !=) → equal to, → not equal to
3. Logical " (&&, ||, !) - AND, OR, NOT
4. Assignment " (+=, -=, *=, /=, %=)
5. Increment & decrement operators (++, --).
6. Conditional operator (? :)
7. Bitwise " (&, |, ^, <<, >>)
8. Special " (sizeof, &, *, .)

Expression is a sequence of operands & operators that reduces to a single value.

eg. (10+15) = 25
↓ ↓
expression value.

Special operators in c++

:: scope resolution operator
:: * pointer-to-member declarator
→ * " " operator
. * " " "

delete - memory release operator

new - memory allocation ,,

Scope resolution example

Syntax :: variablename. — access to global version of a variable

e.g.

```
#include<iostream.h>
int m=10;        // global
int main()
{
   int m=20;     // local to main
   {
      int k=m;
      int m=30;  // M declared local to inner block.
      cout<<" Inner block";
      cout<<" k is "<<k;
      cout<<" m is "<<m;
      cout<<" scope resolution "<<::m;
   }
   cout<<" Outer block";
   cout<<" M is "<<m;
   cout<<" Again m now is "<<::m;
   return 0;
}
```

o/p

Inner block

k=20

m=30

Scope resolution 10

Outer block
M is 20
Again m now is 10

In this program, m is declared 3 times/places (outside main(), inside main(), inside innerblock).
∴ m - refers to global m.
∴ m (in inner block) refers to 10.

eg2

delete [] p → delete the entire array pointed by p.

Operators precedence

Rank	Operator
1	::
2	→ . () [] ++ -- (postfix
3	prefix ++, --, ~, ! unary + -
4	&, sizeof, new, delete
5	→ * *
6	* / %
7	+ -
8	<< >>
9	< <= > >=
10	== !=
11	&
12	^
13	|
14	&&
15	||
16	?:

17	= *= /= %= += -=
18	<<= >>= &= ^= |=
19	, (comma operator)

Examples

a < b

10 < 20 - true

a+=1 → a=a+1

a*=1 → a=a*1

eg.

```
main()
{
    int a;
    a=2;
    while(a <100)
    {
      cout<<" value "<<a;
      a*=a;
    }
}
```

o/p

2

4

16

:

Increment & decrement.

m++ → m=m+1 (or) m+=1

m-- → m=m-1 (or) m-=1

Prefix operator → ++m

Prefix operator first adds 1 to operand and then the result is assigned to the variable on left.

e.g. m = 5;
y = ++m;
value of y = 6, m = 6

Postfix operator → m ++

Postfix operator first assigns the value to variable on the left and then increments the operand.

e.g. m = 5;
y = m++;
value of y = 5, m = 6.

Conditional operator

eg. a = 10;
b = 15;
x = (a > b) ? a : b;

10 > 15 → true so print a

if false, it will print b.

can be written as

if (a > b)
x = a;
else
x = b;

Logical

&& - logical AND, || - logical OR
! - logical NOT

if (age > 50 && salary < 1000)

- used to test more than one condition and make decision

Bitwise

- tests bit by bit

& bitwise AND
| " OR
^ " XOR
<< left shift
>> right shift.

e.g. x = 10

x << 1 = 10 x 2 = 18

x >> 1 = 10 / 2 = 5

Expressions

eg. x = a - b / 3 + c * 2 - 1.

a = 9, b = 12, c = 3.

x = 9 - 12 / 3 + 3 * 2 - 1

= 9 - 4 + 3 * 2 - 1

= 9 - 4 + 6 - 1

= 5 + 6 - 1

= 11 - 1

x = 10

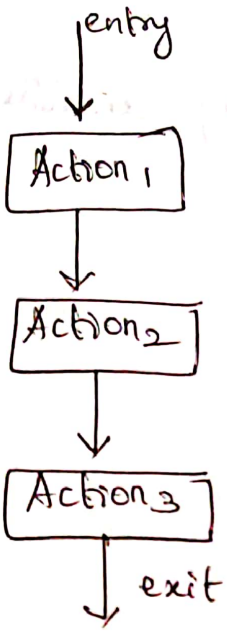
(preference)

If parenthesized means, first it is evaluated. g. (c*2) - parenthesized

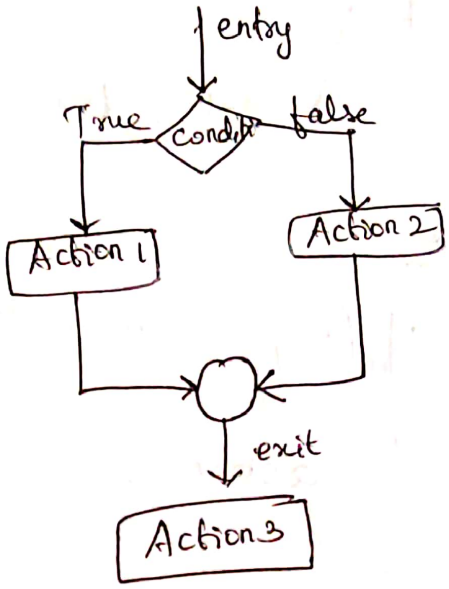
Control structures

There are 3 control structures to trace the flow of execution of statements.

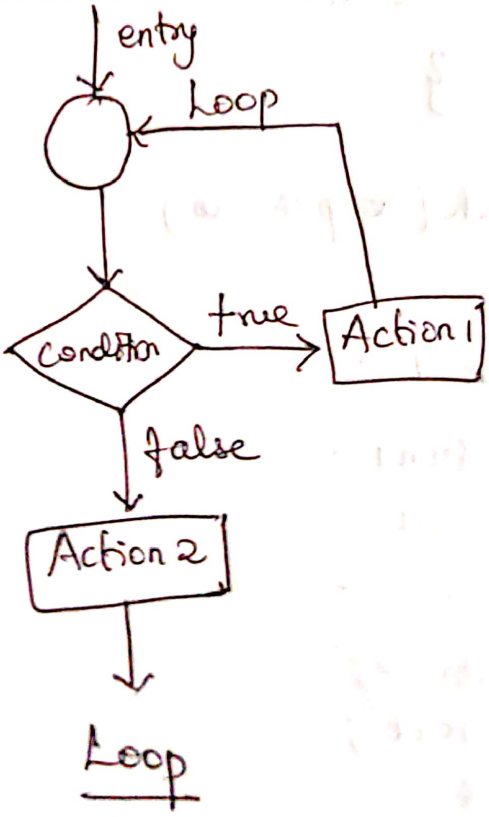
1. Sequence structure
2. Selection structure – branching statements
3. Looping structure – iteration "



Sequence



Selection



Loop

All program processing can be coded by using only these 3 logic structures.

- **Selection**
  - 2 way branch – if-else
  - Multiple branch – Switch
- **Loop**
  - Do while – exit controlled loop
  - while, for – entry controlled loop

Syntax

1)
```
if (expression is true)
{
    action1;
}
else
{
    action 2;
}
```

2)
```
Switch (expression)
{
  case 1 :
    action1 ;
    break;
  case 2 :
    action2;
    break;
  default :
    break;
}
```

3) Nested if else

```
if (condition 1)
{  if (condition 2)
   {  action;
   }
   else
   {  action 2;
   }
}
else
{  statements;
}
```

4) else-if ladder

```
if (condition)
==
else if (condition 2)
==
else if (condition 3)
==
else
==
```

5)
```
while (condition is true)
{
==
}
```

6)
```
do
{
 actions;
}
while (condition is
       true);
```

7) for (initial value ; test ; increment)
{
action 1 ;
}

8) break ;

Comes out of loop

9) Continue ; → Continues back to loop

eg1:
```
for (i=10 ; i<20 ; i++)
{
    if (i=15)
     break;
    cout<< "Value is" << i;
}
```

o/p Value is 10 11 12 13 14

eg2:
```
for (i=10 ; i<20 ; i++)
{
    if (i=15)
     Continue;
    cout<<"Value is " << i;
}
```

o/p Value is 10 11 12 13 14 16 17 18 19.

Value 15 is skipped

Example for switch

Calculator program

```
#include <iostream.h>
int main()
{
  char c;
  float num1, num2;
  cout << "Enter an operator +,-,*,/";
  cin >> c;
  cout << "enter 2 operands";
  cin >> num1 >> num2;
  switch(c)
  {
    case '+':
      cout << "Addition result is" << num1+num2;
      break;
    case '-':
      cout << "Subtraction result is" << num1-num2;
      break;
    case '*':
      cout << "Multiplication" << num1 * num2;
      break;
    case '/':
      cout << "Division" << num1 /num2;
      break;
```

```
    default :
        cout << " Enter operator correctly";
        break;
    }
    return 0;
}
```

Output

Enter an operator + - * /
+
Enter 2 operands : 25 35

Addition result is 60

UNIT-2

PROGRAMMING IN C++

2.1 CLASSES & OBJECTS

Class is an user defined data type with a template that serves to define its data properties.

* "A class is a template to bind the data and its associated functions together".

- it allows the data to be hidden from external use.

General structure

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

} class members

Note:

By default, the members of the classes are 'Private'.

"The binding of data and functions together into a single class-type variable is refered to as encapsulation".

example:

```
class Products
{
    int member;
```

```
    float cost;
  Public:
    void getdata(int a, float b);
};
```

Accessing class Members

To access the data of a class, member functions, objects are cesed.

Objectname . functionname (actualarguments);

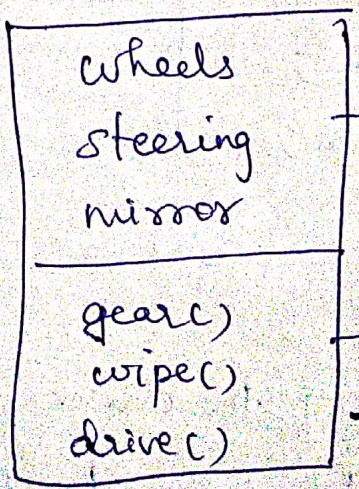
eg:- Ob . getdata (10, 15.6);

a b

Objects

* Objects are instances of classes which holds the data variables declared in class
* Objects are initialised using special functions called constructors.

vehicle – object



→ characteristics / behavior of object
– called data members
– actions done by object
– called member functions.

Member functions can be defined :

- Outside the class
- Inside the class

Outside the class definition

Syntax is :

```
return type classname :: functionname(arguments)
{
    function contents ;
}
```

:: scope resolution operator

eg void product :: getdata(int a, float b)

```
{
    num = a;
    price = b;
}
```

Inside the class definition

```
class Products
{ int num; float price;
  public :
    void getdata(int a, float b)
    {
        num = a;
        price = b;
    }
};
```

C++ program for classes and objects

```
#include<iostream.h>
class Car
{
    int wheels;
    float cost;
public:
    void getdetails(int w, float c)
    {
        wheels = w;
        cost = c;
    }
    void display()
    {
        cout<<"Number of wheels "<<wheels;
        cout<<"Cost of car "<<cost<<"\n";
    }
};
int main()
{
    Car c;
    c.getdetails(4, 758638.95);
    c.display();
    Car d;
    d.getdetails(4, 858354.89);
    d.display();
```

data members

object 1

Member functions called

object 2

Output

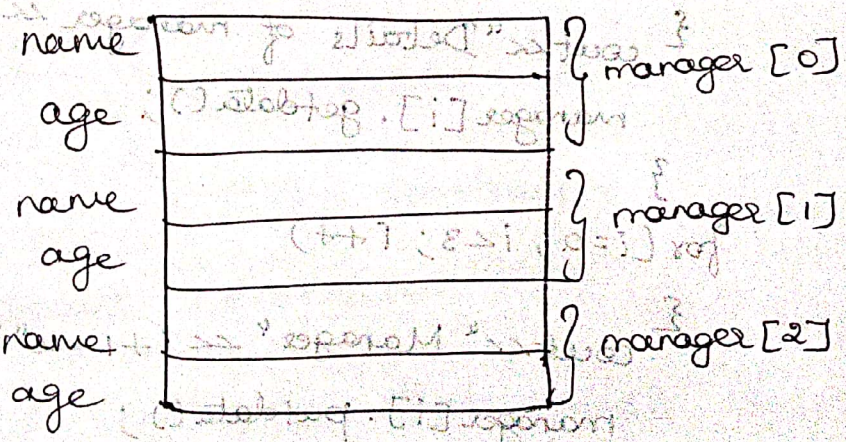
Number of Wheels 4
Cost of Car 758638.95

Number of Wheels 4
Cost of Car 858354.89

Arrays of Objects

Arrays of variables that are of the type class are called as arrays of objects

Array of objects are stored inside the memory in same way as a multidimensional array.



Program

```
class Employee
{
    char name[30];
    float age;
public:
    void getdata(void);
    void putdata(void);
};
```

```
void employee :: getdata(void)
{
    cout<<" enter name ";
    cin>> name ;
    cout<<" enter age ";
    cin>> age;
}
void employee :: putdata(void)
{
    cout<<"Name"<< name ;
    cout<<" Age"<<age;
}
int main()
{
    employee manager[3];
    for(int i=0 ; i<3 ; i++)
    { cout<<"Details of manager"<< i+1 <<"\n";
      manager[i].getdata();
    }
    for(i=0; i<3; i++)
    {
      cout<<" Manager "<< i+1 <<"\n";
      manager[i].putdata();
    }
    return 0;
}
```

O/P

```
Details of Manager1
Enter Name : Mr. XYZ
      Age  : 50
Details of Manager2
Name : Mr. PQR
Age : 55
Details of Manager 3
Name : Mr. LMN
Age : 35
Manger 1
Name : Mr. XYZ
Age : 50
Manager 2
Name : Mr. PQR
Age : 55
Manager 3
Name : Mr LMN
```

2.2 CONSTRUCTORS

A constructor is a special member function whose task is to initialize the objects of its class.

- it has the same name as the classname
- it is invoked whenever an object of its associated class is created.

* It is called as constructor because it constructs the values of the data members of the class.

Types
- Default
- Parameterized
- Copy constructor

Default constructor

A constructor that accepts no parameter is default constructor.

eg. for class A

```
A :: A()
```

- if suppose no compiler constructor is defined, compiler supplies a default constructor.

A obj ;

↳ invokes default constructor to create object 'obj'.

- 'initializes data members to zero'.

Characteristics

* They should be declared in public
* They dont have return types, no return values, no void
* They cant be inherited
* They make implicit calls to 'new' operator for memory allocation.

Parameterized Constructor

To initialize the various data elements of different objects with different values when they are created is important.

C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created.

The constructors that can take arguments are called parameterized constructors.

Program

```
class Point
{
    int x, y;
    public:
    Point(int a, int b) // parameterized constructor
    {
        x=a;
        y=b;
    }
    void display()
    {
        cout<<x<<y;
    }
};
int main()
{
    Point P1(5,6); // invokes parameterized constructor
    point P2(10,8); //    "        "
    Cout<<" Points P1 is ";
    P1.display();
    Cout<<" Points P2 is ";
    P2.display();
    return 0;
}
```

O/P

Points P1 is 5, 6

Points P2 is 10, 8

Copy Constructor

A copy constructor is used to declare and initialize an object from another object.

`A(A&);` → a constructor accepts a reference to its own class as a parameter.

↓
class name

Program

```
class code
{
  int id;
public:
 code (int a)          // parameterized constructor
  {
    id=a;
  }
 code( code &x)  // copy constructor
  {
     id = x.id;      // copy in the value
  }
 void display(void)
  {
    cout<<id;
  }
};
int main()
 {
    code A(100); // object A is created & initialized
    code B(A); // copy constructor
```

```
Code  C = A ;  // copy constructor again
cout << " id of A " ;
A.display();
cout << " id of B " ;
B.display();
cout << " id of C " ;
C.display();
return 0;
}
```

o/p

```
id of A 100
id of B 100
id of C 100
```

Note

When no copy constructor is defined, the compiler defines its own copy constructor.

`Code B(A);` → define the object B and at the same time initialize it to the values of A.

`code C = A;` → copy initialization.

- Process of initializing through a copy constructor is known as copy initialization.

4 OPERATOR OVERLOADING

The mechanism of providing a special meaning to the operators i.e. a flexible option for the creation of new definitions for operators is operator overloading.

E.g. We can overload '+' operator to add two complex numbers / vectors.

The operators that cannot be overloaded are:

* class member access operators (., .*)
* Scope resolution operator (::)
* Size operator (sizeof)
* Conditional operator (?:)

Syntax - Operator function

returntype classname :: operator op(args)
{
==
}

(keyword: operator; op: operator to overload)

Operator function is the special function which is used to define an additional task to an operator.

Overloading Unary Operators

Example - unary minus operator. It takes only one operand.

This operator changes the sign of an operand when applied to a data item.

To overload this operator, it can be applied to an object (in the same way as if to an int variable) & it change the sign of many data items

Program

```
class space
{
  int x, y, z;
 public :
    void getdata (int a, int b, int c);
    void display (void);
    void operator -();     // overload unary minus
};
void space :: getdata (int a, int b, int c)
{
    x = a;
    y = b;
    z = c;
}
```

```
void space :: display()
{
    cout << "x = " << x;
    cout << "y = " << y;
    cout << "z = " << z;
}
void space :: operator -()
{
    x = -x;
    y = -y;
    z = -z;
}
int main()
{
    space s;
    s.getdata(10, -20, 30);
    -s;  // calls the operator -() function to negate x, y, z
    cout << "Negated values are: ";
    s.display();
    return 0;
}
```

- takes no argument.
- it changes the sign of data members of the object s.

O/P

Negated values are:

X = -10

Y = 20

Z = -30

Overloading Binary Operators

- Overload '+' operator using operator+() function

Program

```
class Complex
{
    float x, y;
  public:
    complex(float real, float imag)
    {
        x = real;  y = imag;
    }
    Complex operator+ (Complex);
    void display(void);
};
```

overloading function

```
Complex complex :: operator + (complex c)
{
    complex temp;
    temp.x = x + c.x;        → receives one complex type argument
    temp.y = y + c.y;
    return(temp);            → returns complex value
}
```

```
void complex :: display(void)
{
    cout << x << "+j" << y << "\n";
}
int main()
{
    Complex c1, c2, c3;
    c1 = complex(2.5, 3.5);  // parameterized constructor call
```

```
c2 = complex ( 1.6, 2.7);
c3 = c1 + c2;        // operator - function called.
cout << " c1 = ";
  c1.display();
cout << " c2 = ";
  c2.display();
cout << " c3 = ";
  c3.display();
return 0;
}
```

o/p

c1 = 2.5 + j 3.5

c2 = 1.6 + j 2.7

c3 = 4.1 + j 6.2

Note

C_1 → object - invokes the function.

C_2 → argument passed to function.

`c3 = c1 + c2`

is equivalent to

`c3 = c1.operator+(c2);`

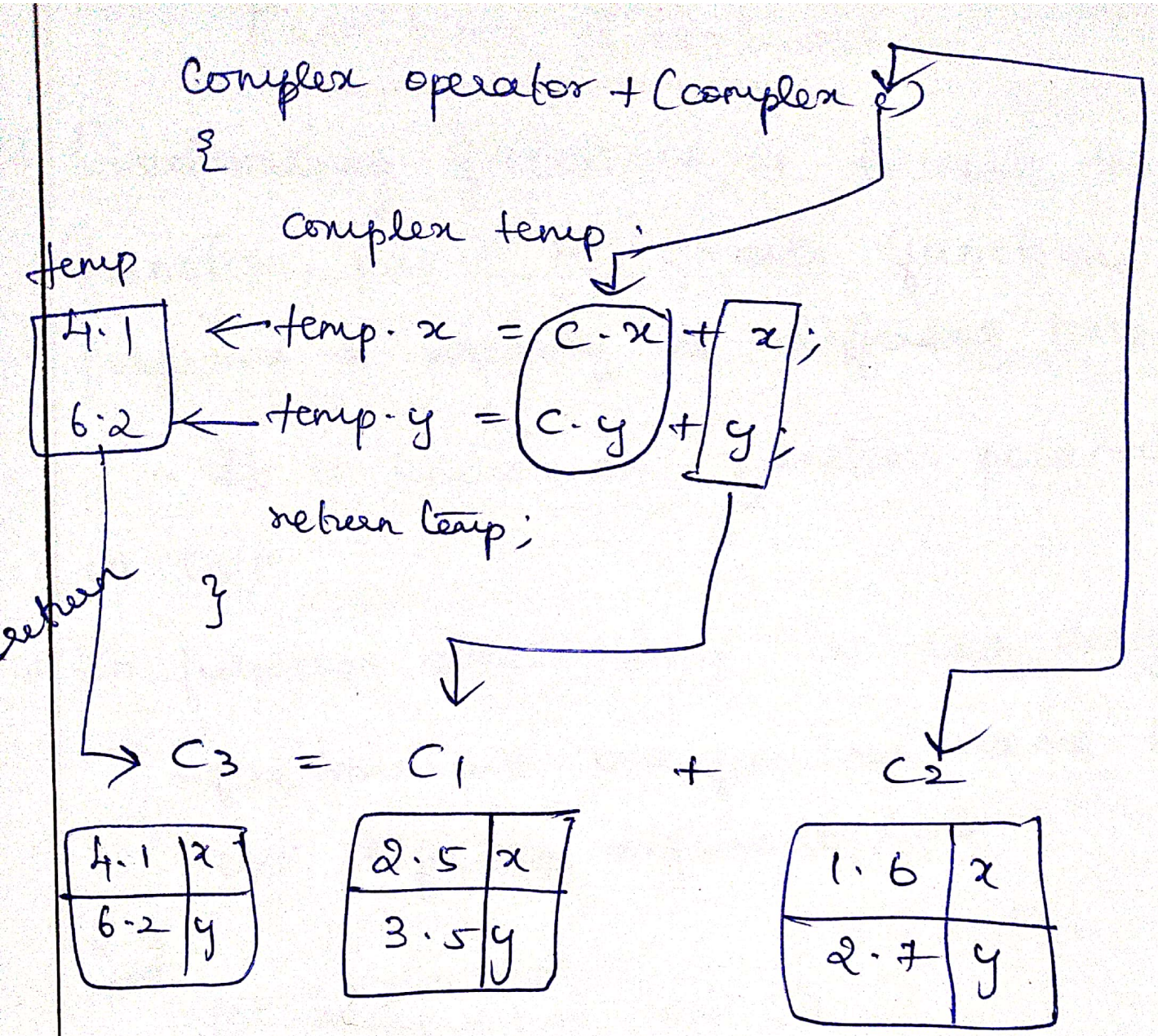
↓

operator function

Rule

Left hand operand - used to invoke the operator function

Right hand operand - passed as an argument.



Function Overloading

* Overloading functions is to use the same function name to create functions that performs a variety of different tasks.
  - It is known as function polymorphism in OOP.

* In function overloading, we can design
  - functions with one function name but with different argument lists.

e.g.

```
int add(int a, int b);
float add(float a, float b, float c);
double add(int p, double q);
```

// function calls:

```
cout << add(5,10);
cout << add(2.5, 10.5, 23.5);
cout << add(10, 0.75);
```

Function call will match the prototype (function declaration) which has same number & type of arguments and then it calls for functions execution.

Sample Program

Overloading function area() – 3 times.

```
#include<iostream.h>
int area(int);
int area(int, int);
float area(float);
int main()
{
  cout<<" Area of square"<< area(5);
  cout<<"Area of rectangle"<< area(5,10);
  cout<<" Area of circle"<< area(5.5);
  return 0;
}
int area(int side)
{
   return (side * side);
}
int area(int length, int breadth)
{
  return( length * breadth);
}
float area( float radius)
{
   return(3.14 * radius * radius);
}
```

O/p

```
Area of square 25
   "   rectangle 50
   "   circle 94.99
```

INLINE FUNCTIONS

Why we go for function?

The objective of using functions in a program is to save memory space which is useful when a function is to be called many times.

Problem

Every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, push arguments into stack, returning to calling function.

- if a function is too small, a substantial % of execution time may be spent in overheads

Solution

To eliminate the cost of calls to small functions, C++ has a new feature called inline function

An inline function is a function that is expanded in line when it is invoked.

* The compiler replaces the function call with the corresponding function code.

Syntax

```
inline functionname
{
  ==
}
```

e.g.

```
inline double cube(double a)
{
   return(a*a*a);
}
```

Inline functions will not work for :

* function returning values if a loop, switch exists.
* >> not >> >> if a return statement exists
* if functions contain static variables
* if inline functions are recursive

Program

```
#include<iostream.h>
inline float mul(float x, float y)
{
   return(x*y);
}
inline double div(double p, double q)
{
   return(p/q);
}
int main()
{
   float a = 12.35;
   float b = 5.85;
   cout<< mul(a,b);
   cout<< div(a,b);
} return 0;
```

INHERITANCE

The mechanism of deriving a new class from an existing class is called as inheritance.

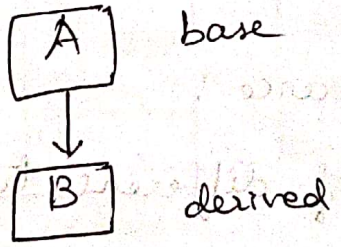
Existing class - base class
New class - derived class / sub class

* It supports the concept of reusability.

Types

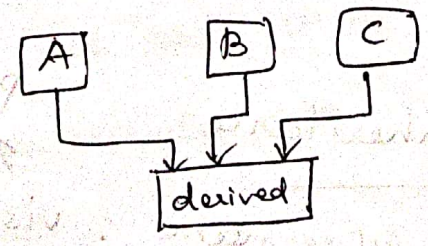
1. Single inheritance

A derived class with only one base class is called as single inheritance

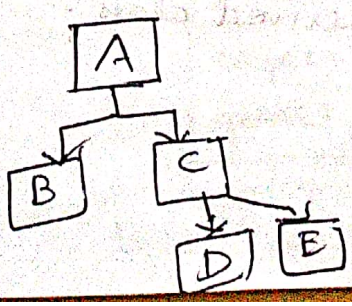


2. Multiple Inheritance

A derived class with several base classes is called as multiple inheritance.



3. Hierarchial Inheritance



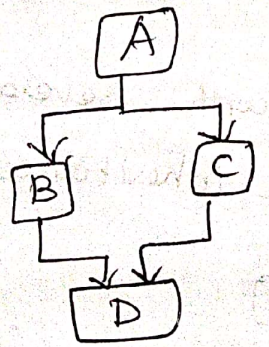
The base/parent class has more than one derived classes.

4. Multilevel Inheritance.

The mechanism of deriving a class from another derived class is known as multilevel inheritance.

5. Hybrid Inheritance:

Combination of different types of inheritance



Syntax for Inheritance

private, public, protected

class derivedclass name : visibility baseclass name

{

members of derived class;

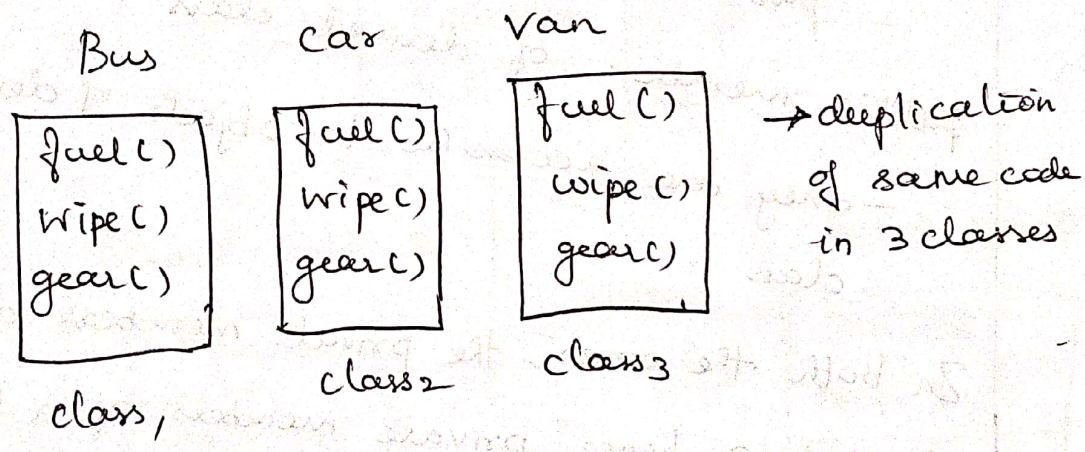
};

```
class derived : public base
{
    -members-;
};
```

Why we go for inheritance?

Consider a group of vehicles. We need to create classes for bus, car, van.

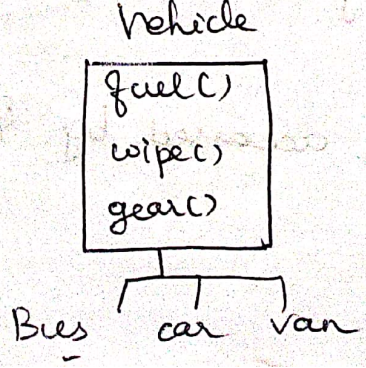
fuel(), wipe(), gear() - methods will be same for all the three classes.



→ deplication of same code in 3 classes

- Data redundancy increases.

To ↓ data redundancy & ↑ reusability of code we can write these three functions in a class vehicle and inherit the rest of the classes.



Base class when:

1. **Privately Inherited** by derived class –
   - public members of base – become private
   - " accessed by mem
   - functions of derive

Base class members is not accessible to obje of derived class.

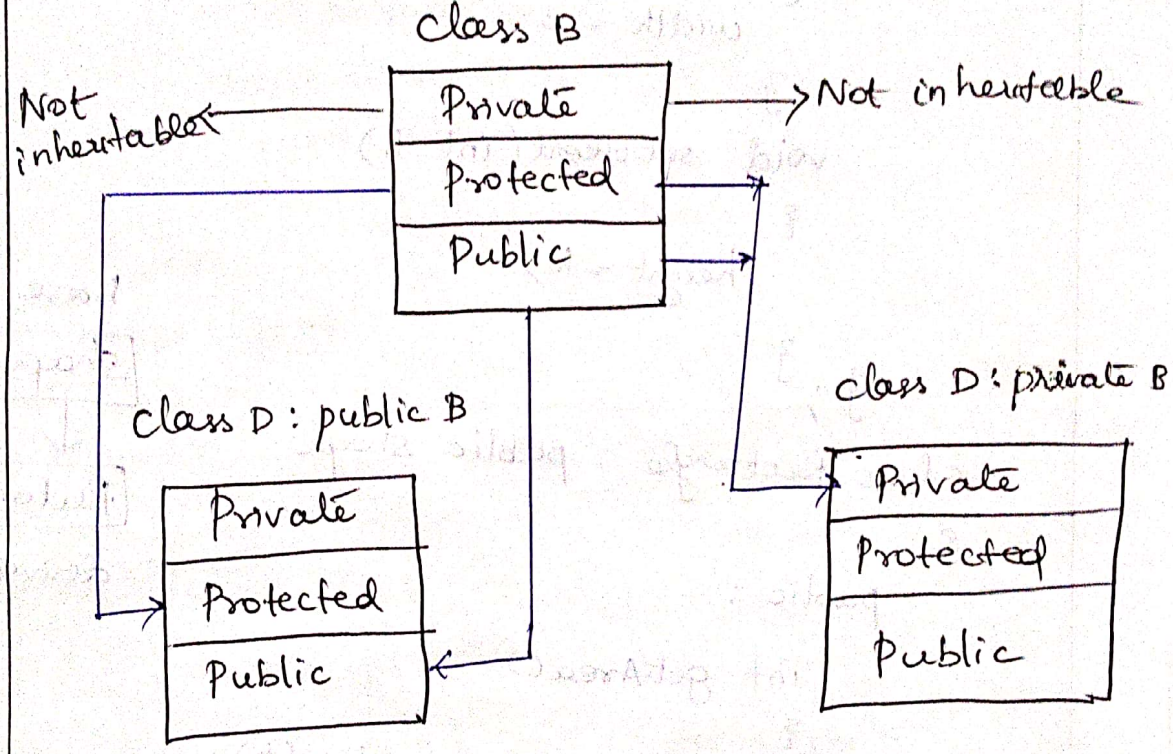
2. **Publicly inherited** –
   - public members of base class becomes public members of derived class
   - they are accessible to objects of derived class

In both the cases, the private members are no inherited & hence, private members of a base class will never become members of its derived class.

3. **Protected**
   - accessible by member functions within its class and any class immediately derived from it.
   - it cannot be accessed by functions outside these two classes.

Visibility of inherited Members

Base class visibility	Derived class visibility		
	Publicly derived	Privately derived	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected




```
class A
{
  Private :
    ==
  Protected :
    ==
  Public :
    ==
};
```

// visible to member functions within its class

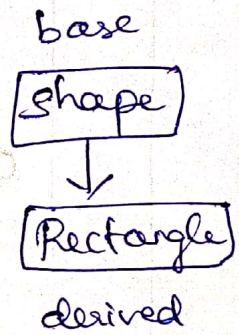
// visible to member functions of its own and its derived class

// visible to all functions in the program

Programs for all types of inheritance.

1. Single Inheritance

```
class shape
{
    protected:
        int width, height;
    public:
        void setWidth(int w)
        {
            width = w;
        }
        void setHeight(int h)
        {
            height = h;
        }
};
class Rectangle : public shape
{
    public:
        int getArea()
        {
            return (width * height);
        }
};
int main()
{
    Rectangle r;
    r.setwidth(5);
    r.setHeight(7);
    cout<<"Area is ";
    r.getarea();
    return 0;
```



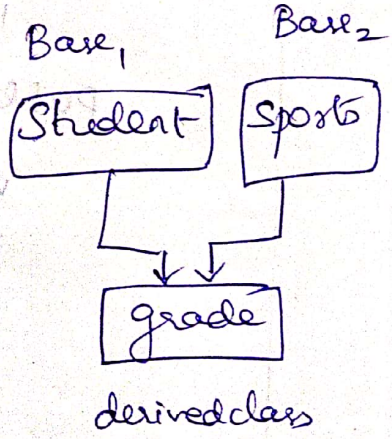
2. Multiple Inheritance

```
class student
{
   Protected :
      int rno, m1, m2;
   public :
      void get()
      {
         cout<<"enter rollno";
         cin>>rno;
        cout<<"enter 2 marks";
         cin>>m1>>m2;
   };
};
class sports
 {
    protected :
       int sm;
    Public :
       void getsm()
       {
          cout<<"enter sports marks";
          cin>>sm;
    };
};
class grade : public student, public sports
{
   int tot, avg;
   public :
     void display()
     { tot = m1 + m2 + sm;
```

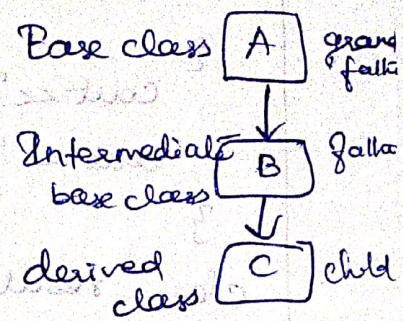
```
    avg = tot/3;
    cout<<" Roll number :"<<rno;
    cout<<"Average:"<<avg;
   }
};
void main()
{
    grade g;
    g.get();
    g.getsm();
    g.display();
   getch();
}
```

o/p

```
Enter roll no :100
Enter 2 mark
  90
  80
Enter sports mark : 90
Roll Number : 100
Total : 260
Average : 86.66
```



3. Multilevel Inheritance




```
Class Student
{
  protected :
     int rollnumber ;
  public :
     void getnumber(int);
     void putnumber(void);
};
void getnumber(int a)
{
   rollnumber = a ;
}
void putnumber ()
  {
   cout<< " Rollnumber is " << rollnumber ;
  }
};
Class Test : public student
{
    protected :
      float sub1, sub2 ;
    public :
      void getmarks (float, float);
      void putmarks (void);
  };
  void test :: get_marks( float x, float y)
    {
       sub1 = x ;
    } sub2 = y ;
```

```
void test :: putmarks()
{
   cout<<" Marks in subject 1 "<<sub1;
   cout<<" Marks in subject2 "<<sub2;
}
class result : public test
{
   float total;
  public:
    void display(void)
    {
       total = sub1 + sub2;
       putnumber();
       putmarks();
      cout<<" Total = "<< total;
    }
int main()
{
    result r;
    r. getnumber(111);
    r. getmarks(75.0, 85.5);
    r. display();
    return 0;
}
```

O/p

Roll number is 111
Marks in subject1 75.0
Marks in subject2 85.5
Total = 160.5

4. Hierarchial Inheritance:

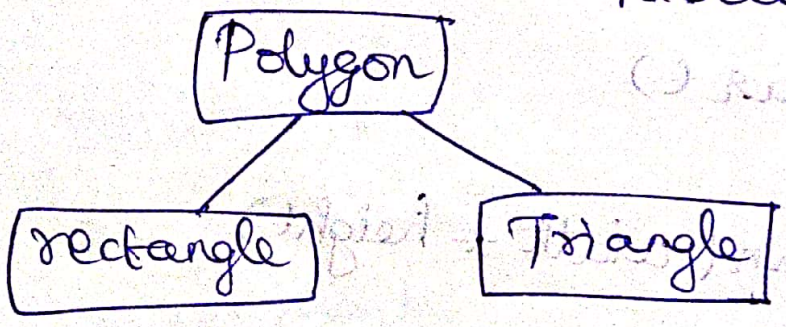
```
class Polygon
{
  protected:
    int width, height;
  public:
    void input(int x, int y)
    {
      width = x;
      height = y;
    }
};
class rectangle : public polygon
{
  public:
    int area()
    {
      return(width * height);
    }
};
class triangle : public polygon
{
  public:
    int areat()
    {
      return(width * height/2);
    }
};
```

```
void main()
{
  rectangle rect;
  triangle tri;
  rect.input(6,8);
  tri.input(6,10);
  cout<<"Area of Rectangle"<<rect.area();
  cout<<"Area of triangle"<<tri.areat();
  getch();
}
```

o/p

Area of rectangle : 48
Area of triangle : 30

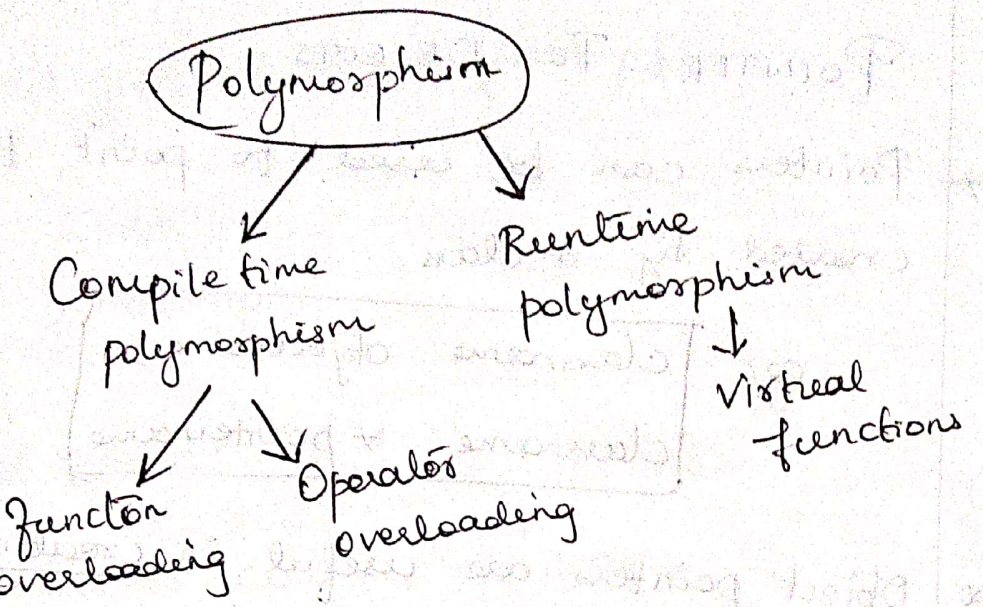
Hierarchial inheritance simple example



25 POLYMORPHISM

- one name, multiple forms.

* Polymorphism is implemented using the overloaded functions & operators.



* The overloaded member functions are selected for invoking by matching arguments both type & number.

- this information is known to compiler at compile time itself and the compiler is able to select the appropriate function for a particular call at the compile time itself.

This is called as "Early binding / static binding / static linking / compile time polymorphism".

Runtime polymorphism:

The appropriate member function could be selected while the program is running. It is done by the help of virtual functions

- The function is linked with a particular class only after compilation, hence this process is called as 'late binding' or 'dynamic binding'

* It needs the use of pointers to objects.

POINTERS TO OBJECTS

* Pointers can be used to point to an object created by a class.

eg.
```
classname objectname;
classname *pointername;
```

* Object pointers are useful in creating objects at runtime & access public members of an object.

class - student
object - s
pointer to object - *po

Syntax
```
student S;
student *po = &S;
```

eg.
```
S.getdata(100,150);
(or)
po -> getdata();
(or)
(*po).getdata();
```
} all are equal

```
#include<iostream.h>
class products
{
   int code;
   float price;
  public:
   void getdata(int c, float p)
    {
      code = c;
      price = p;
    }
  void show()
  {
    cout<<" code is "<< code;
    cout<<" Price is "<<price<<"\n";
  }
};
int main()
{
  products *p = new products;
  int x, i;
  float y;
  for(i=0; i<5; i++)
  {
    cout<<" Enter product's code & price";
    cin>>x>>y;
    p->getdata(x,y);
2   2  p->show(); p++;
```

used to create objects using pointer

o/p

Enter products code and price 40 500
50 600

code is 40
price is 500
code is 50
price is 600

Pointers to Derived objects / classes

* Pointers to objects of a base class are type compatible with pointers to objects of a derived class.

∴ A single pointer variable can be made to point to objects belonging to different classes.

```
Baseclass  *bptr;  → pointer to base class
Baseclass  b;      → object  "   "
Derivedclass d;    → object to derived class

*bptr = &b;
*bptr = &d;
```

} base class pointer stores base & derivedclass objects

Rule

*bptr - cannot access members that originally belong to derived class

*bptr - can access only those members that are inherited from B

* Base pointer cannot be directly used to access all the members of the derived class.

∴ Need to use other pointer to derived type.

```
#include<iostream.h>
class B
{
  public:
   int b;
   void show()
   {
    cout<<"Base class"<<b;
   }
};
Class D : public B
{
  public:
     int d;
  void show()
  {
    cout<<"Base "<<b;
    cout<<"Derived"<<d;
  }
};
int main()
{
  B *bptr;
  B Obj1;
```

```
bptr = &obj1;
bptr -> b = 100;
cout << "Base pointer points to base object";
bptr -> show(); // points to base class
D obj2;
bptr = &obj2;
bptr -> b = 200;
cout << "Bptr points to derived object";
bptr -> show(); // executes show() - base class
D *dptr;
dptr = &obj2;
dptr -> d = 300;
cout << "Derived type pointer";
dptr -> show();
return 0;
}
```

```
bptr -> d = 200 // won't work
```

bptr → base ptr
d → derived object

// create object for derived class to access derived class members

o/p

Base pointer points to base object
Base class = 100
Bptr points to derived object
Base class = 200
Derived type pointer
Base 200
Derived 300

When bptr is made to point to the derived object, show() displays contents of base class.

- Because base ptr can't directly access the members defined by derived class.

2.6 VIRTUAL FUNCTIONS

"The property by which objects belonging to different classes are able to respond to the same message but in different forms" - Polymorphism

- We need a pointer variable to refer to all classes.

* A base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class.

  - To solve this, we use 'Virtual functions'

* "When we use same function name in both base & derived classes, the function in base class is declared as 'virtual'".

* When a function is made virtual, C++ determines which function to be used at run time based on the type of object pointed to by the base pointer.

∴ Base pointer is made to point to different objects & we can execute different versions of virtual function.

```
#include <iostream.h>
class Base
{
  public:
    void display()
    {
      cout << "Display base";
    }
    virtual void show()
    {
      cout << "Base class's show method";
    }
};
class Derived : public Base
{
  public:
      void display()
      {
        cout << "Display derived";
      }
      void show()
      {
        cout << "Derived class's show method";
      }
};
```

```
int main()
{
  Base b;
  Derived d;
  Base *bptr;
  bptr = &b;
  bptr -> display();
  bptr -> show();
  cout<<"bptr points to derived";
  bptr = &d;
  bptr -> display();
  bptr -> show();
  return 0;
}
```

o/p

```
Display base
Base class's show method
bptr points to derived
Display base;  // calls display function of base class
Derived class's show method
```

because display() is not made as virtual.

"Run-time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class".

2.4 DATA ABSTRACTION

An abstract class is not used to create objects. It is designed only to act as a base class to be inherited by other classes.

- it is a design concept in program development and provides a base upon which other classes may be built.

Example:

```
class Vehicle    // abstract base class
{
  private:
    datatype v1, v2;
  public:
    virtual void spec() = 0;  // pure virtual function
};
class LMV : public vehicle
{
  public:
    void spec()
    {
      == definition of specifications for LMV
    }
};
class HMV : public vehicle
{
  public:
    void spec() { } ==
};
```

vehicle - base class is derived by LMV
HMV (light motor, heavy motor vehicle)

A pure virtual function spec() get specifications of LMV, HMV.

Here, ~~the~~ abstract base class vehicle is optional here, it holds the significance as far as logical program design is concerned.

* Abstract classes should have atleast one pure virtual function.

* Pure virtual functions is serving like a placeholder and are called as "do-nothing" functions.

```
Virtual void display() = 0;
```

* 'Virtual' - function declared as virtual inside base class & redefined in the derived class.

- in base class it is not used for performing any task.

A pure virtual function is a function declared in a base class that has no definition relative to the base class.

A class containing pure virtual functions

cannot be used to declare any objects. Such classes are called as abstract base classes.

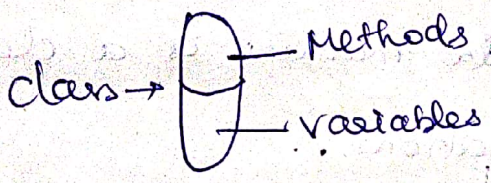
Objective of abstract base class:

- provides some traits to derived classes
- create a base pointer for achieving run time polymorphism.

Encapsulation

The wrapping up of data and functions together into a single unit is known as encapsulation.

It can be achieved using class keyword




```
class classname
{
    private:
        datatype data1, data2;

    public:
        Memberfunctions (arguments);

};
```

```
int main()
{
  classname object1, object2;
    object1. memberfunction();
}
```

Benefits

* Provides abstraction between an object and its clients
* Protects an object from unwanted access by clients

Real time Example

Automatic cola vending machine - We can request for a cola. The machine processes your request and gives the cola.

Here, automatic cola machine is a class. It has data and operations..

data → cola

operations → services done by machine

They are wrapped under a single unit cola vending machine. This is called as an Encapsulation.

2.8 TEMPLATES

Templates concept is used to define generic classes and functions for generic programming.

* "Generic Programming" is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types & data structures.

* A template can be used to create a family of classes or functions.

Example

A class template for an array class is used to create arrays of various data types such as int, char, float arrays.

A function template is used to create various versions of function for different type values

eg. mul() for multiplying int, float, double vales

Syntax

```
template <class T>
class classname
{
    ==
};
```

A class created from a class template is called a template class.

Syntax

```
Classname<type> objectname(arglist);
```

Example

```
template <class T1, class T2>
class Test
{
   T1 a;
   T2 b;
  public:
    Test (T1 x, T2 y)
     {
       a=x;
       b=y;
     }
    void show()
     {
       cout<<a<<"and"<<b<<"\n";
     }; }
int main()
{
  cout<<"Instantiating class template with int,
        float data types";
  Test <float, int> test1 (1.23, 123);
  test1.show();
  cout<<"Instantiating as test2 with int,
        char data types";
  Test <int, char> test2 (100, 'W');
} test2.show(); return 0;
```

o/p

Instantiating class template with int, float datatype

test1 : 1.23 and 123

Instantiating as test2 with int, char datatypes

test2 : 100 and W

Function Template syntax

```
template <class T>
returntype functionname(arguments of type T)
{
   ==
}
```

Example

```
template<class T>
void swap(T &x, T&y)
{
   T temp = x;
   x = y;
   y = temp;
}
void fun(int m, int n, float a, float b)
{
   cout<<" m and n before swap"<<m<<n;
   swap(m,n);
   cout<<" After swap"<<m<<n;
   cout<<" Swapping float values";
   swap(a,b);
} cout<<" After swapping float values"<<a<<b;
```

```
int main()
{
    fun(100, 200, 11.2, 22.5);
    return 0;
}
```

o/p

m and n before swap 100 200

After swap 200 100

Swapping float values 11.2 22.5

After swapping float values 22.5 11.2.

2.9 EXCEPTION HANDLING

Errors — Logic errors
Errors — Syntactic errors

Logic errors occurs due to poor understanding of the problem and solution procedure.

Syntactic errors arise due to poor understanding of the language itself.

* We can detect these errors by debugging.
* Some peculiar problems other than logic or syntax errors is known as exceptions.

Exceptions are runtime anomalies or unusual

Conditions that a program may encounter while executing.

Anomalies may be like

- division by zero
- access to an array outside bounds
- running out of memory space

* A built-in language feature to detect and handle exceptions which are basically run time errors is available in C++. It is called as Exception handling.

* Exception handling Mechanism

Basic keywords are:

* try
* throw
* catch

(1) Detect errors
(2) Throw exceptions
(3) Catch exceptions
(4) Take appropriate actions

} steps to handle.

* Try block

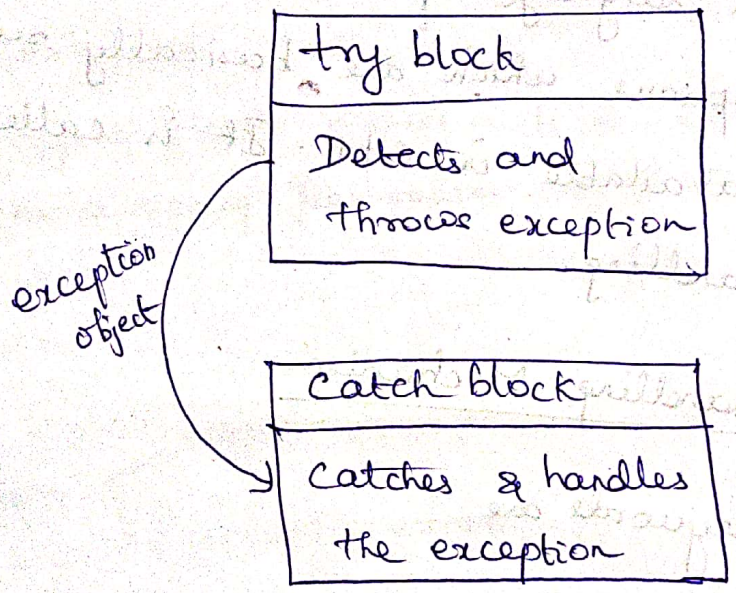
- used to preface a block of statements

which may generate exceptions.

* **Catch block**

When an exception is detected, it is thrown using a **throw** statement in the try block

Catch block catches the exception thrown by 'throw' statement and handles it effectively.




```
try
{
  ...
  throw exception;
  ...
}
Catch(type argument)
{
  ...
  ...
}
```

Program

```
#include<iostream.h>
int main()
{
  int a,b;
  cout<<"Enter values of a & b";
  cin>>a>>b;
  int x=a-b;
  try
  {
    if(x!=0)
    {
      cout<<"Result is"<<a/x;
    }
    else                    // there is an exception
    {
      throw(x);             // throw the int object
    }
  }
  catch(int i)              // catches the exception
  {
    cout<<"Exception caught: Divide by zero";
  }
  return 0;
}
```

O/p

```
Enter values of a & b
  15  10
Result is 1.5
```

Again execute the same program

o/p

Enter values of a & b

10 10

Exception caught Divide by zero

x = a - b
x = 10 - 10 = 0
so exception

Multiple catch statements

```
void test(int x)
{
  try
  {
    if (x==1) throw x;          // int
    else
      if (x==0) throw 'x';      // char
    else
      if (x==-1) throw 1.0;     // double
    cout<<"End of try-block";
  }
  catch(char c)
  {
    cout<<"Caught a character";
  }
  catch(int m)
  {
    cout<<"Caught an integer";
  }
  cout<<"End of try catch system";
}
```

```
catch(double d)
{ cout<<"double";
}
```

```
int main()
{
    cout<<"Testing multiple catches";
    cout<<" x==1";
    test(1);
    cout<<" x==0";
    test(0);
    cout<<" x==-1";
    test(-1);
    cout<<" x==2";
    test(2);
    return 0;
}
```

O/P

```
Testing Multiple Catches
x==1
Caught an integer
End of try catch system
x==0
caught a character
End of try-catch system
x==2
End of try block
End of try-catch system
```

* Here, multiple catch statements are used to handle various types of exceptions

* When the try block does not throw any exception and it completes normal exception, control passes to the first statement after the last catch handler associated with that try block.

Note

'try block' does not throw any exception when test() is invoked with x = 2.

2.10 C++ Stream Classes – I/O Operations

C++ uses the concept of 'stream' & 'stream classes' to implement its I/O operations with console and disk files.

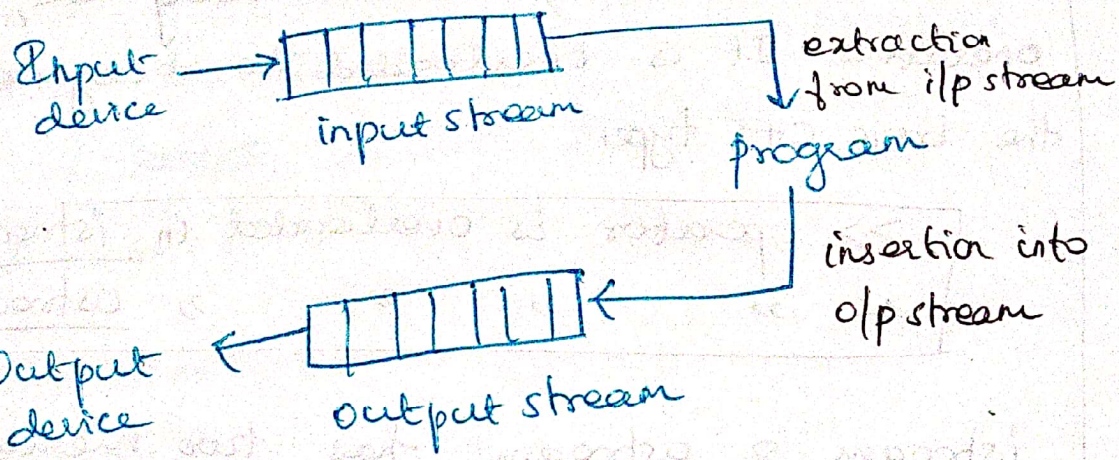
* C++ designed to work with a wide variety of devices including terminals, disks & tape drives

- I/O system supplies an interface to the programmer that is independent of the actual device being accessed.

This interface is known as "stream".

* A stream is a sequence of bytes.

* The destination stream that receives output from the program is called as output stream

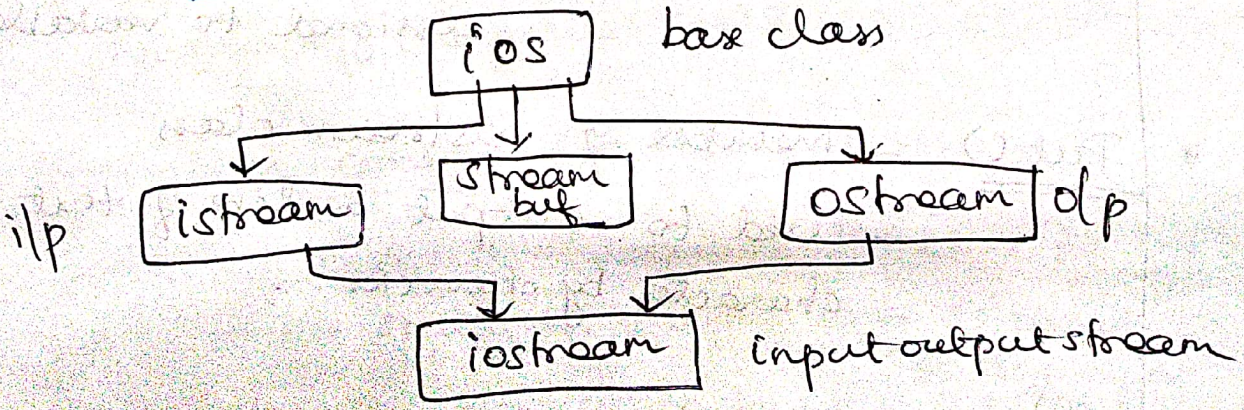


* cin - represents the input stream connected to standard input device (keyboard)

* cout - represents the output stream connected to standard output device (screen) / (files)

C++ stream classes

* The hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes.



Stream classes for console I/O operations

streambuf - provides an interface to physical devices through buffers.

* In cin & cout, we are using <<, >> operators. It is overloaded to recognize all the basic C++ types.

> >> operator is overloaded in istream class
> << " " " " ostream "

* istream & ostream has two member functions get(), put() to handle the single character i/o operations.

get() — get(char *) - fetch character including blankspace tab & newline
get(void)

eg1

* cin.get(c); → get a character from keyboard & assign it to c.

eg2

```
char c;
c = cin.get();
```
- value returned by the function get() is assigned to variable 'c'.

* Put() - member of ostream class
- used to output a line of text character by character.

eg1

```
cout.put('x');
```

eg2

```
cout.put(ch);
```
→ value of variable 'ch'.

cout.put(68); put() converts
↳ int value 68 to char value & display character whose ASCII value is 68.

Read character & display - C++ program
(character I/O with get() and put())

```
int main()
{
  int count = 0;
  char c;
  cout << "Input text";
  cin.get(c);
  while(c != '\n')
  {
    cout.put(c);
    count++;
    cin.get(c);
  }
  cout << "Number of characters" << count;
  return 0;
}
```

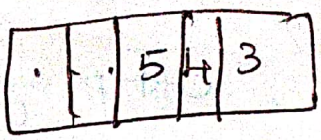
O/p Input text : Object Oriented Program

put(c) → Object Oriented Program
Number of characters = 27

Formatted console I/O operations

width() - specifies required field size for displaying o/p value
precision() - specify no. of digits to be displayed after decimal point
fill() - specify character to be filled for unused portion
setf() - format flags to control o/p display
unsetf() - clear flags

eg1. cout.width(5);
cout<<543;



right justified in first 5 columns.

eg2

cout.precision(3);
cout<<sqrt(2);

1.141 - rounded value

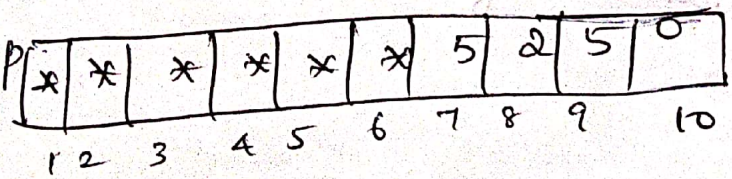
3 digits after decimal point.

eg3

cout.fill('ch'); → represents the character used for filling unused position

cout.fill('*');
cout.width(10);
cout<<5250;

o/p



width = 10.

eg4

cout.setf(arg1, arg2);

formatting flags specifies format action required for o/p.

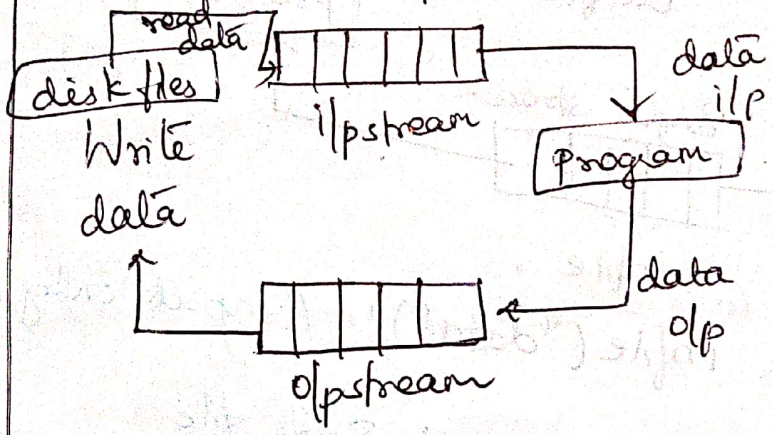
cout.setf(ios::left, ios::adjustfield);

↓ left justified o/p ↓ bitfield

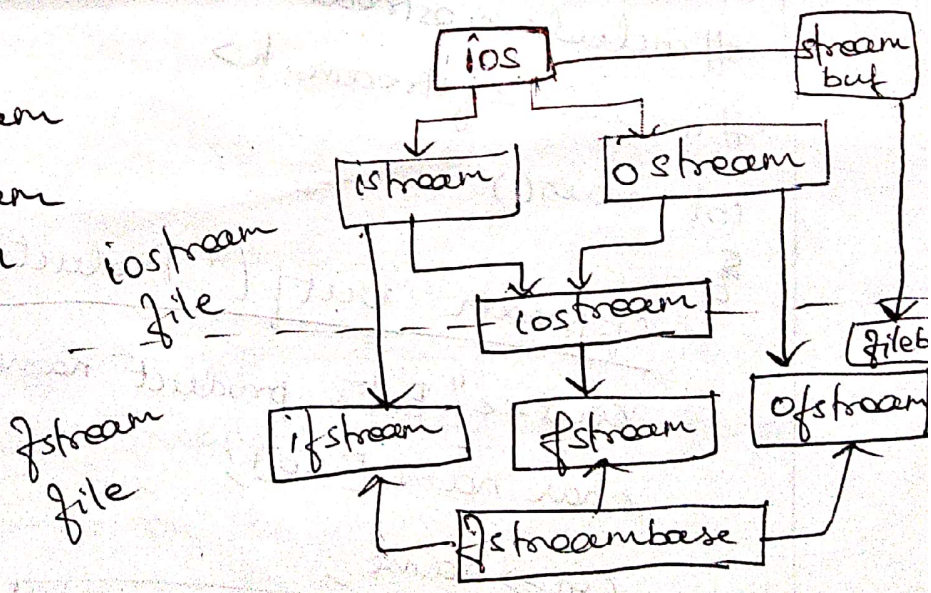
File Classes & Operations

A file is a collection of related data stored in a particular area on the disk.

* Programs can be designed to perform the read & write operations on these files



* A set of classes that define file handling methods.
  * ifstream
  * ofstream
  * fstream



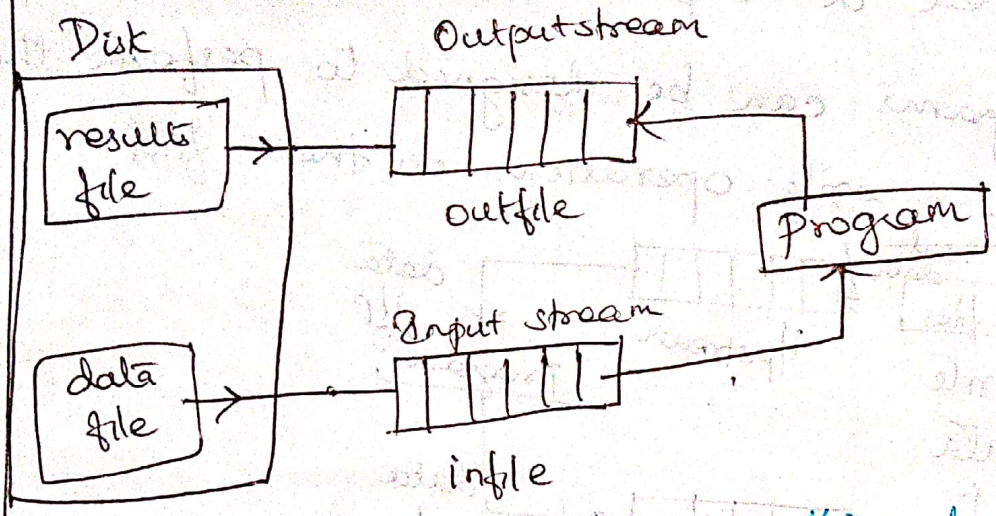
* A file can be opened in 2 ways:
  1. Using constructor function of the class
  2. Using the member function open() of the class.

* ofstream - used to create output stream
  ifstream - create the input stream

e.g. ofstream outfile("results"); // for o/p

↓ object of ofstream

↳ file



eg. ifstream infile("data"); // input only.

Program for working with Single file

```
#include<iostream.h>
#include<fstream.h>
int main()
{
    ofstream outf("Products");
    cout<<"Enter product name";
    char name[30];
    cin>>name;
    outf<<name<<"\n";          // write to file Products
    cout<<"Enter cost";
    float cost;
    cin>>cost;
    outf<<cost<<"\n";          // write to file products
    outf.close();              // disconnect products file from outf
```

```
ifstream inf("Products"); // connect products file to inf
inf >> name;
inf >> cost;
// read name from file products
cout << "Product name & cost :- " << name << cost;
inf.close();
return 0;
}
```

// above program is creating file with constructor function.

// A single file is used for both reading and writing the data.

* In the progeam, it takes data from the keyboard and writes it to file. After writing is done, the file is closed.

* The program again opens the same file, reads the information already written to it and displays the same on the screen.

Opening files using open()

- open() used to open multiple files that use the same stream object.

eg. ofstream outf;
outf.open("file1");

```
#include <fstream.h>
int main()
{
    ofstream fout;
    fout.open("country");
    fout<<"UK";
    fout<<"South korea";
    fout<<"Canada";
    fout.close();
    fout.open("capital");
    fout<<"London";
    fout<<"Seoul";
    fout<<"Ottawa";
    fout.close();
  const int N=80;
  char line[N];
  ifstream fin;
  fin.open("Country");
  cout<<"Contents of country file";
 while(fin)
   {
      fin.getline(line,N);
      cout<<line;
   }
 fin.close();
 cout<<"Contents of capital country file";
 while(fin)
 {
```

```
    fin.getline(line, N);
    cout<<line;
  }
  fin.close();
  return 0;
}
```

O/P

Contents of country file

Uk
South korea
canada

Contents of capital file

London
Seoul
Ottawa

Detecting end-of file condition is necessary for preventing any further attempt to read data from file.

```
if(fin1.eof()!=0)
{
  exit(1);
}
```

eof() - member function of ios class.

The above statement terminates the program on reaching end of file