PPL

C++ by Detal & Detal

→ Set of execuitible set of instruction is called computer programm.

→ High level prog lang. → Understandable to human

High level lang. (in En, Hi) ⟶ Compilation ⟶ low level prog. (in 0 & 1)

Diff. prog. domain :-

→ Scientific applications :-

① It requires relatively simple data structure but large no. of computions.

② The most common data structure was arrays & matrices & The most common control structure were loops.

③ The first scientific application is fotran (1957)

④ These require floating point operation.

→ Buisness application :-

① Producing report

② Precise way of describing & storing decimal numbers & character data.

③ The ability to perform decimal arithametic operation

④ First lang. is COBOL (1959)

→ System programming :-

① Operating system and the programming support tools of a computer are collectively known as system. s/w

② The most system software are now written in C/C++

③ The unix operating system is written in C (1972)

→ Internet / Web based application :-

① Multiple layer application ⓐ Frontend ⓑ Logic ⓒ Backend.

② For each layer / level we use different language.

③ The world wide web uses makup language such as HTML (1993) and general purpose prog. language such as JAVA (1995)

④ Dynamic pages.

⑤ Scripting.

→ Artificial intelligence :-

① First we created LISP (1958) as it uses symbolic rather than numeric computation.

② Symbolic computation means that symbol consisting name rather than the number are manupilated.

③ Symbolic computation is more conveniently done with linked list of data rather than array.

Types of programming language

① Imperative programming language

→ In this program specifies how it is to be done step by step.

→ It describes the control flow of computation.

→ The programming languages are:

procedural programming languages

Object oriented prog. languages.

② Declarative programming languages :-

→ In this programs specify what is to be done

→ It simply express logic of computation

Ex :- ① Logic prog. lang.

② Database prog. lang.

① Procedural programming language :-

→ A procedural language relies on well organize procedure or function or sub-routines in a program.

→ programs are made up of modules which are parts of a program that can be coded tested seprately and then assembled to form the complete program.

→ A procedure is implemented on data to perform a task, in procedural programming, data and funct. are stored in seprate memory location.

→ Can't handle real world problem easily.

② Object-oriented programming languages :-

→ In Object-oriented programming data & functions these two concepts are bundled into objects this makes it possible to create more complicated behaviour with less codes.

→ The use of objects increase the reuseability of code.

→ In OOProg. data is hidden from external functions.

→ Object's may communicate with each other through functions.

③ Logic programming language :-

→ Logic is the study of formal reasoning based upon statements of propositions.

→ Any program written in a logic programming lang. is a set of sentences in logical form, expressing facts & & rules about some problem domain.

Ex :-

Why PPL?

→ Increased capacity to express ideas.

The language in which programmers develope software places limit on the kind of control structure, data structure & abstractions they can use, thus the form of algorithm they can construct or likewise limited.

Awareness of wider varity of programming language feature can reduce such limitations in software developement.

Ex Use class in case of storing student data in place of multiple array.

→ Improved background for choosing appropriate language.

For many old professionals the prog. lang. they learned may be no longer used & many feature are now avalible in prog. languages were not known at the time they learned it, if that is the case then many prog. when given a choice of languages for new project, use the language with which they are the most familier even if it is poorly suited for the project.

→ Increased ability to learn new languages :-

Computer progamming is a still relatively young decipline & design methodology, software developement tools & prog. lang. are in the state of continuous evolution, the process of learning a new programming language can be lengthy & diff. Once a thourogh understanding of prog. lang. is acquired it becomes far easier to see how these concepts are incorporated into the design of the language being learned.

Ex:- Those who knows about OOP's can learn java easily

→ Better understanding of significance of implementation :-

An understanding of why languages are designed the way they are leads the ability to use a language more intelegen-tly.

We can become better programmer by understanding the choice among programming language construct & consequence of those choices.

→ Better use of languages that are already known.
Many programming languages are large & complex so it is uncommon for a programmer to be familier with all the features of a language, by studing the concept of progra-ming languages programmers can learn about previously unknown the part of the language they already use.

Ex. learning object oriented features of C++.

→ Overall advancement of computing
Many belives the most popular programming language are not always the best avalible for your problem.
A lang. become widely used bcz those in position to choose lang. were not sufficiently familier with prog. lang. concept.

Ex. Using Java for scientific application.

Ex Many people belive that if ALGOL 60 has replaced the potron.

Language evaluation criteria :-

	→ Readability	→ Writability	→ Reliability
	R	W	Reli
(1) Simplicity	✓	✓	✓
(2) Orthogonality	✓	✓	✓
(3) Data types	✓	✓	✓
(4) Syntax design	✓	✓	✓
(5) Support for abstraction		✓	✓
(6) Expressivity		✓	✓
(7) Type checking			✓

(8) Exception handling ✓
(9) Restricted aliasing. ✓

Readability:-

Simplicity →

→ A lang. with large number of basic constructs is more diffi-cult to learn than one with smaller number.

→ Readibility problem occours whenever the program authour has learned different subset from the subset with which the reader is familier

→ features multiplicity (take C++)
count = count+1; count+=1 ; count ++ ; ++ count
reduces readability.

→ Operator overloading
In python + toadd & to concatinate the string

Orthogonality →

→ Orthogonality in a programming language means that a relatively small set of premittive construct can be combined in a relatively small number of ways to build the control and data structures of the language.

→ Every possible combination of primitives is legal and meaningful.

→ The example of lack of orthogonality in C prog. lang.

- C has two kind of structure data type i.e arrays & structs, structs can be written from a function but not array's
- A member of struct can be of any data type except void & a structure of the same type.
- To a function parameter are passed by value unless they are array.

→ If orthogonality of a prog. lang. is very high it also reduces the readability.

Data type →

→ The presence of sufficient facilities for defining data types and data structures in a language is always a significant aid to readability.

- Suppose a prog. lang. does not have boolean data type in that case to define a flag we use an integer which is not that much meaningful as that of boolean data-type.

Syntax design →

→ Special words :-

- Program readability is affected by the form of language special words. for ex:- while, for, if.
- Which type of special words or syntax we are using to define a block or group of statement in a prog. lang.

→ Form and meaning:-

- Designing statements so that their presence indicate their purpose, always increases readability

Writability :-

All features affecting readability will also affect writa-bility.

Support for abstraction →

Abstraction mean the ability to define and then use complicated structure and operation in the way that permit many of the details to be ignored.

There are two types of abstraction

1. Process abstraction
2. Data abstraction

→ A simple example of process abstraction is the use of a sub program to implement a sorting algorithm that is

required multiple times in your program.

→ An example of data abstraction is a binary tree that stores integer data in it nodes, this tree can be implemented by using abstraction of the tree node in the form of a simple struct with two pointers and an integer.

Expressivity →

→ There are different characterstick that affects the expressivity of language

① The language is having very powerful operator that perform a great deal of computation with very small program.

② Language should have relatively convinient way of specifing computation like in C-lang Count++ is more expressive as compared to count = count + 1.

Reliability →

→ Type checking :-

- It is simply testing of type error in a given program either by compiler or during program compilation.
- Compile time type checking is more preffered ex : JAVA
- In the initial version of the C it was not having the typecheck facility during the function call actual parameters type were not checked against the formal parameter.

→ Exception handling :-

- The ability of a program to intercept run-time error & take corrective measure and then continue
- Ex: JAVA

→ Alaising :-

- Is having two or more distinct name that can be used to access same memory location

- Most famous way is using pointer, multiple pointer point the same address.
- It reduces relaibility of prog. hence JAVA didn't have concept of pointers.

Cost of selecting a programming language

→ Their is cost of training programmers to use the language which is the function of simplicity and orthogonality of the language.

→ Their is a cost of writing programms in the language this is the function of writability of language & also depend upon domain of the application.

→ Cost of compiling program in the language

→ Cost of executing a program in the language.

- Their is a tradeoff between the compilation cost & execution speed of the compiled code.
- Optimization is the technique used by compiler to decrease the size of the code and increase the execution speed
- If you are using the code in the production environment so its good to pay extra cost of optimization.

→ Cost of language implementation system.

- for MATLAB we need to pay while JAVA is free.

→ Cost of poor reliability

- If the software fails in a critical system such as nuclear power plant or an X-ray machine for medical use, the Cost will be very high.

→ Cost associate with maintainence of the program

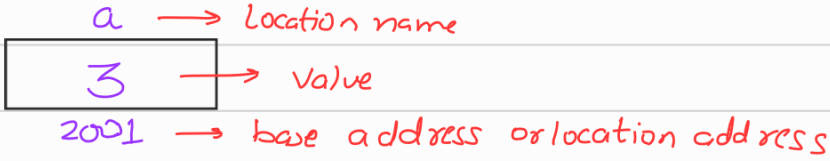
Pointer →

Consider int a = 3;

→ Reserve space in memory to hold the integer value.

→ Associate name a with this memory location.

→ Store the value 3 at this location



→ A pointer is a variable which holds the memory address of another variable.

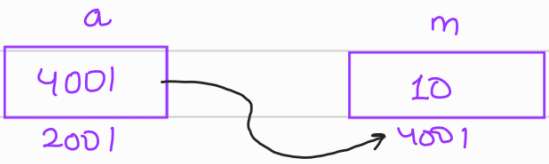
- Pointer operator
- Address operator

→ A pointer operator can be represented by a combination of * with a variable & a data type is associated with it.

Syntax → data_type * ptr_name;

→ Address operator can be represented by a combination of & with a variable name.

Ex
```
int m = 10;
int *a;
a = &m;
```



Pointer expressions →

```
int x, y;
int* ptr1, *ptr2;
ptr1 = &x;
y = *ptr1;
ptr2 = ptr1;
```

Null pointer →

A null pointer is a methode in which a pointer variable is guranted not to point to a valid object.

```
int * ptr;
ptr = NULL;
```

Pointer Arithmetic →

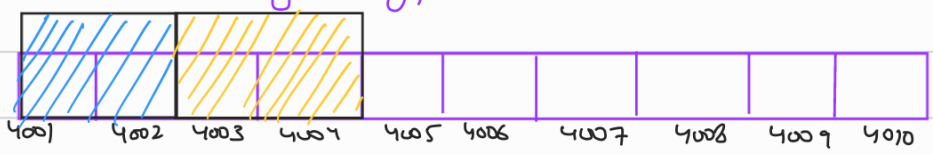
Addition +

Subtraction −

Incrementation ++

decrementation − −

→ When a pointer is incremented it points to the immediately next location of its type.




```
int * i = j ;
```

(for ex: &j = 4001;) Say int take 2 byte

```
i = &j;
i++;
```
← Immediate next integer add.

Dekh lo

*ptr++ and ++*ptr

Pointers & Arrays →

→ Array is a collection of elements of similar data type stored in adjacent memory location.

→ Pointer & 1D Array

```
int A[10];
int * ptr;
```

A++ or A+5 is not valid as environment will forget base address.

23	34	12	56	17	19	53	62	7	49

4001 ↑ 4019

```
ptr = A;
// A[0] = *(A+0)
```
→ System understand like this

The following notation are of same type.

A[i] = *(A+i) or i[A] = *(i+A)

Multi dimensional array :-

→ In memory wheather its a 1D or 2D Array the elements are stored in a contineous chain each row of a 2D array can be treated as 1D array.

Ex int A[5][2] = { {10,20}, {30,40}, {50,60}, {70,80}, {90,100} }

10	20	30	40	50	60	70	80	90	100
4001	4003	4005	4007	4009	4011	4013	4015	4017	4019

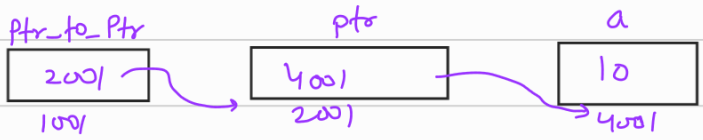
A[0][0] = 10
1st row 1st element

A[4][1] = 100
4th row 2nd element

A[4][1] = * (*(A + 4) + 1)

Pointer to pointer :-

```
int a = 10;
int * ptr ;
int ** ptr_to_ptr ;
 ptr = & a;
ptr_to_ptr =& ptr;
```



Array of pointers :-

```
int *arr[4];
int i = 31, j = 5 , k = 19, t = 71, m ;
 arr[0] = & i;
 arr[1] = & j;
 arr[2] =  & k;
arr[3] = & L;
for (m = 0; m <= 3; m++ )
    cout << * arr[m];
```

Pointer to array :-

int (* ptr)[4] → Pointer to an address of size 4.

Ex: int A[3][4] = { {1,2,3,4} {5,6,7,8}, {9,10,11,12} , }

int (* ptr) [4]

(*p)[i] → interal array travel
*p[i] → external array travel.

```
ptr = A;
cout << **ptr;
ptr++;
cout << **ptr;
```

Pointer and string :-

→ A string is a one dimensional array of character terminated by null or '\0'.

```
char Name = {'I','N','D','I','A'}
    cout << Name[2];
    cout<<Name;  //Print whole string.
```

→ String is any series of characters enclosed in double quotes

→ The compiler automatically appends a null character to the end of string.

Ex char Name[4] = "INDI"; → X

→ We must allocate enough characters to hold the string if one want to specify an array size of declaring string.

```
char Name[6] = "INDIA"
char *ptr;
ptr = Name;
cout << ptr;
while (*ptr != '\0')
  {
    cout << *ptr; // This will display INDIA.
      ptr++;
  }
```

Q. Write A program to copy the content of 1 string to another using pointers?

Sol.

```
{
  char source[20], dest[20];
  source [20] = "INDIA";
  char * source_ptr, *dest_ptr;
   source_ptr = source;
 while (*source_ptr != '\0') {
  *dest_ptr = *source_ptr;
        dest_ptr++;
        source_ptr++;
   }
  *dest_ptr = '\0';
   dest_ptr = dest;
 while (*dest_ptr != '\0')
   {
    cout << *dest_ptr;
     dest_ptr++;
   }
```

Q. Write a program to concatinate given two strings into 1 string using pointer?

```
{
 char source 1[20], source2[20], destination [80];
 source 1 [20] = "INDIA";
 source 2[20] = "COUNTRY";
 char * source1_ptr, *source2_ptr, *dest_ptr;
 source1_ptr = source1;
 source2_ptr = source2;
 dest_ptr = dest;
 while (*source1_ptr != '\0')
   {
     *dest_ptr = *source1_ptr;
       dest_ptr++;
   }   source1_ptr++;
```

```
    *dest_ptr = ' ';
      dest_ptr++;
    while (*source2_ptr != '\0')
      {
        *dest_ptr = *source2_ptr;
          dest_ptr++;
          source2_ptr++;
      }
}
```

Variable storage duration & scope →

The attributes of a variable includes name, type, size, value possible & each variable has some other attribute ie scope & storage duration.

Storage duration →

An identifier storage duration determines the period during which that identifier exists in memory.

1. Some variable exist for very short time.
2. Some are repeatedly created and destroyed.
3. Some var. exists during whole execution time of program.

So there are 4 type of storage duration.

1. Automatic
2. Static
3. Dynamic
4. Thread.

Automatic storage duration →

→ Such variable are created when program execution enters a block in which they are defined, they exist while block is active

→ An automatic variable exists in the nearest enclosing pair of curly braces within the body of the function in which the definition present or entire function body in the case of

function parameter.

→ Variable having Auto. storage duration
local variable declared in function, function parameter.

Static storage duration →

→ Variable with static storage duration exist in memory from the point at which the program start execution & untill the program terminates.

→ Such a variable is initialized once when its declaration is encountered.

→ Two types of variable with static storage duration.

1. Global variable
2. Static local variable.

→ local variable declared static are still known (or accessable) only in the function in which they are declared, but unlike automatic variable, static local variable retains their value when the function returns to caller, the next time function is called the static local variable contains the value they had when function complete execution last time.

Dynamic storage duration →

→ A local variable can have dynamic storage duration if it storage is allocated by calling an allocation function using malloc in C & new in C++;

Ex *ptr = (int *) malloc(10 x Sizeof (int))

→ A variable with dynamic storage duration the lifetime of its storage last untill the memory is deallocated explicitly i.e using free in C & delete in C++.

Scope of variable →

An identifier scope is where the identifier can be refferenced in a program.

1. Block scope
2. Global scope
3. The function prototype scope.

Block scope →

Identifiers declared inside a block have the block scope, which begins at the identifiers declaration & ends at the terminating at the right curly brace of the block in which identifiers is declared.

→ local variable & function parameter have block scope.

→ When blocks are nested and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is hidden until the inner block terminates.

Global scope →

→ An identifier declared outside any function or class has global scope.

Ex Global variable, function prototype declaration, function defination

→ Such an identifier is "known" in all functions from the point at which its declared until end of the file.

Function prototype scope →

→ The only identifiers with function prototype scope and those used in parameter list of a function prototype.

Ex void fun(int b).

```
Void uselocal( );
Void usestaticlocal( );
Void useglobal( );
  Int n=1;
```

```
main() {
  cout<< "Global x =" << x;
   int x = 5;
  cout<<"local x =" << x;
     {
       int x = 10;
       cout << " Block x =" << x;
     }
     cout << "local x =" << x;
     Use local();
     Use static local();
     Use Global();
     Use local();
     Use static local();
     Use Global();
     cout<< " local x = " << x;
}
  void uselocal ()
  {
    int x = 25;
    cout << " local x =" <<x;
      x++;
   cout<< " Local x =" << x;
  }
   void usestatic local()
  {
    Static int x = 50;
     cout <<"local static x =" <<x;
      x++;
     cout<< " local static x =" << x;
  }
   Void UseGlobal()
  {
```

```
    cout<< "Global x = " << x;
        x = x * 10;
    cout << "Global x = " << x;
}
```

Function or subprogram :-

→ A complex problem can be decompose into small and easily managable parts, each part is called a subprogram.

→ A sub-program is a program unit which performs a particular task.

→ Sub-program are useful to read, write debug and modify, it can't execute itself and it is called either in the main prog. or by other sub-program.

→ It may be called any number of time in diff. places with diff. parameter value.

1. Function declaration — return type functionname (datatype arg1, ...)
2. Function defination — { .... }
3. Function calling. — functionname (arg1 , arg2, ...)

→ The act of transeffering control to a function is called function calling.

3 types of user defined function →

1.
```
void functionname( )
{
    // funct. does not return anything.
}
```

→ A function is called without taking any formal argument from the calling function of prog.

→ The called function does not return any value to calling function

② void function name (int a, int b)
{
// It will take argument but don't return.
}

③ int function name (int a, int b)
{
// return value ✓ + Argument ✓
}

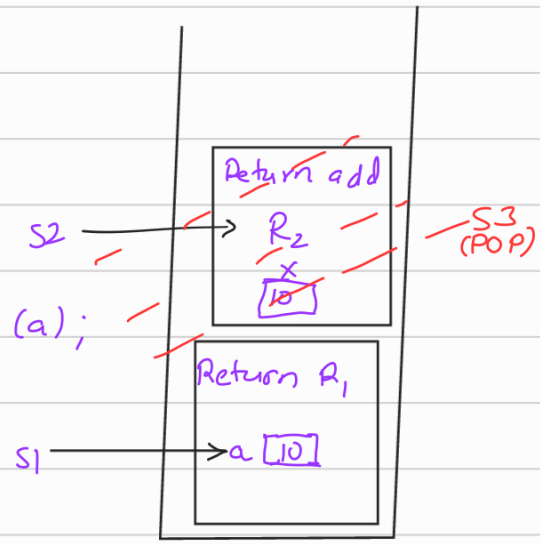
formal parameter →
→ A variable and its type as present in the prototype of the function

Actual parameter →
→ The variable or expression corresponding to formal parameter that is present in function call.

function call stack →
→ Stack data structure is used to perform the function call operation
→ Stacks are known as LIFO data structure.
→ Every time when a function calls to another function and entry is pushed into stack, this entry is call stack frame or activation record.
→ Activation records contains the return address that the called function needs in order to return to the calling function.
→ Activation record also support the creation maintainence & destruction of each function automic variable.
→ When the function returns the activation record for the function call is pop from the stack and control is transfered to the mention return address.
→ Activation records exist as long as the called function is active, when the function returns the local automatic variable are

no longer known to program.

```
int square(int)
  main ( ) {
  int a = 10;
  cout << a << " square of a = " << square (a);
  }

int square (int x)
  {
    return(x * x)
  }
```



Step1→ OS call the main function, this pushes a activation record into the stack.

Step2→ Main function call square 1 more Activation record pushed to stack.

Step3→ When function square returns, the (S2) activation record get popped from stack.

Step4→ main will pop & control will transfer back to OS.

Stack overflow →

→ The amount of memory in a computer is finite so only a limited amount of memory can be used to store activation records on function call stack.

→ If many function call occours and all have their activation record stored on function call stack we will not have enough space to store next activation record, a error will occour known as stack overflow.

Function calling →

1. Call by value
2. Call by reference.
3. Call by copy-restore

① Call by value →

→ When a argument is passed by value a copy of the arguments value is made and passed to the called function.

→ Changes to the copy do not affects the original variable value at the caller.

② Call by reference →

→ In call by reference the caller gives the called function the ability to access the callers data directly and to modify that data.

int &y = x y is reference to x.

Eg

```
int squarebyvalue (int)
void square by reference (& int)
 main () {
     int x = 2, y;
     int z = 4;
    cout << "x =  " << x;
    y = square by value(x);
    cout "x = " << x << " y = " << y << " z = " << z;
    Square by reference (z);
    cout << "z = " << z;
 }
 int squarebyvalue (int)
   {
      return number *= number;
   }
 void square by reference (int& ref)
   {
        Ref *= Ref;
   }
```

Pass by reference with pointer argument

Ex
```
void cubebyreference (int*)
main() {
  int number = 5;
  cout << "number = " << number;
  cubebyreference (&number);
  cout << "number = " << number;
  }

Void cubebyreference (int*)
  {
    *ptr = *ptr x *ptr x *ptr;
  }
```

③ Call by copy restore →

→ When a function call is made the values of actual parameters are copied to the formal parameter or formal arguments of the called function.

→ The formal parameters are changed inside the function, leaving the actual parameter unchanged upto this point.

→ But as soon as the called function completes its execution the values of formal parameters are copied to the actual parameter.

Ex
```
int y;
main() {
 y=10;
 copy-restore (y);
 print (y);
}

copy-restore (int)
 {
    X = 99;
    Y=0;
 }
```

← Not in C, C++

y
[0]
[99]

Y = 99
finally

Function overloading →

→ Function overloading enables multiple functions of the same name to be defined as long as they have different signature

→ A signature is combination of functions name & its parameter type in order.

Ex
```
int area (int x)
{
   cout << "Area of sq = " ;
   return (x*x);
}
int area (int x, int y)
{
   cout << " Area of Rect = " ;
   return (x*y);
}
main() {
   area(5);
   area(5,7);
}
```

Default arguments →

→ Its common for a program to call a function multiple times with same argument value for a particular parameter, in such case we can specify that such a parameter has default argument value.

→ Whe a program excludes an argument for a parameter with default argument in function call, the compiler rewrites the function call and inserts the default value of argument.

→ Default argument must be the rightmost argument in funct. parameter list.

Ex
```
int boxvolume (int l=1, int w=1, int h=1)
    {
        return( l * w * h ) ;
    }
main() {
 cout << " Defaut volume is = " << boxvolume() ;
 cout << " vol. of a box with L=10" << boxvolume (10) ;
 cout << " vol. of a box with L=10, b=5" << boxvolume (10, 5) ;
cout << " vol. of a box L=10, b=5, h= 2 " << boxvolume (10, 5, 2) ;
 }
```

User defined datatype →

① Structures ② Union

Structure →

→ Heterogenous type of data member

→ In prog. lang. different type of data can be grouped to form a structure.

→ This entire collection can be reffered by the name of structure

→ The individual components of a structure are called members and can be processed independently.

→ Every member of structure should have its own data type

Ex
```
Struct date {
    int day;
    int month;
    int year;
};
struct date d1, d2;
d1.day = 10 ;
d1.month = 7 ;
d1.year = 2022;
```
~~d1 = d2~~ Wrong.

Ex
```
Struct student {
    int roll no; int age ;
    char Name [20];
    struct date DOB;
};
student S1;
S1.roll no = 101;
S1.name = "ABC" ;
S1.DOB.Day = 10;
S1.DOB.month = 7;
```
⋮

Array of structure :-

Ex

```
struct student s[100];
s[i].roll no =
s[i].name =
s[i].age =
s[i].DOB.day =
```

Pointer to structure :-

```
struct sample {
    int x;
    float y;
    char z;
};
struct sample *ptr;
struct sample one;
  ptr = &one;
Ptr -> x = 10;        or    (*ptr).x = 10
Ptr -> y = 20.5;
ptr -> z = 'A';
```

Self referential structure :-

→ Inside struct we can't have variable of same type but we can include it by using pointer.

Ex

```
struct node {
    int data1;
    char data2;
    struct node *link;
};
node = n1, n2 ;
n1.data1 = 10 ;
n1.data2 = 'A' ;
n1.link = &n2 ;
```

n_1

10	A	\0

n_2

20	B	\0

```
n2.data1 = 20;
n2.data2 = 'B';
n2.link = NULL;
```

Unions →

→ Union is similar to structure datatype with a difference in a way the data is stored.

→ Union stores values of different type in single location.

→ If a new value is assigned, the previous value automatically erase.

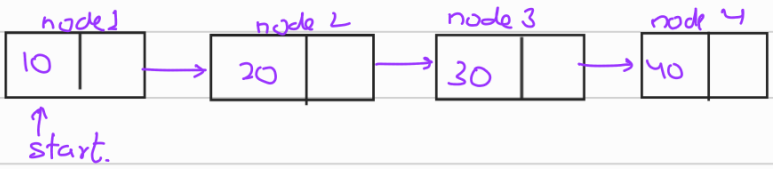
Ex

```
Union value {
    int a;
  double d;
};
value x;
x.a = 12
x.d = 123.4
cout << x.d;
```

] first x.a = 12 created then destroyed when x.d is declared.

Simple link List of 4 nodes →

```
struct node {
    int data;
  struct node* link;
};
main() {
struct node node1, node2, node3, node4;
  struct node * start;
  node.data = 10;
  node.link = NULL;
  node2.data = 20;
  node2.link = NULL;
  node3.data = 30
  node3.link = NULL;
```

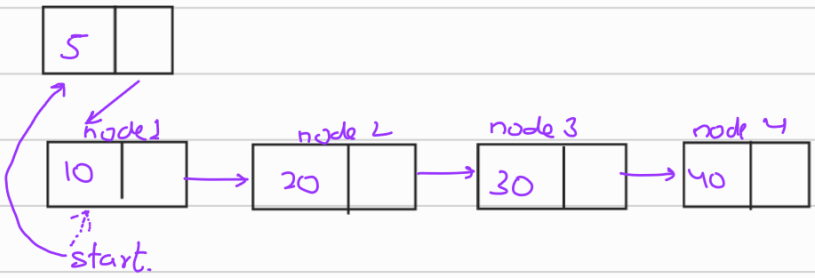


```
node4.data = 40;
node4.link = NULL;

node1.link = &node2;
node2.link = &node3;
node3.link = &node4;
start = &node1;
}
```

Insertion of node in linked list →

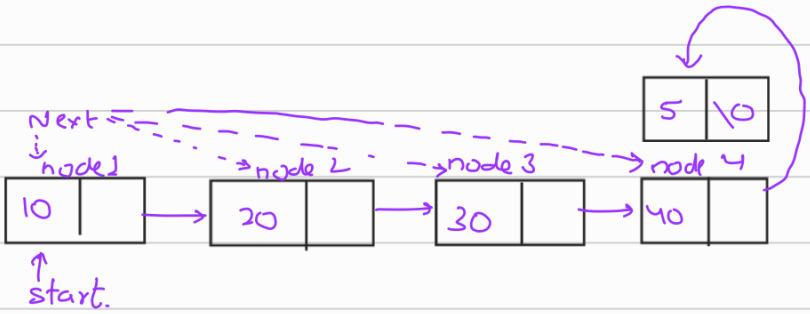
→ At start , at end , at given location.

At start →




```
struct node new_node;
new_node.data=5;
new_node.link=NULL;
new_node.link= start;
start = &new_node;
```

At end →




```
struct node *next;
next = start;
while (next -> link != NULL)
        next = next -> link;
next -> link = &new node;
```

insert where node.data = 30;

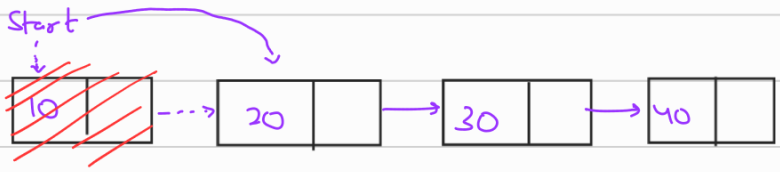
```
struct node new_node;
new_node.data=5;
new_node.link=NULL;
struct node * Next;
next = start;
```

```
while (next -> data ≠ 30)
    next = next -> link;
new_node.link = next -> link;
    next -> link = &new_node;
```

deletion of node in linked list →

→ first node, last node, at some position.

<u>first node deletion</u> →



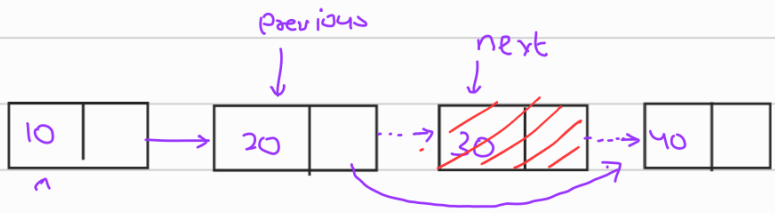

```
struct node {
    int data;
  struct node* link;
  };

main() {
  struct node* next;
      next= start;
      start= start -> link ;
      free (next);
```

<u>Deletion of last node</u> →

```
main() {
  struct node* next, *previous;
    next=start;
while (next -> link  != NULL)
   {
        previous= next;
         next = next -> link;
    }
      previous -> link = NULL;
      free(next);
```

Delete the node data = 30 →

```
struct node *next , * previous;
 next = start;
 while ( next → data != 30)
  {
     previous = next;
     next = next → link;
  }
   previous → link = next → link;
    free ( next);
```



OOP's →

→ Object Oriented Programming is a software developement phylosiphy based on the following central ideas.

(1) Class
(2) Object
(3) Data abstraction
(4) Data hiding
(5) Data Encapulation
(6) Inheritance
(7) Polymorphism

(1) Class →

→ It is a user defined data type which holds both data and function together.

→ The internal data of a class is called member data.

→ The internal funct. of class are called member function.

→ Member function of class mostly maninuplate internal member data of class.

(2) Objects →

→ A variabel/ instance of class is called object

→ Every object is a collection of data members and associated member function.

(3) Data abstraction →

→ It refers to providing only essential information to the outside world & hiding their background detail

→ It is a prog. technique that works on sepration of interface and implementation.

(4) Data hiding →

→ It is a technique of hiding the internal members of an object so that other part of prog can't access it.

→ In C++ prog. environment there are 3 access specifier to implement data hiding i.e public, private, protected.

(5) Data encapsulation →

→ It is a mechanism bundling the data and function in a single unit.

```
class   user-def_name {
 Public:
        data_type members;
        implementation function;
        list of friend function;
 Private:
        data type members
        implementation function
        list of friend function
Protected:
        data_type members;
        implementation function;
        list of friend function;
};
```

→ General defination, declaration of class.

public →

→ The member which are declared in public section can be accessed by any function outside the class.

→ The public implementation operation are also called interface of the class.

Private →

→ The members which are declared in private section of a class can only be accessed by the member functions and friends of this class.

Protected →

→ The members which are declared in protected section of class can be accessed by member function & friends of this class.

→ These members can also be accessed by the member function and friends of the class derived from this class.

Ex

```
Class sample {
    private:
      int x;
      int y;
    public:
      int sum() {
          return (x+y);
      }
      int diff() {
          return (x-y);
      };
main() {
    sample one;
    one.sum();
    one.diff();
}
```

~~one.x = 10~~ → x is private.

Constructor →

- A constructor is special member function for automatic initialization of an object.
- Whenever a object is created the constructor will execute automatically
- A constructor function can be overloaded to accomadate diff forms of initialization.
- Constructor name must be same as that of class name.
- It is declared with no return type.
- Constructor is not declared static or virtual.

Ex

```
Class abc {
    public:
     abc ( ) {
     cout<< "constructor call";
     }
  };
main() {
   abc obj1;
   abc obj2;
 }
```

Output :-

```
constructor call
constructor call
```

- Most of time constructor is defined in public section of a class, but some time we can define it in private section of class.
- If we have defined in private section of class then only friend function & friend classes can create object of that class

Q Write a program to generate a fibonacchi series using constructor?

```
Class Fibonacci {
   Private:
        int f0, f1, fib;
   Public:
```

```
    fibonacci() {
      f0 = 0;
      f1 = 1;
    fib = f0+f1;
    cout<< f0 << " , " << f1 << " , ";
  }
  void increament() {
     f0 = f1;
     f1 = fib;
    fib = f0+f1;
  }
  void display() {
    cout << fib << " , ";
  }
  main() {
     fibonacci number;
     for( int i=3; i<=10; i++) {
       number.display();
       number.increament();
     }
  }
```

Types of constructor

1. Deafult constructor
2. Parameterised constructor
3. Copy constructor.

Default constructor →

→ Defaut constructor is a constructor with no argument or all argument should have defaut value.

→ If user does not define default constructor in prog., compiler will define empty default constructor.

Type 1:-

```
Class Student {
    private :
        char name[30];
        int rollno;
        float height;
    Public:
        student() { }
            ⋮
    }
```

Type 2:-

```
Class Student {
    private :
        char name[30];
        int rollno;
        float height;
    Public:
        student(int r=123)
            { roll no = r ; }
    }
```

Type 3:-

```
Class Student {
    private :
        char name[30];
        int rollno;
        float height;
    Public:
            ⋮
    }
```

<u>Parametrized constructor</u> →

```
Class abc {
    int a, b;
```

```
Public:
    abc (int x, int y) {
        a=x;
        b=y;
    }
void display() {
    cout<< a<<b;
  };
main() {
  abc obj1;        → Error as no defaut constructor
  abc obj (2, 10);
}
```

Copy constructor →

→ It is a constructor which is used to initialize an object by another object of same class, which has been created previously.

```
Class-name ( const Class_name & obj )
```

Ex

```
Class Point {
Private:
  int x, y;
Public:
  Point ( int x1, int y1)
   {
      x = x1;
      y = y1;
   }
  Point (const Point & p)
   {
      X = P.X;
      Y = P.Y;
   }
}
```

```
main() {
  Point  P1 = (10,5);
  Point  P2= P1;
  Point P3 (P1);
}
```

3 Places where copy constructor get called

1. Passing obj as argument
2. returning obj in function
3. Initialization an obj with obj.

```
Class Test {
      Private :
         int *ptr;
       Test (int n)
         {
           ptr = new int;
           *ptr = n;
         }
      void print() {
         cout<< *ptr;
      }
     void set value (int y)
      {
         *ptr = y;
      }
  };
```

```
Test (const Test &T)
{
   Ptr= new int;
  *ptr= *(T.ptr);
}
```

So the problem of same memory location will be solved

```
main() {
      Test t1 (10);
      Test t2 (t1);
      t1. setvalue (5);
      t2. print();
```

→ Ptr of t2 points to same location i.e Ptr t1 if we will not create copy constructor.

}

Constructor overloading →

Ex:
```
Class ABC {
    public:
      ABC ();
      ABC (int);
      ABC (int, float);
  };


ABC :: ABC ()
        {
         cout << " Calling Default Constructor " << endl;
        }

ABC :: ABC (int n)
        {
         cout << " Calling parameterized constructor with int " << endl;
        }

ABC :: ABC (int A , float B)
        {
         cout << " constructor with int A & float B " << endl;
        }

main ()
    {
      ABC obj1;
      ABC obj2(2);
      ABC obj3(2, 2.1);
    }
```

Destructor →

→ A destructor funct. get called automatically whenever a object get destroyed or an obj. go out of existance.

→ Primary use of destructor function is to release space occupied by an object.

→ Destructor funct. may be called explicitly.

→ A destructor function is same as that of the class it belongs except that the first character of the name must be a tilde (~) sign.

→ It is declared with no return type.

Ex
```
Class ABC
  {
   ⋮
  ~ABC()
  }
```

→ If we don't define destructor in a class compiler will auto. add it.

→ Destructor can't be overloaded

→ Destructors can't be declared static, but it can be virtual.

→ Destructor is called in public section of class

Ex
```
Class ABC {
    Public:
      ABC () {
          cout << " C.Call" << end;
      }
      ~ABC() {
          cout << "Dest. call" << endl;
      }
    };

main() {
        {
          ABC obj1;
        }
    }
```

← Here as soon as blockend, Destructor call take place

Ex Explicit call obj1.~ABC()

Nested class →

→ C++ permits declaration of class within another class

→ A class declared as member of another class is called nested class.

→ Nested class is in scope of enclosing class.

Ex

```
Class student_info {
    private:
        char name[30];
        int rollno;
    Public:
        class date {
            Private:
                int day, month, year;
            Public:
                void set_date (int d, int m, int y)
                {
                    day = d;
                    month = m;
                    year = y;
                }
            };
    void display();
    void info();
main() {
    student_info obj1;
    student_info :: date obj2;
    obj. set_date (2, 11, 2022);
}
```

→ Inner class can access of Outer class but not vice versa.

Ex

```
Class outer {
  private:
    int n;
  Public:
   class inner{
     private:
        int y;
     Public:
       void fun (outer O1)
        {
          cin>>y;
          int z= O1.X + Y;
          cout<<z;
        }
   };
 void outer_fun () {
       cin>>n;
     }
};

main() {
  outer obj1;
 outer:: inner obj2;
  Obj1. outer();
  obj2. outer (Obj1);
}
```

Similar funct is not possible to access inner private data. in outer class.

→ Inner most Destructor is called first. (i.e in nested one first)

Destructor in nested classes →

→ The execution order of destructor under nested class in that the inner most class object will be released first and so on.

Ex →

```
Class ABC {
    Pubic :
     ~ABC();
    Class X {
        public:
          ~X();
         Class Y {
             Public:
               ~Y();
                 };
             };
    };

ABC::~ABC()
  {
 cout<<"ABC Class destructor"<<endl;
  }
ABC::X::~X()
  {
   cout<<" x Class destructor"<<endl;
  }
ABC::X::Y::~Y()
  {
   cout <<" Y class destructor"<<endl;
  }
main()
 {
    ABC obj1;
   ABC::X obj2;
   ABC::X::Y obj3;
}
```

Friend function & friend classes →

Friend function → (ff)

→ It is a special mechanism by which a non member function can access the private data of a class

→ A friend funct. can be declared and defined within the scope of class defination.

→ The keyword friend informs the compiler that it is not a member function of class.

→ If a ff is declared within the class it must be defined outside the class without repeating the friend keyword.

Ex

declaration + defination inside

```
Class ABC {
    private:
      int x;
    Public:
      void get data()
        {
          cin >> x
        }
      friend void display (ABC A1)
        {
          cout << " Value of x = " << A1.x;
        }
};
```

Ex2

declaration inside

```
Class ABC {
    Private:
      int x ;
    Public:
      void get data() ;
      friend void display (ABC  A1 ) ;
};
```

```
void ABC::getdata ()
  {
    cin >>x;
  }
void display ()
  {
    cout << " Value of x = " << A1.x;
  }
main ()
 {
  ABC  obj1;
  obj.getdata();
  display (obj1);
 }
```

<u>Two classes having same friend</u> →

→ A non-member funct. may have friendship with one or more classes

→ When a function has friendship with more than 1 class the friend classes should have forward declaration.

→ This funct. can access private members of all classes where it is mentioned as friend.

Ex

```
Class Second;
Class first;
class first {
  Private:
    int x;
  Public:
void getdata();
void display();
friend int sum (first, second)
};
```

```
Class Second {
  Private :
      int y;
  Public:
  void getdata( );
  void display( );
friend int sum ( first , second)
 };


int sum ( first one , Second two)
 {
    int temp;
    temp = one.x + two.y;
    return (temp);
 }
 void first::getdata ( )
  {
     cin >>u;
  }
void first :: display ( )
  {
   cout << " Value of x = " << AL.x;
   }
```

```
void second::getdata ( )
  {
     cin >>u;
  }
void second :: display ( )
  {
   cout << " Value of x = " << AL.x;
   }
```

```
main( )
{
  first a;
  second b;
  a.get data( );
  b.get data( );
  int r= sum (a, b);
    cout<< r;
}
```

Class as friend of Class →

→ If the class first mention the class 2nd as friend the private data members of class first are permitted to be accessed by the member function of class 2nd, but on the other hand the member function of class 1st can't access the private data of class 2nd.

Eg

```
Class second;
Class first {
    friend Class second;
    private:
     int x;
   Public:
    void get data()
    {

     cin >> x;
    }
};
Class Second {
  private:
    int y;
  Public:
    void display (Class first temp)
    {
      cout << temp.x;
    }
};
main()
{
  first obj1;
  second obj2;
  obj1.get data();
  obj2.display(obj1);
}
```

Can't access Y with x until we make class first as friend of Class second.

Static member of class →

- Whenever a data or function member is declared as static it belongs to a class, not to an object of class
- Static data members are common to all objects of a class
- Static members are already created before the finite object of the respective class.
- Static members are used in information that is commonly excessable to all.
- Static members can be defined in any group. i.e public, private or protected.
- Static member function can access only the static data members or static member function of the class.
- Static member funct. can be called directly by using the class name and scope resolution operator.

Ex

```
Class Sample {
    private :
        static int count;
    public :
        sample( );
    static void display();
};

int sample :: count = 0;      ← defination, initialization outside always.
    sample :: Sample()
        {
            count ++;
        }
void sample :: display()
        {
            cout << count;
        }
main()
 {
```

```
Sample::display()
Sample obj1;
Sample obj2;
Sample obj3;
Sample obj4;
Sample::display();
}
```

Output = 4.

→ Static funct. can't access/use non-static members directly as it will depend on object.

This pointer →

→ Every object in C++ has access to its own address through a pointer called this pointer.

→ This pointer is an implicit parameter to all member function. therefore inside a member function this may be used to refer the invoking object.

→ friend funct, static funct. don't have this pointer.

Ex

```
Class Box {
    Private:
        int length;
        int breadth;
        int height;
    Public:
        Box (int l=2; int b = 2; int h = 2)
        {
            length = l;
            breadth = b;
            height = h;
        }

int volume()
{
```

```
    return (length * breadth * height);
  }
int compare (BOX b1)
  {
    return((this -> volume) > b1.volume( ));
  }
main()
 {
  Box Box1 (3, 1, 4);
  Box Box2 (8, 6, 4);
 if (Box1.compare (BOX2))
   {
     cout<< "Box2 is smaller than Box2";
   }
 }
```

Cascaded function call →

→ Invoking multiple member function in the same statement is called cascaded funct. call, it is implemented with the help of this pointer.

```
Class Time {
  private:
    int hour;
    int minute;
    int second;
 public:
 Time& set_hour (int h)
   {
     hour = h;
    return *this;
   }
```

```
Time & set_minute (int m)
  {
    minute = m;
    return *this;
  }
Time & Set_second (int s)
  {
    second = s;
    return *this;
  }
};
main() {
    Time T;
  T.sethour(20).Set minute(18).set second(30);
}
```

Step1 → T.sethour(20)

Step2 → T.Set minute(18).set second(30);

Step3 → T.set second(30);

Composition →

→ A class having object of another class as a member is called composition, also reffered as has-a relationship.

E.

```
Class basicinfo {
  Private:
    char name[30];
    int roll no;
    char gender;
  Public:
    void getdata();
    void display();
};
```

```
Class acedemicinfo {
  Private:
    char course[20];
    char sem[10];
    int rank;
  Public:
    void getdata();
    void display();
};
Class Financialassist {
 private:
   basicinfo bdata;
   Acedemicinfo Adata;
   float amount;
Public:
  void getdata();
  void display();
};
void financialassist::getdata()
 {
     bdata.getdata();
     Adata.get data();
     cin>> amount;
}
void financialassist:: display() {
  bdata.display();
  Adata.display();
  cout<< amount;
}
 main()
 {
  financial assist f;
  f.getdata();
```

```
f.setdata();
}
```

(Database through Navathe)

Inheritence →

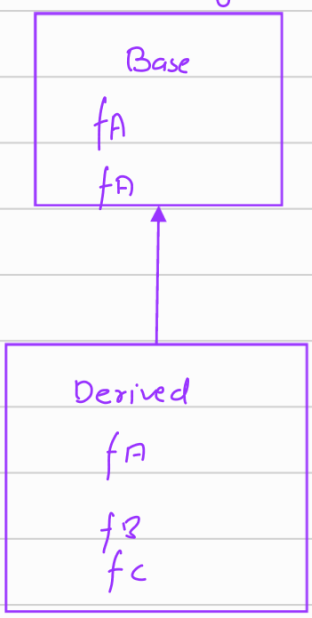
- It is the process of creating a new class from an existing class.
- It is a form of software reuse in which you create a class that absorb an existing class capabilities and enhances them.
- The existing class is known as base class and newly created class is called derived class.

BASE CLASS	DERIVED CLASS
Person	student
vehicle	car, truck

- Main advantages of inheritence:

(1) Reusability of code
(2) Increase the relybility of code
(3) Enhancement's to the base class.

Class Dig.



Syfax to make a derived class →

```
Class Derived_classname : public / private / protected  base_class
{
   private :
      datamember;
   public:
      data member;
   protected :
      data member;
};
```

Ex

```
Class person {
   private:
      char Name[20];
      int age;
      char gender;
   public:
      void getdata();
      void display();
};
Class student ::Public person
{
   private:
      char branch[10];
      int rollno;
   public:
      void get info();
      void display_info();
};
void person :: getdata()
{
   cin >> name;
   cin >> age;
   cin >> gender;
```

```
}
Void person:: display()
{
  cout<<name;
  cout << age;
  cout<< gender;
}
void student :: get_info()
  {
    cin >> Branch;
    Cin>> Rol_no;
  }
void student::display_info ( )
  {
    cout << Branch;
    cout<< Rol_no;
  }
main()
{
Student A;          ← Derived class object can call funct. of base class.
A.getdata();
A.get_info();
A.display();
A.display_info();
}
```

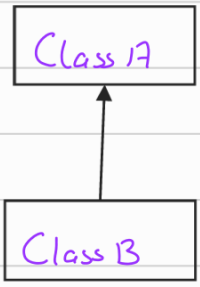
→ With all the forms of inheritence private members of a base class are not accessable directly from derived class but these private base class members are still inherited.

→ Friend functions are not inherited.

→ To use base class private member we have to call public funct. of base class.

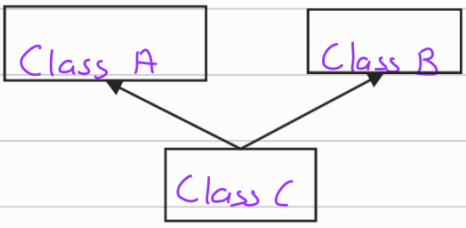
Type of inheritence →

(1) Single inheritance →



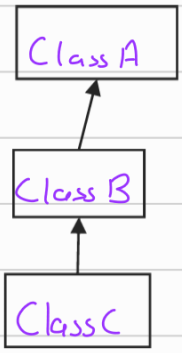

```
Class A {
    ⋮
};

Class B : Public A
{
    ⋮
};
```

(2) Multiple inheritance →




```
Class A {---};
Class B {---};
Class C : Public A, Private B
    {---};
```

(3) Multilevel inheritance →




```
Class A {
    ⋮
};

Class B : Public A
{
    ⋮
};

Class C : Public B
{
    ⋮
};
```

Types of derivation →

		Base	Derived
(1)	Public inheritence	Public	Public
(2)	Private	Private	private
(3)	Protected	protected	protected.

→ Private member of base class remain private in base class.

Base	Derived
Public	Private
protected	private
private	private

Base	Derived
Public	protected
protected	protected
private	private.

Ex-

```
Class base {
  Private:
    int x;
  Protected:
    int z;
  Private:
    int y;
  void basefun( )
  {
    cin>> x;
    cin>>y;
    cin>> z;
  }
};
class derived: public base {
  Public:
    void derived
    {
      base::x++;  → error (private)
```

```
        base :: Y ++;
        base :: Z++;
        cout<< base :: Y;
        cout<< base::Z;
    }

  friend void friend_Derived (Derived D1)
    {
      ~~D1.base :: X++;~~ → error (private)
      D1. base :: Y++;
      D1. base :: Z ++;
      cout<< D1.base :: Y;
      cout<< D1.base ::Z;
    }

main ( )
 {
 Derived D;
 D.basefun( );
 D.derived_fun ( );
 friend_derived ( D);
 }
```

Ambiguity in inheritence →

→ Whenever a data member and member function are defined with similar name in both the base class and derived class these name can be accessed without ambiguity with the help of ::

E-
```
Class A {
  Public:
    int i;
    void getdata ( )
      {
        cout<<"A";
```

```
    }
};
Class B {
  Public:
    int i;
    void getdata ( )
    {
      cout<<"B";
    }
};
Class C {
  Public:
    int i;
    void getdata();
    {
      cout<<"C";
    }
};
main()
{
  C obj;
 obj.getdata();
 obj.A:: getdata();
 obj.B:: getdata();
}
```

<u>Constructor and Destructor in Derived class</u> →

→ when a prog. creates a derived class object the derived class constructor immediately calls the base class constructor, the base class constructor body executes first and then the derived class constructor execute

→ Wheather derived class default constructor or parameterized constructor is called base class default constructor is always called

inside them.

→ To call the base class parameterized constructor, inside the derived class parameterized constructor, we must mention it explicitly inside the derived class parameterized constructor.

→ If the base class is having only parameterized constructor then derived class constructor must call it inside derived class default or parameterized constructor.

→ When a derived class destructor is called the destructor perfor-ms its task then invokes the destructor of the next base class up in the hirarchy, this process repeate untill the destr-uctor of final base class at the top of the hirarchy is called.

Ex

```
Class Base {
    int x;
Public:
   Base ()
   {
    cout<< "Base Constructor (default)" <<endl;
   }
   ~Base ()
   {
    cout<< "Base Destructor "<<endl;
   }
};
Class derived : public Base {
    int y;
Public:
    Derived ()
     {
        cout << " Derived constructor" <<endl;
     }
    Derived (int i)
     {
        y = i;
```

```
    cout << " Derived parameterized constructor" <<endl.
    }
  ~ Derived( )
    {
      cout<< "derived destructor" <<endl;
    }
};
main ()
{
  Base B;
  Derived D1;
  Derived D2(5);
}
```

Ex2

```
Class Base {
  private:
    intx;
  public;
   Base (int i)
   {
     x = i;
     Cout << " Base Constructor " <<end;
   }
};
Class Derived : public Base {
   private:
     inty;
   Public:
     Derived (int j)
     {
       y = j;
       cout << " Derived constructor" <<endl;
```

```
    }
};
main(){
    Base B;
    Derived D1;
}
```

How to access private data of base class in derived class. public member function →

```
Class base{
    friend class derived;
    private:
        int value;
    Public:
        Base()
        {
            cin>> value;
        }
};
class derived : private Base
{
    Public:
        void display()
        {
            value++;
            cout<<value;
        }
};
main()
{
    derived D;
    d.display;
}
```

In C++, base class pointer can hold derived class object.

Polymorphism →

many forms

→ In oops polymorphism refers to identically named methods (member function) that have diff. behaviour depending on the type of object they refer.

→ Polymorphism enables you to programm in general rather then program specific.

Ex Suppose that we are designing a video game that manipulates object of different types including objects of class car, jeep, truck, each of these classes inherits from a common base class vehicle which contains the member function draw.

Draw vehicle

Draw Car Draw truck Draw Bus

which to use is defined at runtime. as derive class will override the function this is called as runtime polymorphism

→ Every class redefine draw function according to its requirement.

Polymorphism in C++ →

→ The base class pointer can hold address of derived class object

Ex
```
Class A {
        }

Class B : Public A
        {
        }

 main ()
  {
      A* ptr;
      B obj1;
```

```
        ptr = &obj1;
    }
```

Ex →

```
Class baseA {
 public:
     void display ( )
      {
          cout << "one" ;
      }
   };
Class derivedB : public baseA {
  public:
     void display ( )
      {
          cout << "Two" ;
      }
  };
Class derivedC; public baseA {
   public:
      void display ( )
       {
          cout << "Three" ;
       }
  };
main ( )
{
   baseA  obja ;
   DerivedB  objb ;
   DerivedC  objc ;
   baseA * ptr[3] ;
   ptr[0] = & obja;
   ptr[1] = & objb;
   ptr[2] = & objc;
```

```
for (int i=0;  i<=2; i++)
  {
      ptr[i] -> display();
  }
}
```

output ⇒ one one one.
As early binding / compile time binding occour, base display will be called.
compile time binding → during compilation compiler decide which function to be called.

∴ to acheive polymorphism we use virtual function, inorder to achive dynamic binding.

```
Class baseA {
 public:
    virtual void display()
     {
       cout << "one";
     }
  };
Class derivedB : public baseA {
 public:
    void display()
     {
       cout << "Two";
     }
 };
Class derivedC; public baseA{
 public:
    void display()
     {
       cout << "Three";
     }
};
```

→ dynamic binding hogi virtual ke kaaran.
dynamically function of pointed objected will be called.

```
main()
{
  baseA  obja;
  DerivedB objb;
  DerivedC objc;
  baseA * ptr[3];
  ptr[0] = &obja;
  ptr[1] = &objb;
  ptr[2] = &objc;
for (int i=0;  i<=2; i++)
  {
      ptr[i] -> display();
  }
}
```

output:

one two three

Function overriding →

→ If derived class defines same function as defined in its base class, it is known an function overriding

→ It enables you to provide specific implementation of the funct. which is already provided by its base class.

Virtual destructor →

→ A problem can occour when using polymorphism to process dynamically allocated objects of a class hirarchy.

```
Base* b= new derived();
      ⋮
delete b;
```

↓

base class destructor will be called not the derived class. so, derived class part of obj. will not be free & this problem is undefined in C++ environment & it is called as memory/Resource leak

Solution → ??

```
Class Base {
 Public:
   virtual void display ( )
   {
       cout <<"Base";
   }
   virtual ~Base() {}
};
Class derived : public Base {
Public:
    void display();
    {
       cout<<"Derived";
    }
};
main ()
{
  Base *b = new derived();
     b → display;
     delete b;
}
```

→ first deri. class destructor will be called the base hence memory will be free.

Abstract classes →

→ An abstract class is a base class from which other classes can inherit.

→ We can't create any object of abstract class.

→ An abstract class is incomplete class, derived classes must define all the missing pieces before objects of these classes can be created.

→ If derived class miss to give defination to missing piece, then der. class itself become a abstract class.

→ Pure virtual function →

→ Virtual void fun() = 0;

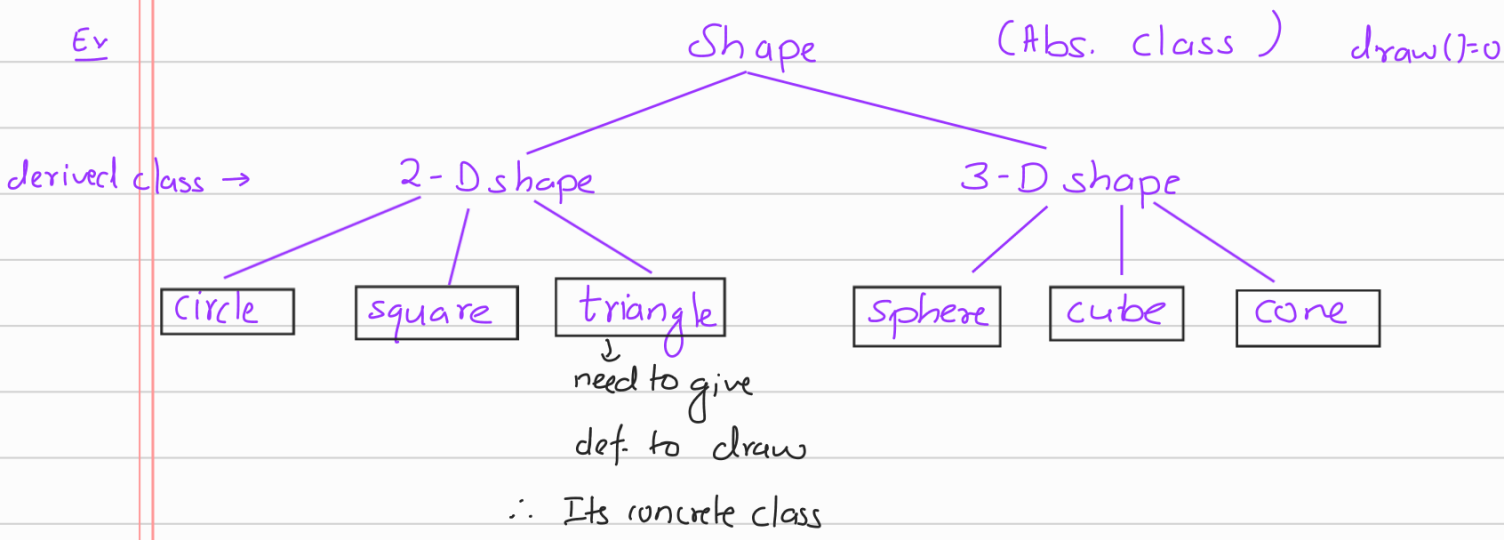
→ An abstract class should have at least 1 pure virtual funct.

→ An abstract class can also have data members and other member function which are subject to normal rule of inheritence by derived class.

→ Each concrete derived class must override all base class pure virtual function, with implementation of those functions, otherwise the derived class also become abstract.

→ We can't use abstract class as a parameter type or a funct. return type, but a pointer can be declared of abstract type.

Ex

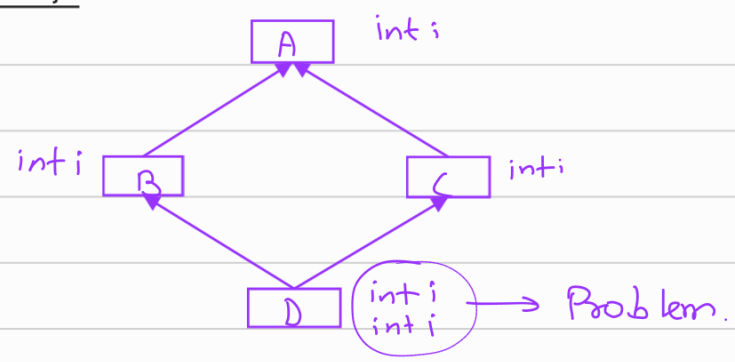


Ex

```
Class Shape {
public:
  virtual int getarea( ) = 0;
  int width;
  int height;
  void setwidth (int w) {
        width = w;
    }
  void set height (int h) {
        height = h;
    }
};
Class Rectangle : public shape {
```

```
public:
int getarea() {
    return width * height;
  }
};
Class triangle : public Shape {
  public:
    int getarea() {
        return (height * width / 2);
        }
  };
main() {
  Rectangle Rect;
  Triangle T;
  Rect.set width(7);
  Rect.setheight(5);
  Rect.get area();
  Tri.set width(10);
  Tri.setheight(20);
  Tri.get area();
}
```

Diamond problem →



Ex

```
Class A {
Public:
    int i;
```

constructor of D can call constructor of A if A is inherited virtually to B & C which is inherited to D.

```
A () {
    cout << "A" ;
    }
void fun( ) {
      cin>>i;
      cout<<" i=  " <<i;
      }
};
                Virtual
Class B : public A  {
Public:
    B() {
      cout <<"B";
      }
};
                Virtual
Class C : Public A  {
 Public:
    C() {
      cout <<" C ";
    }
};
Class D: Public B , Public C {
 Public:
    D() {
      cout <<" D ";
    }
};
main ()
 {
                                          without virtual
    D obj1;                    // O/P:  A  B  A  C  D
  //obj1.fun();  → Ambiguous., as fun is present due to D, C, B
 }                to solve we need to make A a virtual class (↑
                  show where to write virtual to solve problem)
```

Ex

```
Class A
public:
 int i;
 A (int x) {
        i=x;
        cout <<"A";
    }
void fun()
   {
     cout << " i= " << i;
    }
};
Class B : virtual public A {
public:
    B (int y) : A(Y) {
          cout <<" B";
        }
};
Class C : virtual public A {
 public:
    C (int z) : A(z) {
          cout << " C";
      }
};
Class D: public B, public C {
 public:
    D(int P) : A(P) B(P) C(P)
      {
          cout <<" D";
      }
};
```

Exception handling →

→ An exception is an error or an unexpected event occours at run time

ex: divide by '0'

ex: Memory not allocated

ex: Memory out of bound

ex: file not open.

→ exception handler is a set of code in a prog that execute when an exception occurs.

→ The main purpose of exceptional handling prog is to detect & manage run time error.

→ Program must be developed with exception handling in such a way that it identifies the possible error in the prog. and then include code to handle them.

→ In C++ to handle exception we use 3 key words try, catch, throw.

(i) Try Block →

→ A try block protects any code within the block

→ only the code inside a try block can detect or handle exception

→ Try block can also be nested like any other C++ prog.

Syntax:
```
try {
 // the code wants to
// protected from errors
}
```

(ii) Throw block →

→ The throw statement throws an exception of a specified type.

Syntax: throw argument;

throw ⓧ → throw prev thrown value (if x is not there)

(iii) Catch block →

→ The catch block recives the thrown exception.

Syntax:-

```
Catch (parameter)
 {
  // handle error here
 }
```

parameter = type name

This is must.

→ parameter can be named or unnamed denotes the type of exception the "catch" handles.

→ This parameter can be any valid C++ data type.

Q. Write a prog to detect divide by '0' error using exception handling ?

```
Class sample {
private:
   int a, b;
public:
   void getdata ();
   void divide ();
};

void sample :: get data ()
   {
      cin >> a >> b;
   }

void sample :: divide ()
  {
     int x;
     try {
        if (b==0)
          throw x;
        else
           float temp= a/b;
           cout<<temp;
```

```
        }
    Catch (int y)
     {
      cout << " Exception divide by zero";
     }
}

main ()
{
 sample obj;
 obj.getdata ();
 obj.divide ();
}
```

Ex WAP to demonstrate how to use multiple catch statement for diff type of exception?

```
Class Sample {
   private:
      int a;
   Public:
      void getdata ();
      void display ();
};

void sample::getdata()
{
    cin >> a;
}

void sample::display()
{
   int n;
   char ch;
   float f;
  try {
     if (a==0) throw n;
     if (a==1) throw ch;
```

```
    if (a==2) throw f;
    else
      cout<<" x = "<<x;
  }
catch (int n)
  {
    cout <<" Exception_int ;
  }
catch (char n)
  {
    cout <<" Exception_char ;
  }
catch (float n)
  {
    cout <<" Exception_float ;
  }
}
```

instead we can do::

```
catch (...)
{
    cout << "Exception";
}
```

This can catch all exception.

```
main() {
    sample obj1;
    obj1.getdata();
    obj1.display();
  }
```

Throw without argument

```
Class abc {
Public:
  void display (int x, int y)
};
void abc :: display (int x, int y)
{
    try {
      if (y==0)
        throw y;
```

```
        else
            cout<<x/y;
    }
  catch (int)
    {
        cout <<" Exception  Occoured ";
        throw;
    }
}
main()
{
    abc obj;
    try {
            Obj.display(1,0);
        }
     catch (int)
        {
         cout<<"Exception in main";
        }
}
```

<u>Nested try block</u> →

```
try {
      fun1();
       try {
            fun2();
          }
      catch (spec_error) {- - -}
       fun3();
    }
catch (type_error)
    {-----}
```

→ When try blocks are nested and a throw occours in a function called by inner try block control is transferred outward through the nested try blocks untill the first catch block is found, whose argument matches the argument of throw expression

fun1() – Spec. err ✗
type err ✓ fun~~3~~2()
fun2() – Spec-err ✓
type-err ✓ fun~~3~~3()
fun3() – Spec err ✗ (prog aborted)
– type err ✓

<u>Exception with user defined type :-</u>

```
Class Demo {

    };

 main() {
   Demo D1;
   try {
          throw D1;
       }
   catch (Demo D1)
       {
         cout << " Exception of demo class " <<endl;
       }
 }
```

<u>Exception handling with inheritence →</u>

```
Class Demo1 {
  };
Class Demo2 : public Demo1 {
  };
 main() {
```

```
for (int i=1; i<=2; i++)
 {
    try {
          if (i==1)
             throw Demo1();
          else if (i==2)
             throw Demo2();
        }
    Catch (Demo1D1)
      {
        cout<<"Exception of demo 1 class";
      }
    Catch (Demo2D2)
      {
        cout<<"Exception of demo2 class";
      }
  }
}
```

Exception of Demo 2 (derived) can be catched by catch of base class.

∴ o/p ⇒ Exception of demo 1 Class

→ Build in exception exceptions ex: bad alloc, pointer to null typeid in polymorphic value.

Standard exception →

→ All objects thrown by components of the standard library are derived from exception class.

→ ∴ All standard exception can be caught by catching this type by reference i.e catch (exception &)

→ This class exception has what() func. which every derived class inherit & this tells what type of exception is there.

Ex bad alloc → Exception thrown on faliure of memory allocation
bad type_id → Exception throw by type-id if pointer points to NULL.

Exception ← bad_alloc

- This type of exception thrown by the standard defination of operator new, when it fails to allocate the requested memory space.

```
main ()
{
 try {
  int* anyarray = new int [10000000000];
   }
  catch (exception &e)
    {
        cerr<< e.what()
    }
}
```

clog
cin
cerr

Exception ← bad_typeid

- This type of exception thrown by type-id when applied on a pointer to a polymorphic type which has a null pointer value

```
Class polymorphic {
      virtual void member();
      };
main () {
   try{
      polymorphic * ptr = NULL;
          type-id (ptr);
      }
   catch (exception &e)
     {
```

```
        cerr << e.what();
    }
}
```

Compile time polymorphism →

Operator overloading →

→ We can redefine or overload most of the built in operator avaliblein C++ prog environment, a prog. can use operators with user defined datatypes.

→ Overloaded operator is a funct with special name

```
return_type  operator  +  (parameters)
             ↑ Keyword
```

→ The main idea of using operator overloading is to make it more natural to read & write.

→ User cannot change operators template such as wheather it is a binary operator or unary operator & order of precedence

→ The overloaded operators contain at least 1 operand of the user defined type.

Ex. obj1 + obj2 − obj3 * obj4 − 5

4 operator to be overloaded

−ve overload 2times

1 − b/w^n Obj − obj

2 − b/w^n Obj − Natural

→ Overloaded can't have default argument.

→ Operator can be overloaded by

① Non-static member funct.

② Friend-funct

Overloading a binary operator →

→ binary operator overloaded by means of non-static member function. takes 1 formal argument which is the right operand of operator

→ The binary operator overloaded by means of friend funct. takes two arguments

Q WAP to perform overloading of + operator for finding a sum of the given class object?

```
Class Sample {
 private:
    int value;
 public:
    sample(int x);
    Sample operator+(Sample obj);
    void display();
 };
 sample:: sample (int x)
  {
      value = x;
  }
 sample:: sample operator+(sample obj)
  {
      sample objsum;
      objsum.value= value + obj.value;
      return(obj.sum);
  }
 void sample :: display()
  {
      cout << "value = " << value;
  }
 main() {
   sample obj1(10);
   sample obj2(20);
   sample sum;
   sum = obj1 + obj2;        // = is overloaded by compiler as we
   obj1.display();              don't create one.
```

```
    obj2.display();
    sum.display();
  }
```

Q. WAP to overload '<' operator to compare two objects of a class

```
Class Sample {
 private:
   int value;
 public:
   sample (int n);
   void display ();
   void operator < (sample obj);
};

 sample:: sample (int n)
  {
      value = n;
  }

 void sample :: display ()
  {
     cout << " value = " << value;
  }

 int sample :: operator < (sample obj)
  {
      return (value < obj.value)
  }

 main()
  {
      sample obj1 (20);
      sample obj2 (10);
      cout << (obj1 < obj2);
      cout << (obj2 < obj1);
  }
```

We can write obj1 = obj2 due to operator overloading

Q. WAP to overload + operator using friend function?

```
Class Sample {
 private:
    int value;
 public:
    sample (int n);
    void display ();
    friend sample operator + (sample A, sample B);
};

 sample:: sample (int n)
  {
      value = n;
  }

 void sample :: display ()
  {
     cout << " value = " << value;
  }

  Sample operator + (sample A, sample B)
   {
        sample sum;
        sum. value = A. value + B. value;
        return sum;
   }
```

Q WAP to overload = for (obj1 = obj2) ?

```
Class Sample {
 private:
    int n;
    int y;
    int z;
 public:
    sample (int n, int y, int z)
    void display ();
```

```
    void operator =(sample obj);
};

sample:: sample (int a, int b, int c)
 {
        x = a;
        y = b;
        z = c;
 }

void sample :: display ()
 {
        cout << " x =  " << x;
        cout << " y =  " << y;
        cout << " z =  " << z;
 }

void sample :: operator = (sample obj)
 {
        x = obj.x;
        y = obj.y;
        z = obj.z;
 }
```

Overloaded assignment operator (=) are not inherited from base class, rest generally overloaded operator inherited to derived class.

<u>When should we write our own assignment operator</u> →

```
Class Test {
private:
   int * ptr;
public:
   Test (int i=0) {
        ptr=new int[i];
   };
```

```
    void setvalue( int i) {
        *ptr = i
    }
  void display( ) {
    cout << * ptr ;
  }
};
main() {
    Test T1(5);
    Test T2;
    T2 = T1;  →  // both will start pointing to same obj but we
    T1.setvalue (10);        want to copy value this is bcz of
    T2.display();            compiler autogenerated overloaded
}                            = operator. (i.e T2.ptr = T1.ptr)
```

∴ Creating own assignment operator

```
void operator = ( const Test &t);

void Test :: operator = ( const test &t)
  {
      * ptr = * (t.ptr)
  }
```

Now the problem is solved. i.e now *(T2.ptr) = *(T1.ptr)

We can't overload :: . ?: *

Multithreading -

- Process - program in execution.
- Process control block going to store info. req. for running a process.
- like process id, priority, registors, program counter (line 1 – line 2)
- Every process is independent to each other
- Every thread is going to execute in diff. core of processor.

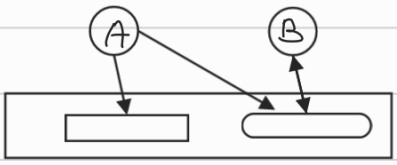
- Thread - A thread is a path of execution within a process, a process can contain multiple threads.
- Thread are light weight process.
- Main idea is to achive concurrarancy by dividing a process into multiple thread.
- The primary diff b/w^n process & thread is that, thread within the same process run in shared memory space while process run in seprate memory space.
- Threads are not independent to each other
- Threads associated with process shares their code section, data section, OS resources.
- But like process A thread have its own programm counter, register set, stack space.
- Contact switching b/w^n threads is faster
- Concurrency - Multiple computations at the same time

  Ex:- Multiple app. running on 1 computer

  Multiple computer running on a network
- Two models for concurrent prog.

① Shared memory model

- In this Model concurrent module interact with each other by reading & writing shared objects in the memory



Shared memory

Ex A & B might be two processors in same computer sharing a physical memory

A & B might be two threads in the same prog.

② Message passing

In this model concurrent modules interact by sending message to each other through communication channel

Ex. A & B might be a web browser & a web server

A & B might be 2 prog. running in same computer where the output

of prog1 is given as input to prog2.

- density of trasistor should be maintained upto a limit ∴ we move to multicore instead of incresing transistor in a core.

Multithreading in c++ -

- The thread class and related funct. are defined in thread header file. (#include< thread >)
- To start a thread we need to create a new thread object and pass the executing code to be called into the constructor of the object.
- Once a thread obj is created a new thread is launched which will execute the code specified in collabal

```
# include< thread >
void function 1( int n )
{
    cout<< " n value in thread = " <<n;
}
main () {
  int y = 10;
  thread T1 (function 1 , y );
      y++;
    cout<< " y = " <<Y ;
  // T1.join();      //Hold main till T1 join main after execution.
     T1.detech ();   // T1 will run seperately, main will execute
}                      independently
```

only once

but we need to check T1 is joinable or not.

```
∴ if (T1.joinable())
        T1.join();
```

```
#include <thread>
void fun(int &x) {
   cout << " x value in thread";
       cout <<x++;
   }
main() {
   int y=10;
   thread T1 (fun, ref(y));
     T1.join();
   cout<<"y in main thread" <<y;
 }
```

Reference.

<u>Race Condition for shared data -</u>

```
void fun1 (int &x)
{
   cout<< " Inside thread function";
   for (i=0; i<=100 ; i++)
     {
          cout <<"x = " <<x++;
     }
}

main() {
  int y = 20;
  thread T1 (fun1, ref(y));
  for (int i=0 ; i< 100; i++);
    {
        cout << "y = " <<y--;
    }
  T1. join();
  cout<< " y in main" << y;
}
```

Random output

- To solve we do lock in of shared variable - #include<mutex>

```
mutex mu;
.
 #include<mutex>
    mutex mu;
void fun1 (int &x)
{ mu.lock();
   cout<< " Inside thread function
   for (i=0; i<=100 ; i++)
     {
          cout <<"x = " <<x++;
     }
mu.unlock();
}
main() {
  int y = 20;
  thread T1 (fun1, ref(y) );
  for (int i=0 ; i< 100; i++);
    {
        cout << "y = " <<y--;
    }
  T1. join();
  cout<< " y in main" << y;
}
```

Thread storage duration -

- The storage for the object is allocated when thread begins and deallocated when the thread ends.
- Each thread has its own copy of object/instance.
- In C++ objects declared with thread_local have thread storage duration.

Ex

```
thread_local int i=0;      //every thread has a copy of i in it
void f (int newval) {
        i = newval;
    }
void g() {
        cout<<i;
    }
void thread_fun (int id) {
    f(id);
    ++i;
    g();
  }
}
main() {
    i = 9;
 thread T1 (thread_fun , 1) ;
 thread T2 (thread_fun , 2) ;
 thread T3 (thread_fun , 3) ;
    t1.join();
    t2.join();
    t3.join();
    cout<<i;
}
```

(i) having thread storage duration