C Programming (1972)

Procedure ~~oriented~~.

Installation.

1. Install the compiler for c/c++. (Mingw)
2. Install IDE or Editor (VScode)
3. Install imp. extentions in (VScode)
   1. c/c++ extentions ~~c/c++~~ (by Microsoft)
   2. Code runner. (by Jun Han)
   3. c/c++ (by Microsoft)

~~Note: set path after i~~

Note: Set path after installing Mingw.
(its v. imp.)

after doing this (above steps) its recommended Resturting PC.

imp ~~libreed~~ libraries

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

int main() {
}
```

execution of.

* used to start the code.

* printf (data);
  used to print the data on screen.

* Scanf ("%od", &<veriable>);
  used to take input from user.

& → 'address of' operator.
\n → new line

Steps of Compilation.

1. Pre processing (removal of comments + adding lib. to actual code + expansion of Macros. Stored in .i file)
2. Compilation (Converted to ~~low lvl or~~ assembly lvl code (saved in .s file))
3. Assembly (converted to machine lvl (i.e Binary))
4. linking (an application which links all Assembly files to make ready to execute)
   - Static linking
   - Dynamic linking.
5. Loading (Memory (RAM))

Tokens

Building Block entities :- variables, literals, functions, etc.
Identifier, Keyword, symbols, etc.

; used to terminate the statement.

Keywords

reserved words used by the language, it can't be defined as variables
C has 32 Keywords :-
example :-
long, unsigned, union, return, if, do, etc.

Identifier

words defined by programmer:

1. Variables
2. functions, etc.

(name given to entities)

* C is a case sensitive.
* C does not allow punchuation Symbols for identifiers.

Tokens Include.

Keywords, Identifier, constant, String litral, symbols, etc.

Keywords :-

auto

Start from let 6.

Variable :-

* A name given to memory location.
* Declaration :-
type variable_name; (only declaration)
or.
type variable_name = variable_value; (declaration & Initialisation)

example
* int a = 10;
* int a;
* int a, b;

int a, b = 1, 2;
// a = 1 ; b = 2

Rules for defining a variable in C.
* Can contain alphabets, digits & underscore (_).
* Start with alphabet and underscore only not digit.
* No whitespaces, reserved keywords, symbols (other then (_)) is allowed.
* Valid ex int a, int atharva, int a1, int _a
* invalid int 1, int 1a, int $a, int a$

Datatypes :-
* Basic Datatypes = int, char, float, double
* Derived Datatypes = array, pointer, structure, union
* Enumeration Datatypes : enum.
* Void Datatype: void.

operator.

Its a symbol used to perform a certain operation.

Types

1 * Arithmetic operators.
2 * Relational Operators.
3 * Logical operators.
4 * Bitwise operators.
5 * Assignment operators

Arithmetic operators.

+ Addition
- Subtraction.
* Multiplication
/ Division.
% Modulus. (remainder)

Relational operators

== is equal to
!= is not equal to
< is smaller than
> is Grater than.
<= is smaller than or equal to
>= is Grater than or equal to

Logical operators.

① &&

Logical And operator: If both the operands are non-zero, then the condition is true (only then)

Example syntax (A && B) (false)

(A && A) true.

②

② ||

Logical OR operator: If any of these two operands is non-zero, then condition becomes true.

③ !

Logical NOT operator: It is used to reverse the logical state of its operand. If condition is true, then Logical NOT operator will make it false.

Ex:- !(A && B) is true.

bitwise operators. (works in Binary).

		and.	or	xor.
a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

* This converts the oprands into binary then perform operations on it.
* Used in time complexity in DSA.

More bitwise operators.

① ~ is binary one's complement op.
② << is binary left shift op.
③ >> is binary Right shift op.

Assignment operators.

① = simple assignment op.
Assigns values from right to left

```
a = 2;
```

Value 2 is assigned to a.

② += Add and assignment op.
Add right & left the assign it to left

```
a += 2;
```

Value (a+2) is assigned to a.

③ -= Subtract and assign op.
Subtract right from left then assigns it to left

```
a -= 2
```

Value (a-2) is assigned to a.

④ *= Multiply and assign op.
Multiplies right & left the assigns it to left

```
a *= 2
```

Value (2a) is assigned to a.

(5) /= Divide and assign op.
divides left with right and assign it to left

a /= 2
Value(a/2) is assigned to a.

Some more miscellaneous op.

(1) sizeof()
Returns size of variable (How many bytes is used by the object inside it ~~<->~~") Prints.
sizeof(a), if a is int, it will return int's size on that architecture.

(2) & (adderess of)
Returns adderss of a varible.
&a returns adderss of a ~~it~~ on memory.

(3) ?: (conditional Expression)

~~<condition> ?: value x~~
<condition> ? value x : value Y
if <conditions> is true then value x otherwise value Y.

(4) * (Pointer)
*a (explained after)

Operator Precedence in C

Catagory	operator	Associativity
Postfix	() [] → . ++ --	L → R
Unary	+ - ! ~ ++ -- (type)* & sizeof	L ← R
Multiplicative	* / %	L → R
Additive	+ -	L → R
Shift	<< >>	L → R
Relational	< <= > >=	L → R
Equality	== !=	L → R
Bitwise AND	&	L → R
BitwiseXOR	^	L → R
Bitwise OR	|	L → R
Logical AND	&&	L → R
Logical OR	||	L → R
Conditional	?:	L ← R
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	L ← R.
Comma	,	

Associativity: In programming languages, the associativity of a operator is a property that determines how operators of the same precedence are grouped in the absence of parenthesis.

Ex 1

Q) Print multiplication table of a no. entered by the user in pretty form.

Format Specifier

its a way to tell the compiler what type of data is in a variable during taking input displaying output to the used.

Syntex :-

```
int a = 2;
printf("%d abc", a);
```

%d: Format specifier (for int)

output :-

2 abc

Imp. Format specifiers :-

1) %c → Character.
2) %d → Integer
3) %f → Floting Point
4) %l → long
5) %lf → double.
6) %Lf → Long double.

Constants

(can't be changed)

Ex :- 15, 23, 3.4, 'a', "Atharva", etc.

* Defination :-

```
#define abc
```

Con

const Keyword

funct^n of <math.h>.

```
pow(x, y);
  //x^y.
```

sin(x); cos(x); tan(x)

asin(x); acos(x); atan(x)
// $\sin^{-1}(x)$; $\cos^{-1}(x)$, $\tan^{-1}(x)$

sqrt(x);
// $\sqrt{x}$

'#' statements are preprocessing statements

examples :-

```
const int a = 15;
#define a 3.14
```

Escape Sequences

* An escape sequence in C prog. lang. is a sequence of characters.
* It doesn't represent itself when used ~~with~~ inside string literal or char.
* It is composed of two or more characters starting with '\'

Example :-

\n (new line)

List.

\a	Alaram or Beep.
\b	Backspace.
\f	Form Feed.
\n	New line.
\r	Carriage return.
\t	Tab (Horizontal)
\v	Vertical tab
\\	Back slash.
\'	Single quote.
\"	Double Quote.
\?	question mark
\nnn	octal no.
\xhh	hexadecimal no.
\0	Null

Comments

* the lines which are written only for understanding the code
* compiler removes this in preprocessing.

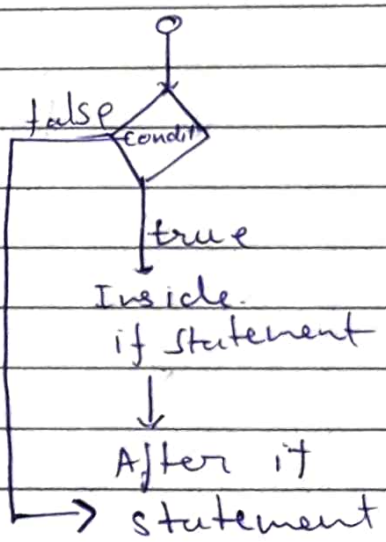
```
// comment (single line)
/*
Comment
multi
line.
*/
```

Controll Statements.

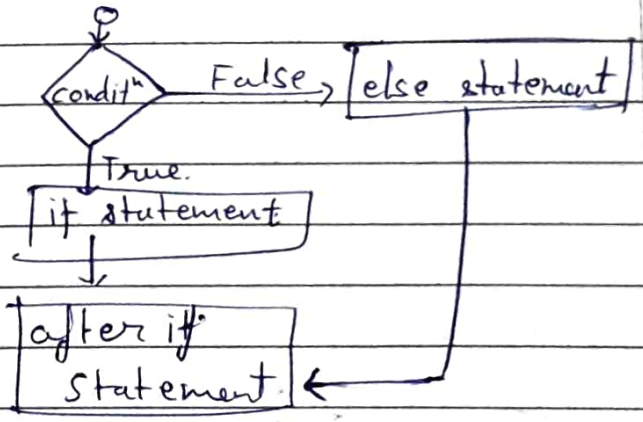
if else statements:-

1) if
2) if else
3) if-else if ladder.
4) Nested if.

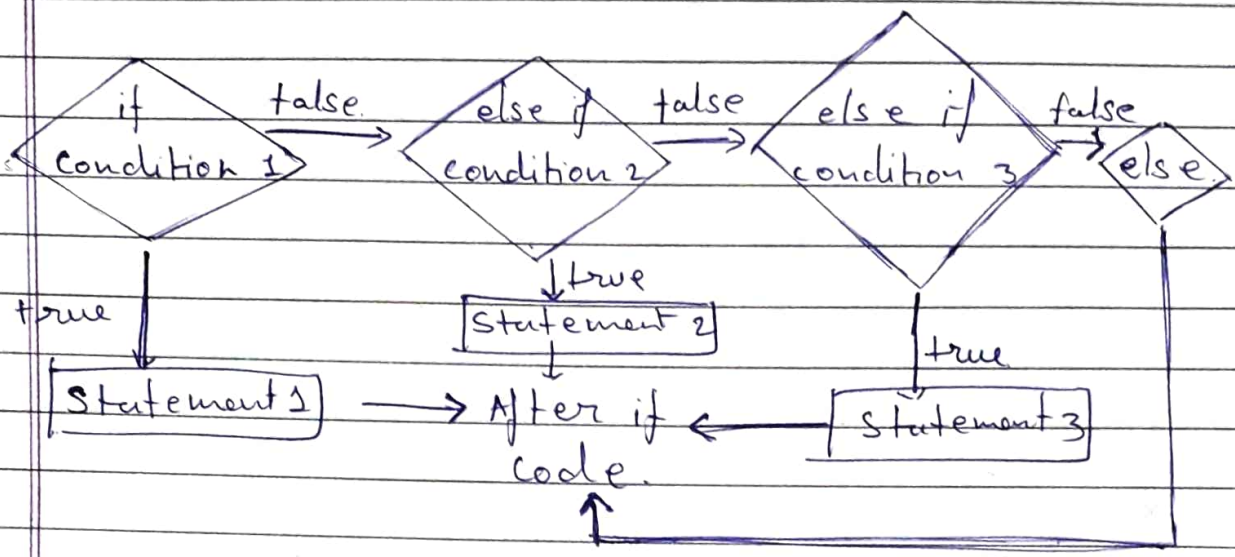
if statement



if, else



Flowchart for if- else if ladder.



Syntax :-

```
if (condit^n) {
  code.
} else if (condit^n 2) {
  code 2
} else if (condit^n 3) {
  code 3
}.
.
.
else {
  code n^th.
}
After if statement.
```

Switch Case Statement
(alternative for if else statements).

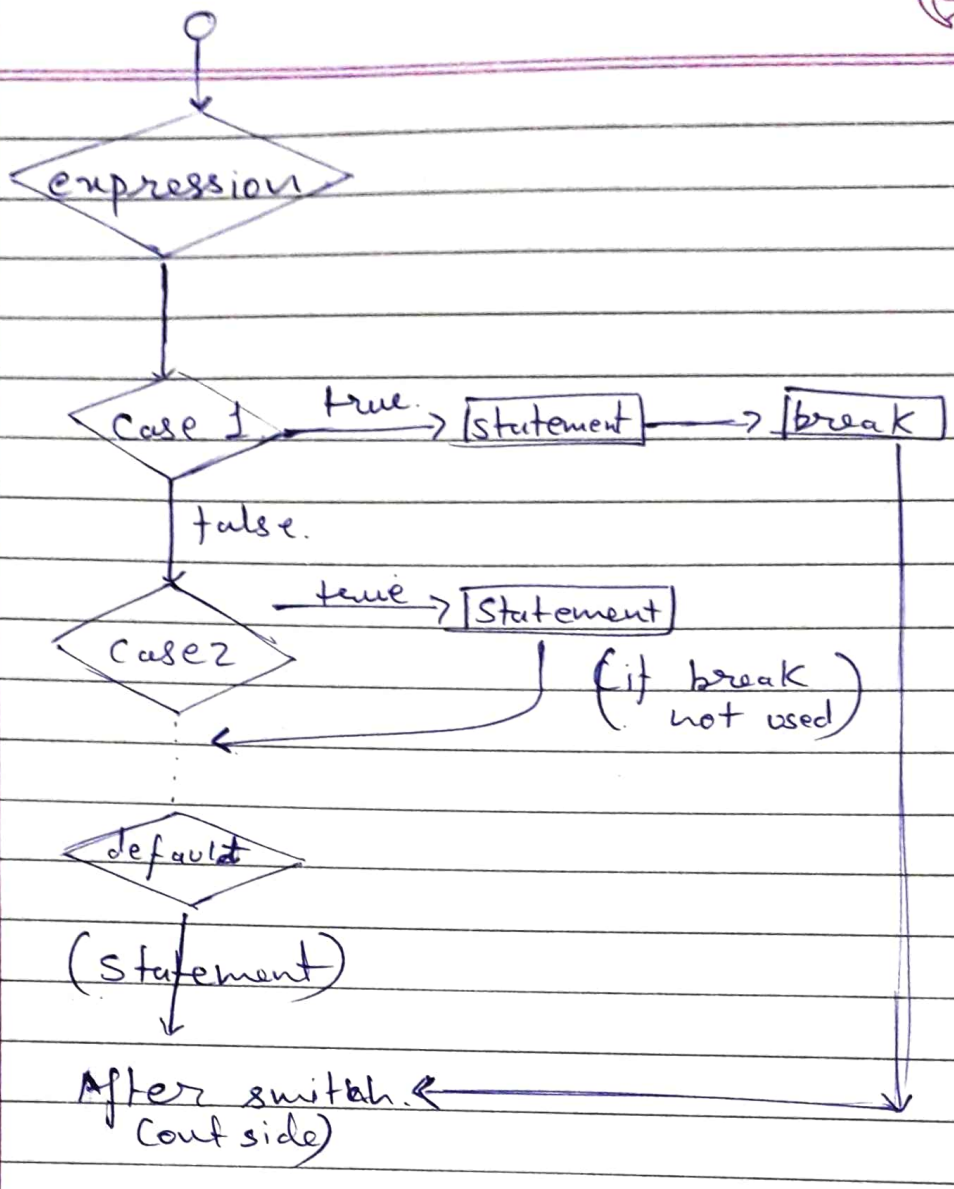
Syntax:

```
int a = 2;
Switch (a) {
   Case 2:
      printf("value is 2"); break;
   Case 3:
      printf("value is 3"); break;
   default
      printf("nothing matched"); break;
}
```

output
value is 2

Rules for switch statement:
1) Switch expression must be an int or char.
2) Case value must be an integer or char.
3) break statement is not must.
4) Case must be inside switch.

flow chart P.T.O →



Loops (let 12)

Advantages of using loops

1) Code reusablity and time saving
3) Traversing

Types of loops.

① do while loop.
② while loop
③ for loop.

Do while loop.

Synten

```
do{
  // cond to be executed.
} while (condition);
```

Example:-

```
int i = 0;
do{
   i += 1;
   printf("%od", i);
} while (i<10);
```

output:-

0 1 2 3 4 5 6 7 8 9.

// after this code will be terminated since.
10<10 is false.

In this loop, it executes the statements inside the do{} atleast once irrispective of the condit^{n} (i.e. first it enters inside, then checks condit^{n}).

While loop.

Synten

```
while (condition) {
   // code to be executed.
}
```

In this loop, unlike do-while loop, the statements inside will only be executed if the condition satisfies (i.e. first it checks the condit^{n} then enter the loop).

for loop.

* The for loop is used to iterate the statements or a part of the program several times.
* It is used to ~~trans~~ traverse the data structures like the arrays and linked lists.
* It has a little different syntax than while and do while loops.

Syntax:-

~~for (Initialization, condit" for terminat" ; Increment or decrement){~~
~~// Code to be run~~
~~}~~

```
for ( Expression 1 ; Expression 2 ; Expression 3 ) {
  // Code to be executed.
}
```

Expression 1 : Initilization.
Expression 2 : Condition for termination.
Expression 3 : Increment or Decrement.

Example.

```
int i;
for ( i=0 ; i<5 ; i++ ) {.
   printf ("%d ", i);
}
```

output

0 1 2 3 4

→ Expression 1 :- (Initialization)

- we can initialize more than one variable Here
- ~~This is~~ exp. 1 is optional.

```
for (i=0, j=0; - - -
```

→ Expression 2 :- (Condit^n for termination)

① It can have more than one condition. However, the loop will iterate until the last condition becomes false. other condit^n will be treated as statements.

② It is optional. C break statement should be used to avoid ∞ loop

③ exp 2 can perform the task of expression 1 and exp 3. That is, we can initialize the variable as well as update the loop variable in exp 2 itself.

④ We can pass zero or non-zero value in expression 2. However, in C, any non-zero value is true, and zero is false by default.

→ Expression 3 :- (Increment or Decrement)

① Exp 3 is used to update the loop variable.

② We can update more than one variable @ the same time

③ exp 3 is optional.

Break & Continue statement :-

① used with loops and switch case.

② Used to bring the program control out of the loop.

③ in nested loops, 'break' only break the loop in which it was.

Continue statement.

* The continue statement skips some code inside the loop and continues with the next intration.
* It is mainly used for a condition so that we can skip some lines of code for a particular condition.

go to statement

* Also called Jump statement.
* Used to transfer program control to a predefined label.
* Goto statement work when we need to break multiple loops using a single statement @ the same time.

goto <label>;

<label>:
// Code to be executed.

Typecasting :-

Converting datatypes :-

Syntax :-

(type) value ;
or.
(type) variable;

Example :-

int a = 10, b = 3;
float result = (float) a/b

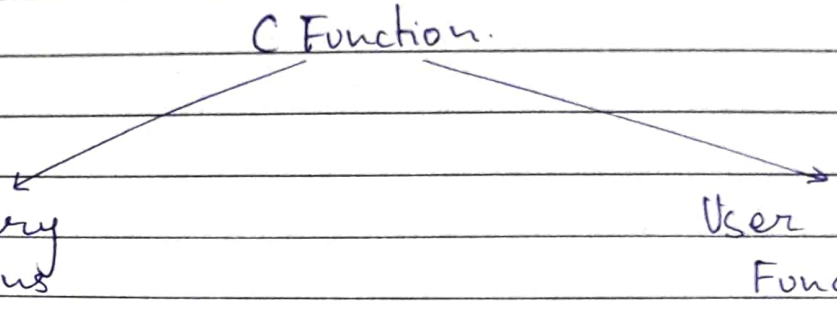
Functions (let #19)

* Also called procedure or Subroutine.
* Can be called multiple times to provide reusability and modularity to the C prog.
* Functions are used to divide a large C program into smaller pieces.

Syntax:-

```
return_type function_name (datatype1 parameter1 ,
datatype2 parameter2, ... ) {
    // Code to be executed
    return <value to be returned>;
}
Void function_name.( datatype1 parameter1, ..){
    //Code to be executed
} // nothing is returned.
```

Declaration, Definition & and Call.



→ Declaration :-

```
datatype function_name (Arguments);
```

Note :- function should be declared or defined b4 Calling.

→ Defination :-

```
datatype function_name (Arguments){
    // Code to be executed
}
```

→ Call

```
a = function_name (Arguments);
// if the function is non void.

function_name (Arguments);
// if the function is void.
```

let go se.

Tower of Hanoi

Date
Page

Recursive function

* Any function which calls itself itself is called Recursive funct°.
* Recursive functions or Recursion is a process when a function calls a copy of itself to work on a smaller problems.
* A termination condition is imposed on such functions to stop them exeecuting copies of themselves forever.
* Any problem that can be solved recursively, can also be solved in iteratively.
* Any problem that can be solved recursively, can also be solved iteratively.

Example:-

```
int factorial (int number) {

    if (number == 1 || number == 0) {
        return 1;
    }
    else {
        return ( number * factorial (number - 1));
    }
}
```

* The case at which the function does't recur is called the base case.
* The instances where the function keeps calling itself is called recursive case

flowchart for the example Previously listed factorial for 5! = ?

The function calling itself.

5 × factorial(4)
↓
5 × 4 × factorial(3)
↓
5 × 4 × 3 × factorial(2)
↓
5 × 4 × 3 × 2 × factorial(1)

Recursive cases

↓ → Base case.

5 × 4 × 3 × 2 × 1 = 120
↓
output

Ex 2

Q) develop a ~~fact~~ unit converter for converting :-
- → kms to miles
- → inches to foot
- → cms to inches
- → pound to Kgs
- → inches to meters.

where it takes from user, what to convert then also takes the value and converts it to respective ~~fac~~ factor given.
Usable again & again until user exits.
Code should be efficient and short.

Row

int arr[2][2] = {{2,3}, {7,8}} ←
int arr[2][3] = {{1,2,3}, {4,5,6}}

Arrays (let → 23)

* An array is a collection of ~~data types of factors~~ data items of the same type.
* Items are stored @ continuous memory locations
* It can also store the collection of derived data types, such as pointers, structures, etc.
* one-dimentional is like lists.
* two-dimentional array is it like table.
* some texts refer to one-dimentional arrays as vectors, two-dimentional arrays as matrices, and use the general term arrays when the number of dimentions is unspecified or unimportant.
* Accessing an item in a given array is very fast.
* 2D arrays makes it easy ~~im~~ in math applicat"(s) as it is used to represent a matrix.
* Each element of array is of same size.
* Each element of array is given an index, by which it makes easy to be accessed.

Synten:-

datatype name[size];
datatype name[size] = {x, y, z ...};
datatype names[rows][culumns];

multi-dimentional array

name of array
name[0] = 0;
index. Value to be stored at that index.

initialization + dicraration
int arr[4] = {0, 1, 2, 3};
int arr[4]; or int arr[] = {0, 1, 2, 3}.

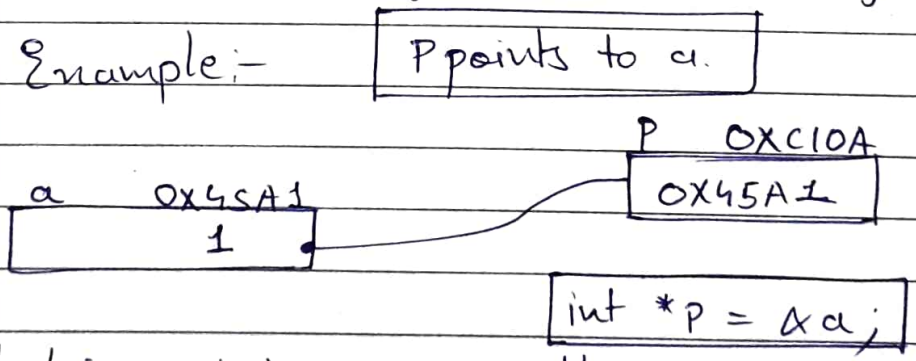
Disadvantages of Arrays.

* Poor time complexity of insertion & deletion operation.
* Wastage of memory since arrays are fixed in size.
* If there is enough space present in memory but not in contiguous form, you will not be able initialize your array.
* It is not possible to change size of array, once you have declared the array.

Pointers

* Variable ~~with~~ which stores the value of another variable
* Can be int, char, array, funct^n, or any other pointer.
* Size depends on the architecture Eg:- 2bytes for 32 bit.
* Declared using * (asterisk symbol)

Example:-



- 'a' is an integer variable.
- 'p' is a pointer to integer.

'0X45A1' is address of a.

- '&' → address of operator
- '*' is the dereference operator (also called indirection operator used to get the value at a given address.

```
printf ("%d", *p);
```

output → 1

```
printf ("%*", p);
```

output => 0x4SA1.

Null pointer.

- A pointer that is not assigned any value but NULL is known as null pointer.
- In computer programming, a null pointer is a pointer that does not point to any object or function.
- We can use it to initialize a pointer variable when the pointer variable isn't assigned any valid memory address yet

Syntax

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

Uses of Pointer:-

- Dynamic memory allocation.
- Arrays, Functions, and Structures.
- Return multiple values from a function.
- Pointer reduces the code and improves the performance

Base Element :- 1st element of an array (having Index=0)

Base address :- address of 1st element of the array.

Date
Page

Arrays and Pointer Arithmatic in C

There are four arithmatic operators that can be used on pointers.

* ++
* --
* +
* -

Explanation with an example:-

Consider of the following:-

```
int arr[10];
```

Then type of this array 'arr' is integer.

* However, 'arr', by itself, without any index subscripting, can be assigned to an integer pointer.

```
int * ptr = arr;
```

*arr points to base element of arr[] *

pointer arithmatic can be applied to arr

&arr[0] } same.
ptr

arr[1] } same.
*(arr+1)
or
arr[i] } same
*(arr+i)

```
int *ptr = arr;
```

then these all are true :-

& arr[0] == ptr; true.

arr[0] == *ptr;

arr[1] == *ptr + 1;

or

arr[1] == *(ptr + 1);

if arr[5] = {1, 2, 3, 4, 5};
then, arr is its pointer.

i.e. below is true.

arr[i] == *(arr + i)

or

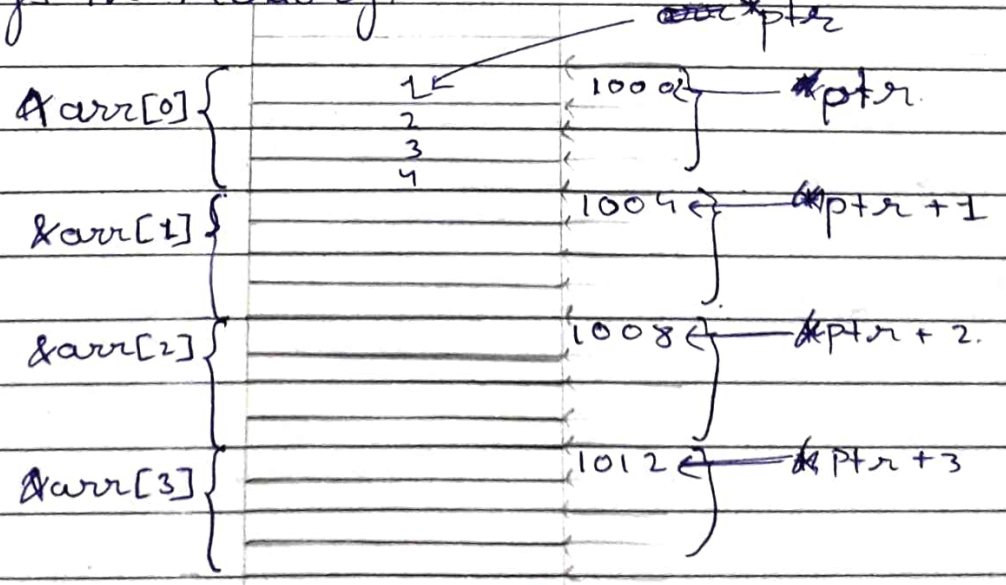
&arr[i] == arr + i

```
arr[i] == *ptr + i
        or
arr[i] == *(ptr + i)
```

this can be used in traversing

pointer arithmatic :- ~~ad~~ arithmatic opprations with address blocks is called pointer arithmatic.

Arrays in Memory :-



Recursions (IN A NUT SHELL)

* Recursion is a good approach when it comes to problem ~~sol~~ solving
* However, programmer needs to evaluate the need and impact of using recursive/iterative approach while solving a particular problem
* In case of factorial calculation, recursion saved a lot of lines of code.
* However in case of Fibonacci series, recursion resulted in some extra unnecessary function calls which was a extra overhead. due to this Running time for Fibonacci series grows exponentially according to input.

Function:- Call by Value VS Call by Reference.

* When a funct^n is called, the value (expression) that are passed in the call are called the <u>arguments or actual parameters</u>

* formal parameters are local variables which are assigned values from the arguments when a function is called.

Formal parameters.

```
int add (int a, int b){
    return a+b;
}
```

```
int main(){
    int x=2, y=3;
    int s=add(x,y);
}
```

This is example of Call by value.

arguments or actual parameters

in call by value a value or a variable is passed in function. Therefore the function could only take value and can not change it.

in call by refrence, address of the variable is passed in function. So function can change the value of actual parameters.

Example

```
func1 (int *a) {
  // code using *a.
}

main () {
int x=7;
  func1 (&x);
}
```

This is example of call by reference.
in this when the function is called, the value of 'x' is changing.

Q) Given two no.s 'a' and 'b', add them then subtract them and assign them to 'a' and 'b' using call by reference.

Example

Input	output
a = 4	a = 7
b = 3	b = 1

* base address of an array = address of 1st element of the array

Passing Arrays as function Arguments.

We pass arrays to a funct^n when we need to pass a list of values to a given fxn.

Passing the array to a function by:-

1 → declaring array as parameter in the fxn.
2 → declaring a pointer in the function to hold the base address of the array

(Method 1)

```
int func(int arr[]){
  for ---
   {
     Sum = Sum + arr[i]
   }
  return sum;
}

int main(){
   int arr[] = {1, 2, 3 ...}
   int SUM = func(arr);
      return 0;
}
```

(Method 2)

```
int func(int* ptr){
  for-<i> ---
   {
     sum = sum + *(ptr+ i)
   }
    return sum;
}

int main(){
  int arr[] = {1,2,3 - - }
  int SUM = func(arr);
  return 0;
}
```

Imp.
Inside func, if you change the value of the array, it gets reflected in the main function.

Same in both cases

Strings (let → 34)

* String is not a datatype in C. We have char, int, float and other data types but no 'string' data type in C.
* String is not a supported data types in C but it is a very useful ~~con~~ concept used to model real world entities like name, city etc.
* We express string using an array of characters terminated by a null character ('\0').

Syntax example

→ char name[] = "Atharva";

→ char name[] = {'A','t','h','a','r','v','a','\0'}

Note: " " is used to store it as string, ' ' is used to store char.

gets(); is used to input string in C

```
char str[52];
gets(str);
```

formate specifier "%s"

```
printf("%s", str);
puts(str); ——→ also used to print string. (Alternative for printf("%s", str); only for string.
```

String functions

C library

<string.h>

Function	use	syntax
strcat()	used to concatenate or combine two given strings	strcat("Hello", "World"); output → HelloWorld.
strlen()	used to show length of a string (not includes '\0')	strlen("Hello"); output → 5
strrev()	used to show reverse of string.	strrev("Hello"); output → olleH
strcpy()	used to copy one string into another	strcpy(*s2, s1); copies s_1 to s_2
strcmp()	used to compare two given strings	returns difference of # of ASCII no.s of 1st unmatched character.

Structures (let → 37)

* Structures are user defined data types in C.
* Using structures allows us to combine data of different datatypes together.
* Similar to array but can be stored different data types.
* Used to ~~store~~ create a complex data type which contains diverse info.

Syntax:

```
struct [structname]
{
datatype variable1
datatype variable2
.
.
} [struct variables];
```

its not variable, its a datatype like int, float, etc but unlike int, float structure is user defined.

Methods of Defining:-

```
struct emp {
int id;
char name[53];
float marks;
};
struct emp e1, e2;
```

Datatype variables

```
strut emp{
int id;
char name[53];
float marks;
} e1, e2;
```

initialisation of structure.

```
struct emp{
int id;
float marks;
}
int main(){
struct emp e1;
e1.id = 12;
e1.marks = 34.12;
struct emp e2;
e2.id = 10;
e2.marks = 36.14;
return 0;
}
```

```
struct emp{
int id;
float marks;
}
int main(){
struct emp e1 = {12, 34.12};
struct emp e2 = {10, 36.14};
return.
}
```

Instance method

arr[0] or arr[1] or so on.....

* array elements are accessed using the subscript variable.

* In a similar fashion, structure member are accessed using dot operator "." or "structure member operator".

Syntax:-

~~Structure name~~

structure_name.member_name = value;

Type def.

Used to give kick names to datatypes.

* Example syntax:-

```
int main(){
    typedef unsigned long ul;
    ul l1, l2, l3;
    return 0;
}
```

Previous name of datatype

user defined new name for the datatype

using the new name or alias name.

typedef <Previous name> <new name>;

* another example syntax.

```
typedef struct student{
    int ID;
    int marks;          // previous name
    char name[15];
} stu;                  // new name

int main(){
    stu s1, s2;         // using new name.
```

Unions (similar to structure)

* Can be used (Alternative to struct) for saving memory!
* Union is a used defined data type (very similar to structures)
* The difference b/w structures and unions lies in the fact that in structure, each member has its own storage locat^n, whereas members of union uses a single shared memory location.
* This single shared memory location is equal to the size of its largest member.
* Syntax is very similar to structure.

```
struct student {
  float marks; //4bytes
  int id; //4bytes
} s1;
```

↓

8 bytes

* all the members can be used at a time
* both can be stored at the same time.

```
union student {
  float marks; //4bytes
  int id; //4bytes
} s1;
```

↓

4 bytes (shared b/w marks & id)

not all the members can be used at a time

Here, if marks is stored, id will removed and if id is stored, marks will removed (values for marks and id)

Static Variables in C (let → 42)

Scope: it is a region of the program where a defined variable can exist and beyond which it cannot be accessed.

★ Note :→ If a local and global variable has the same name, the local variable take preference.

→ Static variables are variables which have a property of preserving their values even when they go out of scope.
→ They preserve their value from the previous scope and are not initialized again.
→ Static variables remains in memory throughout the span of the program.
→ Static variables are initialized to 0 if not initialized explicitly.
→ In C, static variables can only be initialized using constant literals.

syntax :-

static <data_type> <variable_name> = <variable_value>;

example:-

static int a = 2;

Q) You manage a travel agency and you want your 'n' drivers to input their following details:-

1.) Name (string)
2) Driving license no.
3) Route (string)
4) Kms.

Your program should be able to take 'n' as input and your drivers will start inputting their details one by one.
Your program should print details of the drivers in a beautiful fashion.

Dynamic Memory Allocation.

Static Memory Allocation	Dynamic Memory Allocation.
* Allocation is done b4 the program's execution	Allocation is done during the program's execution
* There is no memory reusablity and the memory allocated. Cannot be freed.	There is memory reusablity and the allocated memory can be freed when not required

Stack → LIFO (Last in First out)

Date
Page

Memory assigned to a program in a typical architecture can be broken down into four segments.

1) Code → Text Segment
2) Static / Global → data segment (initialized)
   → bss segment (uninitialized)
3) Stack → (for static memory Allocat^n)
4) Heap → (for dynamic memory Allocat^n)

Stack overflow :-

→ Compiler allocates some space for the stack part of the memory
→ When this space gets exhausted for some bad reason, the situation is called as stack overflow.
→ Typical example includes recursion with wrong/no base condition.
→ There are lot of limitations of stack (static memory allocation) so,

→ Use of Heap

* Some of the examples include variable sized array, freeing memory is no longer required. etc.

* Heap can be used flexibly by the programmer as ~~per~~ per his needs.

Function for Dynamic Memory Allocation.
They all are present inside <stdlib.h>

We have four functions that help us achieve this work

* Malloc
* Calloc
* realloc
* free.

This way we can change size of a data structure in runtime.

malloc()

memory allocation.

It reserves a block of memory with the given amount of bytes.

* The return value is a void pointer to the allocated space
* ∴ the void pointer needs to be casted to the appropriate type as per the requirements
* However, if the space is insufficient, allocation of memory fails and it returns a NULL pointer
* All the values at allocated memory are initialized to garbage values.

Syntax:-

```
ptr = (ptr-type*) malloc (size_in_bytes);
```

Example.

```
int* ptr = (int*) malloc (3* sizeof(int));
```

calloc()

Contiguous allocation.

* It reserves n blocks of memory with the given amount of bytes.
* The return value of is a void pointer to the allocated space
* Therefore, the void pointer need to be casted to the appropriate type as per the req.
* However, if the space is insufficient, allocation of memory fails and it returns a NULL pointer.
* All the values at allocated memory are ~~initialize~~ initialized to 0.

Syntax:

```
ptr = (ptr_type*) calloc (n, size_in_bytes);
```

realloc()

re allocation.

* If the dynamically allocated memory is insufficient we can change the size of previously allocated memory using realloc() function.
* Syntax.

```
ptr = (ptr_type*) realloc (ptr, new_size_in_bytes);
```

free();

- free() is used to free the allocated memory
- If the ~~object~~ dynamically allocated memory is not req. anymore, we can free it using free function.
- This will free the memory being used by the program in the heap.

Syntax

```
free(ptr);
```

Q) ABC Pvt Ltd. manages employee records of other companies. Employ Id can be of any length & it can contain any character for 3 employees, you have to take 'length of employee id' as input in a length integer variable then, you have to take employee id as an input and display it on screen. Store the employee id in a character array which is allocated dynamically you have to create only one character array dynamically.

Storage Classes.

→ A storage class defines scope, default initial value & lifetime of a variable.

Scope
region of application of a variable.

if a varible 'a' is defined in "int main()" then it will work only in "int main()" outside it, it will considered as not defined.
∴ int main() is scope for variable 'a'.

Default initial value
value by which a variable is initialised.

~~dolt~~ Lifetime.
time period of applicat^n of a variable during runtime.

types
1) Automatic Variables
2) External Variables
3) Static Variables
4) Register Variables.

Automatic Variables.
Scope: Local ~~variables~~ to the funct^n body they are defined
Default Value: Garbage Value.
Lifetime: Till the end of the fun block they are defined in.

int a; & auto int a; is same.
by default a variable is defined Auto.

External Variables
(or gobal variables)
Scope :- full program.
Default value :- 0
Lifetime :- throughout the lifetime of the code execution.

int a; written outside any function. in the program.

Static Variables.
Scope :- local to the block they are defined in
Default Initial Value: 0
Lifetime: They are available throught the lifetime of Code.

Syntex :-

static int harry;

Register Variable.
Scope :- local to the funct^n
Defaut :- Garbage.
Lifetime :- till end of funct^n block.

* Register variables requests the compiler to store the variable in the CPU register insted of storing in the memory to have faster access.

Types of Pointer.

Void Pointer.

→ A void pointer is a pointer that has no data type associated with it.
→ A void pointer can be easily typecasted to any pointer type.
→ In simple language it is a general purpose pointer variable.
→ It can be easily typecasted to any pointer type

In dynamic memory allocation, malloc() ; calloc() return (void *) type pointer.
~~This allows~~
rules:-
1) no dereferencing without type casting.
2) Pointer arithmatic is not allowed.

Null Pointer.

* Null pointer is a pointer which has a value reserved for indicating that the pointer or reference does not refer to a valid object.
→ Syntax:-

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

→ A null pointer should be guaranteed to compare unequal to any pointer that points to a valid object
→ Dereferencing a null pointer is undefined in C [Do not ~~use~~ dereference it without pointing it to a valid ~~po~~ container]

NULL pointer vs uninitialized pointer.
An uninitialized pointer contains garbage values. where null is not having any value.
(i.e. ((void*)0))

NULL pointer vs Void pointer.
Null pointer is a value whereas void pointer is a type.

Dangling Pointer.
This can generate issues in the code
* A pointer pointing to a freed memory location or the location whose content has been deleted is called a dangling pointer.

Causes of Dangling pointer:-
↳ Deallocation of memory
↳ Returning local variables in function calls.
↳ Variable is going out of scope.

Example:-
```
#include <stdio.h>
int* myfunc(){
 int a=4;
 return &a;
}
int main(){
 int *ptr = myfunc();
 printf("%d", *ptr); // ptr is dangling pointer.
 return 0;
}
```

Wild Pointer.

* Uninitialized pointer are known as wild pointer
* These pointers point to some arbitrary location in memory and may cause a program to crash or behave badly
* dereferencing wild pointers can cause many bugs
* It is suggest to always initialize unused pointers to NULL.

Q) Exercise → 9 (Rock, Paper, Scissors)

prerequisits :-

time.h functions :-

→ srand(time(NULL));

//srand takes seed as an input and is defined inside stdlib.h.

→ print("random no: %d \n", rand() % 100);

// prints random no b/w 0 to 100.

Create Rock, Paper, Scissor

player 1 : human.

player 2 : Computer.

Write a code which allow user to play this game three times with Computer. Log the scores of computer & the player Display the name of the winner @ the end. you have to display name of the player during the game. Take user's name as input.

Matrix Multiplication ..

Q) Ex -> 10 (Matrix Multiplication)
take 2D arrays from user, if matrix multiplicat" is possible, do it; if not display error. size of the arrays from user and also values

C Pre-Processor Intro & working.
(let → 58).

* pre processor comes under action b4 the actual compilation process.
* its not part of compiler.
* Its a txt substitution tools
* All preprocessor Commands begin with a hash symbol (#)

Examples :-

```
# define    // defines a macro
# include
# undef     // undefines a macro
# ifdef     //
# ifndef
# if
# else
# elif
# endif
# Pragma
```

#include Derivative.

→ The #include derivative causes the prepro. to fetch the contents of some other file to be included in the present file.

→ Commonly, ".h" extension files (i.e. header files) is included with this.

formats.

```
#include <File.h> // angle brackets say to look
                  // in standard system
                  // directories.

#include "myFile.h" // look in the current
                    // directory.
```

#define Directive.

→ The #define directive is used to "define" prepro. "variables"

→ The #define prepro. directive can be used to grobally replace a word with a number.

→ It acts as if an editor did a gr: global search-and-replace edit of the file.

#define for debugging.

→ this can be used for debugging be enabling print statements that we ~~want~~ only want active when debugging.

→ we can "protect" them in a "ifdef" block.

Macros using #define.

→ Macros operate much like fns, but b'coz they are expanded in place and are generally faster.

example.

Syntax :-

```
#define Sq(r) r*r

int main(){

  print("Area: %d", Sq(5));
  return 0;
}
```

output will be :-

Area: 25

~~# Und~~

other

→ __DATE__

~~"MMM DD YYYY"~~

"MMM DD YYYY" Jan 3 2019.

→ __TIME__

"HH:MM:SS" 12:41:32.

→ __FILE__

* The current filename as a string literal.

→ __LINE__

* The current line number as a decimal Constant.

→ __STDC__

* Defined as 1(one) when the compiler complies with ANSI standards

Example syntax:-

```
printf("Line no: %d\n", __LINE__);
printf("Time: %s\n", __TIME__);
printf("Date: %s\n", __DATE__);
printf("Line no: %od\n", __LINE__);
printf("ANSI: %od\n", __STDC__);
```

File I/O in C [Let → 62] to 66

Volatile Memory	Non-Volatile Memory
This is computer storage that only maintains its data while the device is powered	Non-Volatile memory is Computer memory that can retain the stored info. even when not Powered.
The RAM	The ROM or Internal Storage (EEPROM). (Hard Disk, SSD, etc.)
The Volatile memory will only hold data temp.	It is used for long term

Type of files.
* Text files.
* Binary files.

High lvl operat^{n} (s) on files:-
1) Creating a new file.
2) Opening a file
3) Closing a file
4) Reading from or writing to a file.

palindrome
a string which remains same after reversing.
example "ANA", "PNP", "PNNP"

Exercise.

Q) take a string input from user and check it, if it is a palindrome or not and output according ly.

Functions for file I/O (let → 64) to 66

Opening a file.

fopen()

Syntax:

ptr = fopen("fileopen", "mode");

Examples.

fopen("E:\\Code\\Ath.txt", "w");

Closing a file

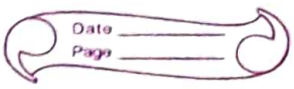
fclose()

Syntax:

fclose(ptr); // ptr is the file pointer.

* Syntax for creating a NULL file pointer:-

```
FILE *ptr = NULL;
```



Reading a file

fscanf();

Syntax:-

```
fscanf (ptr, "%s", string);
```

Writing to file

fprintf();

Syntax:-

```
fprintf (ptr, "%s", string);
```

Syntax for this functions:-

```
FILE * ptr = fopen ("filename", "mode");
```

↓ name of the file

↓

"r" → reading
"w" → writing.
"a" → append.
"r+" → reading + writing
"w+" → reading + writing
+
truncates the file to zero, if it exits. if not it creates a file.

"a+" → reading + writing
creates file if it not exists, reading from beginning but writing can only append to file.

Mode	Description
r	Opens an existing text file for reading
w	Opens a file for writing. If it doesn't exist, then a new file is created. Writing starts from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. The program will start appending content to the existing file content.
r+	This mode will open a text file for both reading and writing
w+	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only append to file.

<math.h>

x & y are varibles (~~for~~ arguments)

→ sin(x); cos(x); tan(x);

→ asin(x); acos(x); atan(x);
// $\sin^{-1}(x)$ $\cos^{-1}(x)$ $\tan^{-1}(x)$

→ pow(x, y);
// x^y

→ sqrt(x); cbrt(x);
// $\sqrt{x}$ $\sqrt[3]{x}$

→ exp(x); log(x); log10(x); exp2(x);
// e^x $\log_e(x)$ $\log_{10}(x)$ // 2^x

→ floor(x)
// rounded ~~off~~ down to its nearest int.

→ fabs(x)
// absolute value.

Q) You have to fill in values to a template "letter.txt". letter.txt:-

"Thanks <name> for purchasing <item> from our outlet <outlet_name>. Please visit our outlet <outlet_name> for any problems. We plan to serve you again".

Use file function for the same.

Other file I/O functions:-
(stdio.h)

* fputc → char. } Writing.
* fputs → string. }
* fgetc → char. } Reading.
* fgets → string }

fputc

syntax → int fputc (<char>, <File_pointer>);

* It returns the written char, on success and returns "EOF" when failure.

"EOF" → "End Of File" a constant defined in stdio.h.

fputs

syntax → int fputs (const char *s , File *P);

null terminated string.

fgetc

syntax → int fgetc (<File-Pointer>);

* returns the read char on success & returns the "EOF" on failure.

fgets

used to read a null terminated string to a file

~~Syntax → int fgets (const char* s, File* fp);~~

syntax → int fgets (const char* s, int n, File* fp);

s → stores the readed string in this

n → no. of char (including null char)

Command line Arguments :-

* Command line Arguments are used to supply parameters to the program when it is invoked.
* Arguments can be passed from the command line to the program a set of inputs
* you can control your program from the console.
* These arguments are passed to the main() method.

Examples:-

* FFmpeg is a free & open-source project consisting of a vast software suite of libraries and programs for handling video, audio, and other multimedia files and streams.
* Ffmpeg.exe is a command line utility written in c language.
* Other examples include git, brew, apt-get, etc.

Example syntax:-

```
#include <stdio.h>

int main(int argc, char const *argv[]){
  // code
  //->argc is no. of arguments passed in command
  // line.
  //-> argv is the arguments  //  //  //.
return 0;
}
```

Q) You have to create a command line Calculator (add, subtract, divide, multiply) two no.s. 1st argument of command line should be oppration followed by two no.s.

Example

Command line (Terminal).

>> Calculator.exe + 1 2

output :-

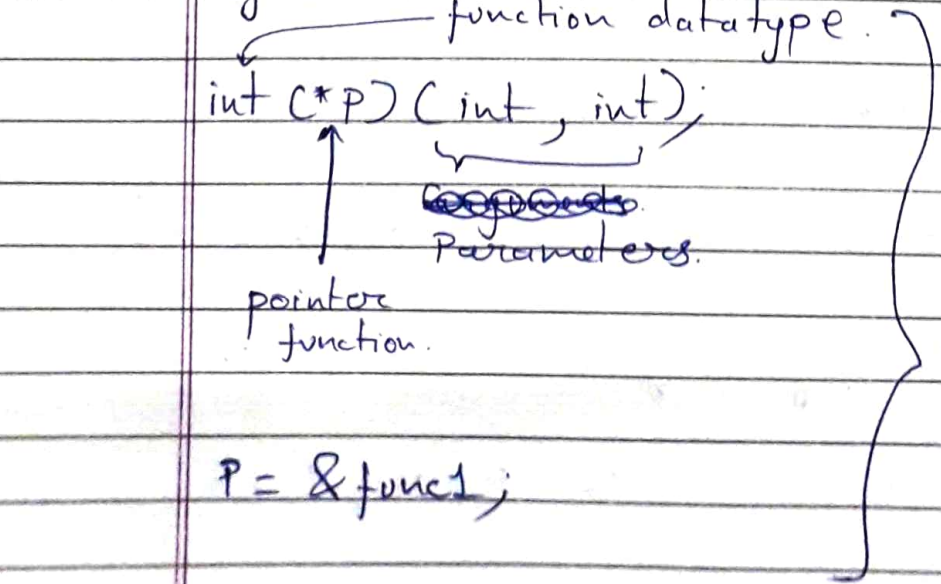
>> 3

function pointers.

(used to implement callback functions)

* We have pointers pointing to functions
* Compiler takes one or more source file and converts them to machine code.
* Unlike normal pointers, we don't allocate memory using this.

Syntax.



Syntax:-

```
int sum(int a, int b){  // defining the function.
    return a+b; }

int main(){
    int (*fPtr)(int, int);  // defining the funct^n ptr.
    fPtr = &sum;  // assigning location of "sum" to the
                  // fPtr.
    int d = (*fPtr)(4,6);  // dereferencing
    printf("The value of d is %d", d);
    return 0;
}
```

output:-

10

Call back functions (use of function pointers)

* function pointers are used to pass a function to a function.
* This passed function can then be called again (hence the name callback function).
* This provides programmer to write less code to do more stuff.

Syntax :-

```
int avg(int a, int b){ return (a+b)/2;}
int sum( int a , int b){
    return a+b;}

Void Hello (int (*ptr)(int, int) ){
    printf ("Hello\n");
    printf ("sum of 5 & 7 is %d", ptr(5,7));
}

int main(){
    int (* p) (int, int);
    p = &avg;
    int (*q) (int, int); // Every prantheis is imp.
    q = &sum;
    Hello (p); // avg of 5 & 7.
    Hello (q); // sum of 5 & 7.
```

Q) you have to ~~cal~~ calc. area of circles. for that you have to 1st calc. radius (dist. b/w two curtician pts (x_1, y_1) & (x_2, y_2))
x_1, y_1, x_2, y_2 is user inputs.

~~int~~ funct^n for calculating dist.
usin (x_1, y_1, x_2, y_2) → Radius.

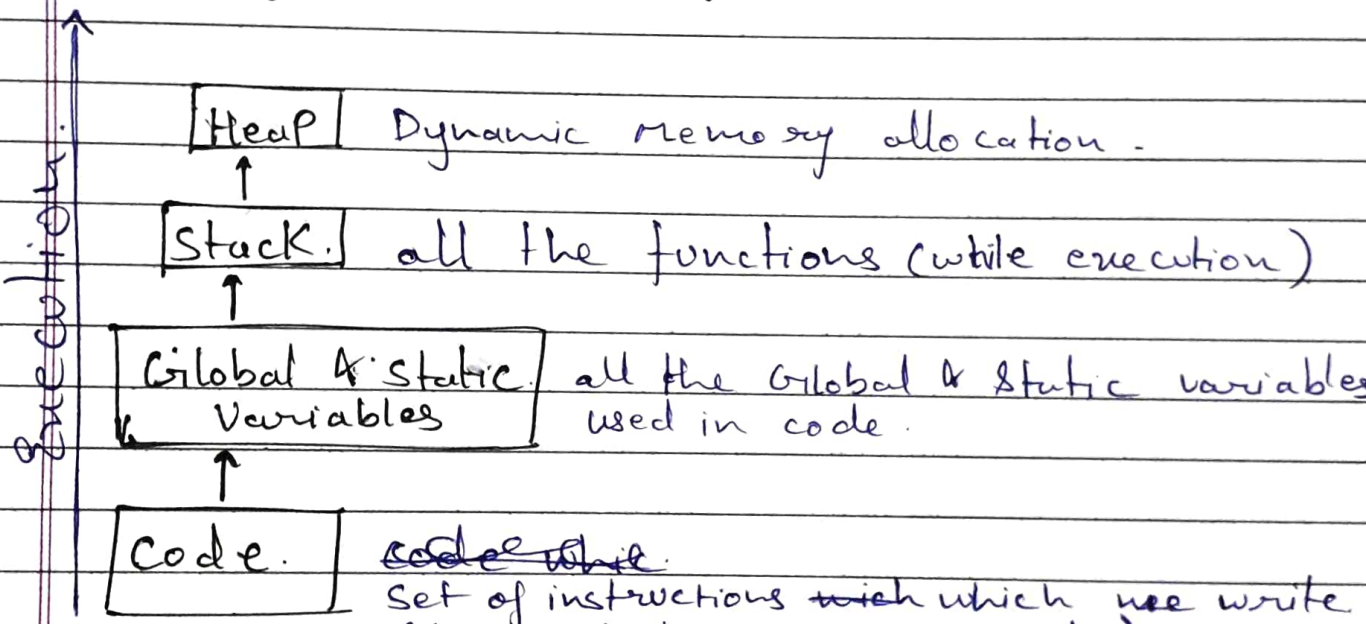
funct^n for calculating area of circle.

→ area.

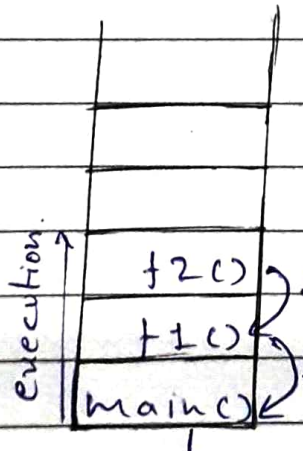
Memory leak in C.

When you create a memory block using Dynamic memory allocation functions in Heap and forgot to free the memory block even after the use of that entity, many garbage memory block will be created in heap. It will consume memory and can cause Malfunctions this is memory leak.

Memory Structure of C:-



Stack:- if f1 & f2 is called inside main. & f2 is called in f1.
(last in first out)



Cause:-
When we keep on allocating memory (using Dynamic memory allocation) in the heap without freeing, the overall memory usage keeps on increasing.
When this happens the code keeps consuming memory and if full memory is consumed eventually system crashes (os crashes).

code with harry : C Programming

C++ 1979

Date
Page

→ Installation is included in C learning.

major updates:-

2011 → C++ 11

2014 → C++14

2017 → C++ 17

→ main used library for I/O:-

```
# include <iostream>
```

→ function for start the exicut" of code.

```
int main() {
    // code
    return 0;
}
```

→ Variables = Containers to store ~~your~~ your data
Variable (scope), defination, initialization and usage; Comments same as C.

cout Syntex:-

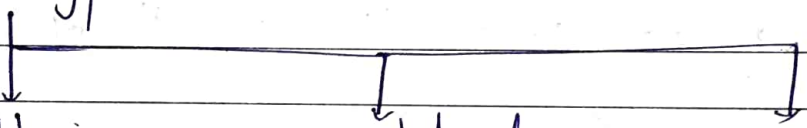
```
int main(){
    int sum = 6;
    std::cout << "Hello World!" << sum;
    return 0;
}
```

or.

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello world";
    return 0;
}
```

Data types :-



Built-in	User-defined	Derived
* int	* struct	* Array
* float	* Union	* func*
* char	* Enum	* ptr.
* double		
* bool		

Note : not have format specifiers, Escape sequences same as C.

Local > global verible.
(if they have same name)

Note : Variable name in c++ can range from 1 to 255 characters

'<<' is called insertion operator
'>>' is called extraction operator.

Date
Page

Basic I/O :-

```
int num1;
Cout << "Enter the value of num1 : ";
Cin >> num1;
```

Header files

There are two types of Header files

1) System header files: It comes with the compiler
2) User defined header files: It is written by the programmer.

Operators :-

1) Arithmatic :- Same as C.

a++
a--

```
Cout << a++ } increment or decrement after
Cout << a-- } printing

Cout << ++a } printing after increment or
cout << --a } decrement
```

2) Assignment :- Same as C.
3) Comparision or relational operators :- Same as C.
4) Logical operators :- Same as C.

:: Scope resolution operator.

c → local. (Precedence)
::c → Global

sizeof() → Same as C.

Literals

34.4 → double
34.4f or 34.4F → Float
34.4l or 34.4L → long double.

Reference variable.

```
float x = 455;
float &y = x;
      └→ Reference variable
```

typecasting.

Same as C.

datatype (variable); is also valid.

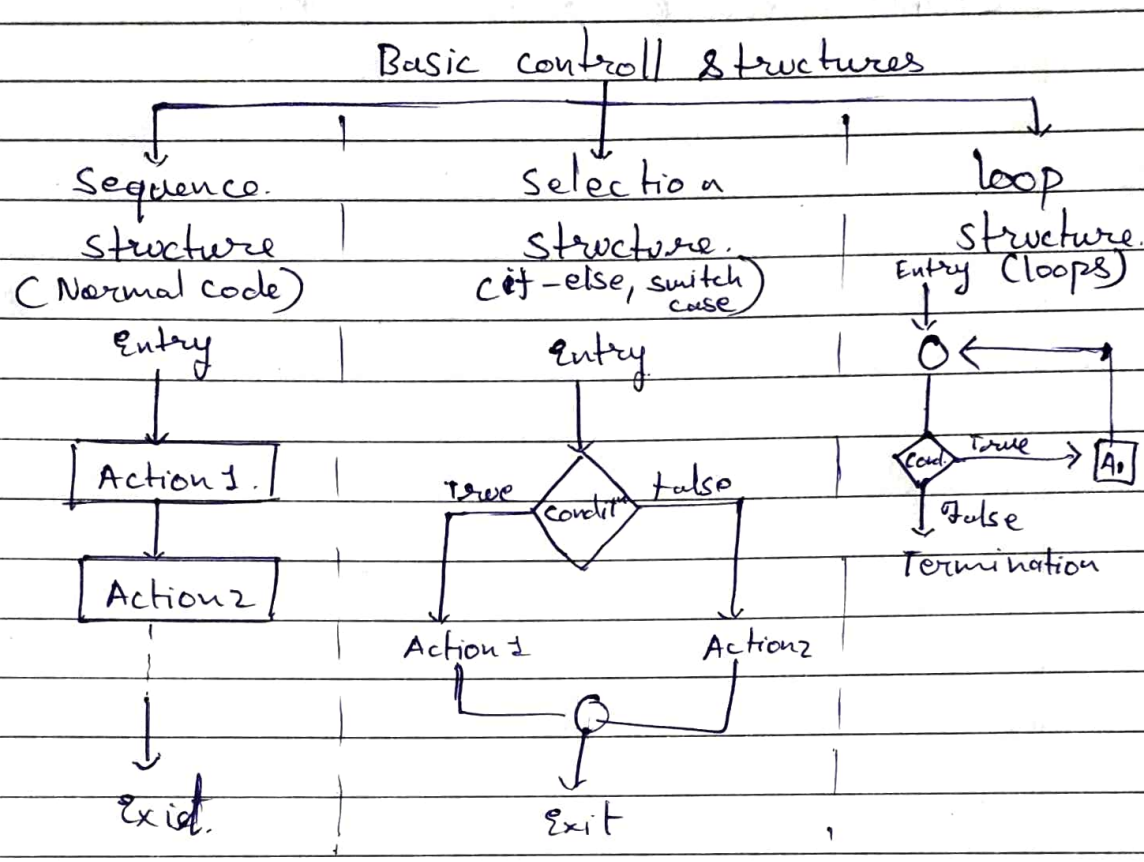
Manipulator.

functions used with output stream 'cout' for formatting

1) endl
for new line.

2) setw (width_to_be_set)
for setting width of entity to be displayed using cout

Controll Structures.



- # if-else statements. and (ladder also)
- # Switch Case.
- # loops.
- # break & Continue statement

} Same as C

Pointer.

Same as C

```
int a = 3;
int *b = &a;
int **c = &b;
```

↳ Pointer to Pointer.

Array & Pointer arithmetic

Same as C

Structures, Unions & Enums.

Same as C

Enums

```
                0       1       2
enum Meal {breakfast, lunch, dinner};
```

Cout

```
cout << breakfast      output -> 0
cout << lunch          output -> 1
```

Functions & Function Prototypes.

Declaring a fxn b4 calling it if defined after calling.

Same as C

Declaring :-

```
int Sum(int a, int b);   ✓
int Sum(int a, b);       ✗
int Sum(int, int);       ✓
```

Call by value & Call by reference

Same as C

Using Reference Variable.

Synten:-

```
void swap (int &a , int &b) {
    int temp = a;
    a = b;
    b = temp;
}

swap (x, y);
```

let -> 16

Inline functions — it is a request to the compiler, its completely on compiler Weather to Accept it or ~~reject~~. ignore

~~Duty~~ recomended to ~~use~~ use where the function is very short and have minimum complexity.

Synten:-

```
inline int product (int a , int b) {
    return a*b;
}
```

When it is called, compiler directly run the statements inside the fxn with arguments ~~on~~ without copying the arguments in formal parameters and do that long stuff.

Note:- in case of Recursions, use of Static variables its not reccommended to use inline fxns.

Default Arguments

Syntax (definition)

```
float moneyReceived(int currentMoney, float factor=1.04){
    return currentMoney * factor;
}
```

Syntax (calling)

```
moneyReceived(10000);
    // its using 'default' value of factor i.e. factor = 1.04.

moneyReceived(10000, 2.04);
    // its using 2.04 as value of factor.
```

Note all default argument are written after all the mandatory arguments.

```
float function(int a, int b=10){
    return a+b;
}
```

✓

```
float function(int a=10, int b){
    return a+b;
}
```

✗

Constant Arguments.

used with pointers as arguments or call by reference, as in it the value of argument can be changed.

```
int string_length ( const char *p){
  //
}
```

it is used such that value of 'p' should not be changed.

Recurssions.

Same as C

Function. over loading.

it allows multiple functions to have the same name, as long as their parameters are different in type or number.

```
int sum( float a, float b) {
    cout << " 2 args";
    return a+b;
}
int sum( float a, float b, float c) {
    cout << " 3 args";
    return a+b+c;
}
```

Object Oriented Programming (OOP)

in C++

OOP is about creating "object", which can hold data & functions that work on that data.

Why OOP.

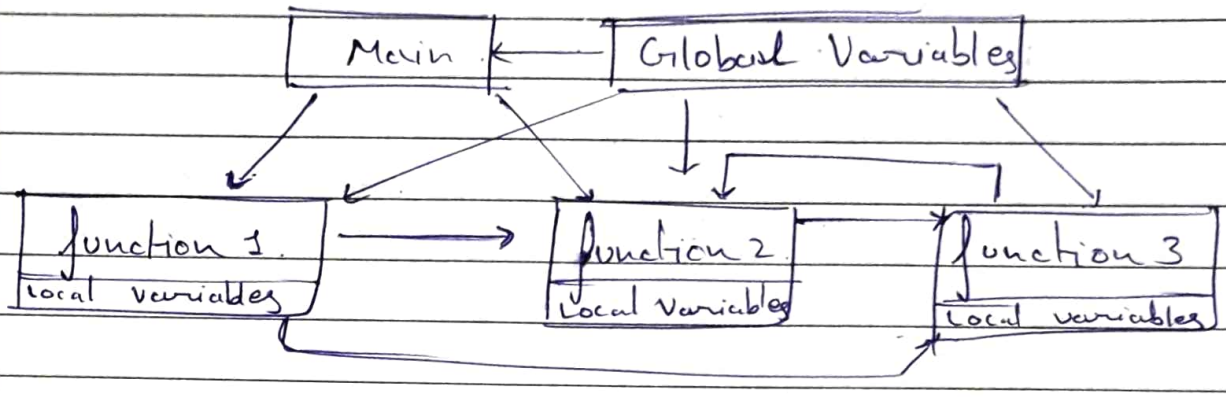
→ As the size of Program increases, readability maintainability & bug-free nature of ~~pro~~ programs decreases

→ POPL → relied on functions & procedure.
OOPL → relied on classes & objects.

→ Data security is major issue in POPL languages like C since data is freely flows b/w functions without any security and can be accessed by anyone.

→ In oopL objects are created using a well defined templete (i.e "classes") which have variables & functions defined in it.
→ Multiple objects can be created & used on the behalf ~~of~~ of this classes.

POPL



OOPL

* Works on the concept of class & objects
* A class is a template to create objects.
* Treat data as critical element

Terminology for OOP's.

* classes → Basic template for creating objects
* Objects → Basic run time entities.
* Data Abstraction & Encapsulation.
  Wrapping data & function into single unit.
* Inheritance: Properties of one class can be inherited into & others.
* Polymorphism: ability to take more than one forms.
* Dynamic Binding - code which will execute is not known until the program runs.
* Message Passing: object. message (Informal) call format.

Advantages.

* Better code reusability using objects & Inheritance.
* Principle of data hiding helps build secure systems.
* Multiple objects can co-exists without any interference.
* Software complexity can be easily managed.

↳ extension of structures in c.

Classes & Objects.

Attributes & Methods are basically variables & functions that belongs to the class. These are often referred to as "class member".

Syntax:-

```
class Employee {
    private:  // Access specifiers
        int a, b; // Attribute.
    public:
        int c, d;
};
```

Syntax.

```
class Employee // class defination.
{
  private: // Access specifier (private)
    int a, b, c; // Attributes (Private)
  public : // Access specifier (public).
    int d, e; // Public Attributes.
    void setdata(int a1, int b1, int c1); // declaration
    void getdata() { // definations of class Methods
      cout<< a <<b<<c<<d<<e;
    }
};

                    Scope resolution operator
                          ↓
void Employee :: setdata(int a1, int b1, int c1){
    a = a1
    b = b1
    c = c1
} // defining a class fxn out side the class.

int main(){
  Employee Atharva; // defining object.
  // Atharva.a = 134; This will throw error as 'a' is Private and can't be used directly.
  Atharva.d // using class Attributes on object
  Atharva.e
  Atharva.setdata(1, 2, 3);
  Atharva.getdata(); // using class methods on object.
  return 0;
}
```

if class declaration ~~is~~ along with object declaration.

class definition
class Employee{
    // definations.
} Atharva, Kaplay;
object definition.

Although it is valid, its not recommended.

Nesting of Member functions.

* member functions of a classes can be called inside of other functions of the same class without '.' operator.

Objects Memory Allocation ~~A using Aror~~

A common memory is allocated for member functions & methods and ~~spee~~ seperate memory is allocated for ~~all~~ attributes of the ~~eac~~ class as per requirement for all objects.

Arrays with classes:

Static Data Members

Syntax :-

```
class demo{
  static int counter;
};

int demo :: counter;
```

Note :- Static variables are automaticly initiallized by zero.

* Value of a static variable defined in a class is same for all objects. *

Static functions.

```
class demo{
 public:
  static int function(){ // defining a
      //code inside function  static fxn
  }
};

int main(){

  demo::function(); // Calling a static fxn
}
```

* Static fxn is not related to object, its related to class and remains same for all objects of that class.

* Static fxn can only access static of other 'static' fxn or 'static' variables of the class and cannot access other than it.

Array of objects.

Syntax:-

```
class Employee {
  int Id;
  int salary;
  public:
  void SetId (void){
      salary = 122;
      cout << "Enter the id of employee" << endl;
      cin >> id;
  }
  void GetId (void) {
      cout << "Employee Id:" << Id << "Salary:" << salary
  }
};

int main() {
    Employee emp[100];
    for (int i = 0; i < 100; i++) {
        emp[i].SetId();
        emp[i].GetId();
    }
    return 0;
}
```

We can make arrays ~~of~~ of objects of a class and use them just like we use array with other ^ datatypes
Primary.

Passing Objects as Function Arguments in C++

We can pass objects as function Arguments in C++ just like other datatypes. for that we have to declare the class at the time of fxn declaration.

```
class complex {
    int a, b;
  public:
    void setData(int v1, int v2) {
        a = v1;
        b = v2;
    }
    void setDatabysum(complex o1, complex o2) {
        a = o1.a + o2.a;
        b = o1.b + o2.b;
    }
    void Printdata(void) {
        cout << a << "+" << b << "i" << endl; }
};

int main() {
    complex c1, c2, c3;
    c1.setData(1, 2);
    c1.Printdata();
    c2.setData(3, 4);
    c2.Printdata();
    c3.setDatabysum(c1, c2);
    c3.Printdata();
    return 0; }
```

If this fxn is defined outside the class, it will throw error as a, b are Private.

output:-

```
1+2i
3+4i
4+6i
```

Friend Function.

Friend function is any external function which can access private attributes or methods of a class. It has to be declared inside the class using "friend" keyword.

Syntax:

```
class complex{
  int a, b
  public:
    void setNumber(int n1, int n2){
      a = n1
      b = n2
    }
    friend Complex sumComplex(complex, complex);
    void printNumber(){
      cout << a << "+" << b << "i" << endl;
    }
};
Complex sumComplex(Complex o1, complex o2){
    Complex o3;
    o3.setNumber((o1.a + o2.a), (o1.b + o2.b));
    return o3;
}
int main(){
  Complex a, b, c;
  a.setNumber(1, 2);
  b.setNumber(3, 4);
  c = sumComplex(a, b);
  return 0;
}
```

class declaration using "friend" in it keyword.

Calling a normal function and always another txn

Properties of friend fxn.

1. Not in the scope of class.
2. Since its not a member fxn of the class, it can't be called by object of the class
3. Can be invoked without the help of any object.
4. Usually contains the objects as arguments.
5. Can be declared in Private or Public in the class.
6. It can't access the members directly by their names and need obj_name_member_name to access any member.

Friend (class functions)

Member fxn of a class can be declared friend fxn of another class

Syntax:

```
class demo{

    friend void demo1 :: fxn1 (void);

};

class demo1.

    void fxn1 (void) {
        //code.
    }
};
```

scope resolution op.

Friend Classes.

a class can be declared as a Friend in another class using "friend" keyword so that the class fxns can access the private members of the another class

Syntex

```
class demo1;
class demo{

    friend class demo1

};

class demo1{

    //code
};
```

all methods of demo1 can access the private members of demo.

Constructors 1 of 29

* its a special method that is automatically called when an object of a class is created. It is initialize object of a class.
* It has no ~~datatype~~ return datatype (not even void) and It's name is same as class's name
* It should be declared in public section of the class
* Syntax:- * We can't refer to their Address.

```
class demo{
  int a, b
  public:
    demo(int x, int y){ // Constructor defination
      a = x
      b = y
    }
};
int main(){
demo object1(1, 2);
```

Automatically initialized values of a, b for object 1.

~~Properties~~

~~OOPs Parameterized~~

Default Constructor.

Constructors which take no Parameter

```
class demo{
  public:
    demo(void){
      //code.
    }
};
```

Paramaterized Constructor

Constructor which take Parameters is called Paramaterized Constructor.

Syntax:-

```
class demo{
  public:
    demo(int a, int b){
      //code
    }
};
int main(){
  demo a(1, 2); //implicit call

  demo b = demo(5, 7); //explicit call

  return 0;
}
```

lec - 31

Constructor Overloading

defining & using more than 1 constructor in a single class. (rules are same as function overloading)

Syntax:

```
class demo{
  Public:
  demo(){                    // default constructor
    // code
  }
  demo(int x){
    // code
  }
  demo(int x, int y){
    // code
  }
  demo(char x){
    // code
  }
};
```

Parametrized constructor

Constructors overloaded

Constructors With default Arguments

```
class demo{
  Public:
  demo(int a, int b=0){
    // code.
  }
};
```

Alternative Way of Calling a Constructor & declearing or defining a object-

```
class demo {
    Public:
        demo( int a);
            // code.
        }
};

int main() {
    demo object1 (2); // standerd way

    demo object2 = demo(2); // Alter way
    return
};
```

Copy Constructor.

A copy constructor is a type of constructor that creates a copy of another object, we can use a copy constructor. If no copy constructor is written in the program, the compiler will supply its own copy constructor

```
class class_name {
int a;
public:
    class_name ( class_name &obj) {
        a = obj.a
    }
};
```

Destructor

* Used to dump any object and free the allocated space for that object
* It never takes an argument nor does it return any value.
* It is called implicitly by compiler.

Destructor Syntax:-

```
class demo{
    public:
        demo(){                    // constructor
            // constructor
        }
        ~demo(){                   // destructor
            // destructor
        }
};
```

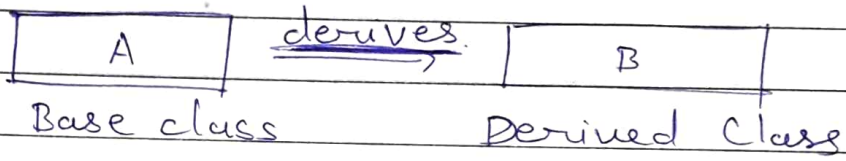
Inheritance In C++

* Used to reuse the code saving lot of time.
* A class properties can be inherit to another class and other properties also can be made.
* We can reuse the properties of an existing class by inherititing from it
* The existing class is called Base class
* The new class which is inherited is called as the derived class

Forms of Inheritance.

- ★ Single Inheritance.
- ★ Multiple Inheritance.
- ★ Hierarchial Inheritance.
- ★ Multilevel Inheritance.
- ★ Hybrid Inheritance.

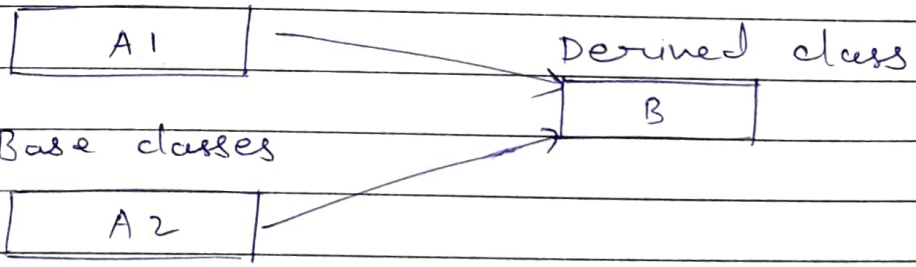
Single Inheritance.



★ only one Base & Derived class

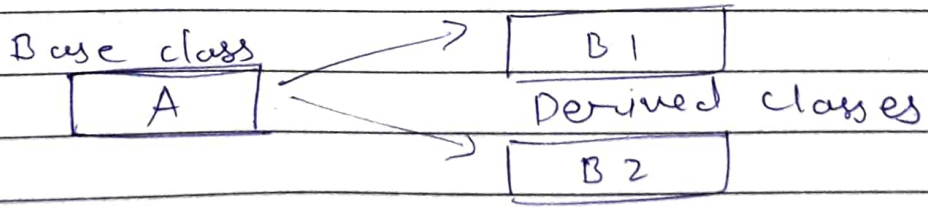
Multiple Inheritance.

★ one derived class with multiple Base classes.

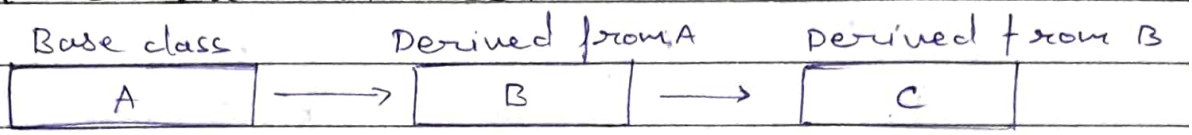


Hierarchial Inheritance.

★ Multiple derived class with one Base class



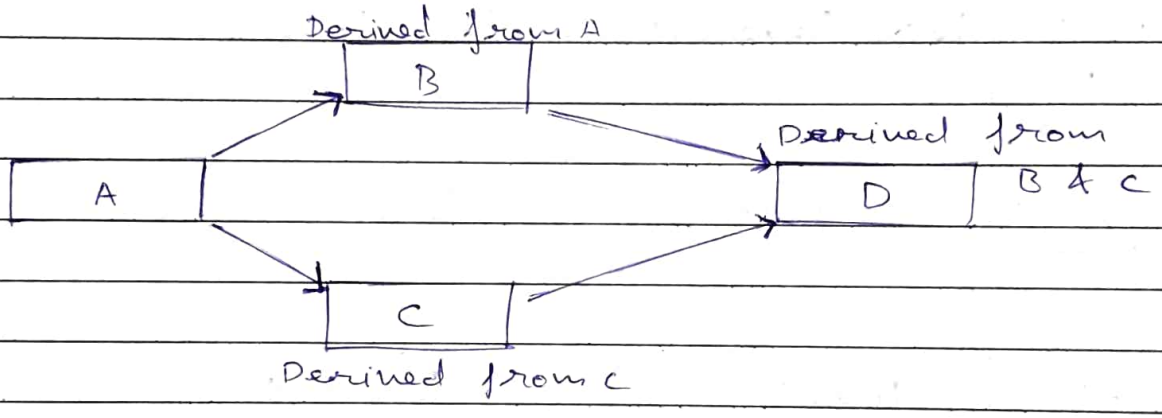
Multilevel Inheritance.



Deriving a class from already derived class

Hybrid Inheritance.

* It combination of Multiple, Hierarchical & Multilevel Inheritance.



Inheritance Syntax

let → 37

```
class <derived_class_name> : <visibility_mode> <base_class>
{
  // code for derived class
}
```

Visibility mode.

Public

all Public members of base class will b'com Public members of derived class.

Private.

All Public Member of base class will b'come Private members of derived Class.

Note1: default visiblity mode is Private.

Note2: Private members of Base class can never be inherited.

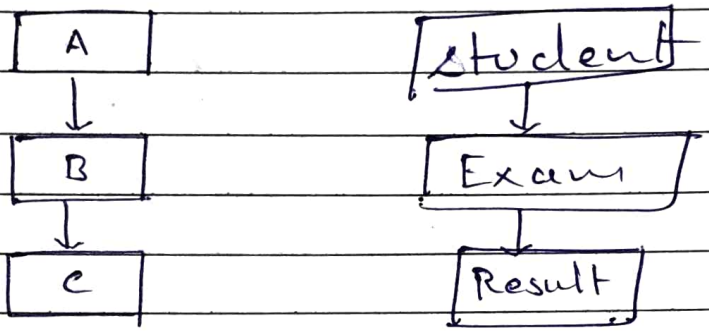
Visiblility M. → / Base cl. mem ↓	Public Derivat^n	Private Derivat^n	Protected Derivat^n
Private members	Not Inherited.	Not Inherited	Not Inherited
Protected members	Protected.	Private	Protected
Public members	Public	Private	Protected.

Access of ~~Derived~~ inherited members of Base class to derived class.

Protected Access Modifier 101 39

* Members inside the Protected Section can't be called outside the class or its derived class but can be inherited in other classes.

Multilevel Inheritance Deep Dive.



~~class student~~ {
~~int roll;~~
~~string name;~~
~~int pho~~
long phone_no;
~~public:~~
void getdata()

A → B → C
Inheritance Path

① A is Base class for B & B is Base class for C
② A → B → C is called Its Inheritance Path

Multiple Inheritance

Syntax:-

~~class derived : visibility mode Base1, Visibility Mode Base2~~

class derived: visibility_mode Base1 , visibility_mod Base2{

//code

}

Exercise

Q) Create 2 class.

1.) Simple Calculator:-
Takes input of 2 no.s using a utility fxn & performs +, -, *, / & displays the results using another function.

2.) Scientific Calculator.
Takes input of 2 no.s using a utility fxn & performs any 4 scientific operations of your choice & displays the results.

Create another class Hybrid Calculator & inherit it usin these 2 classes.

Ambiguity Resolution in C++

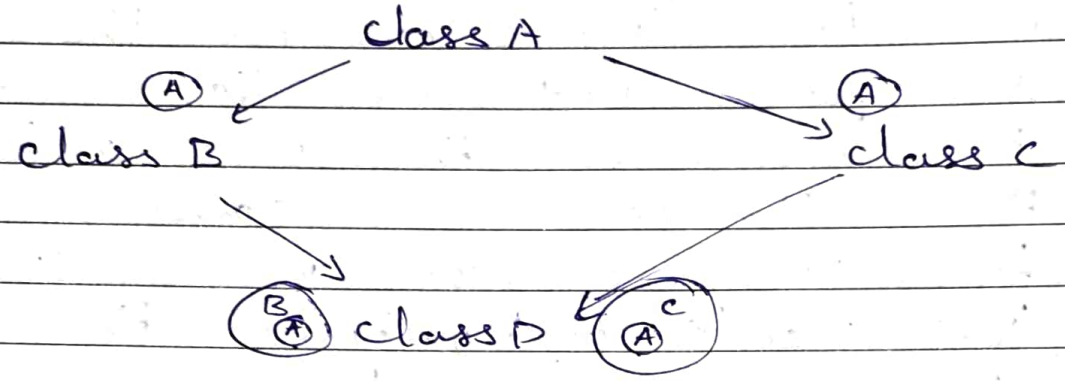
If any ambiguity occours at the time of inheritance for example there is a same function with some different statements in 2 base classes then which will be executed through the derived class (multiplly inherited from the 2 base classes) when it is called. This is called ambiguity and this is to be solved seperately.

Syntax:

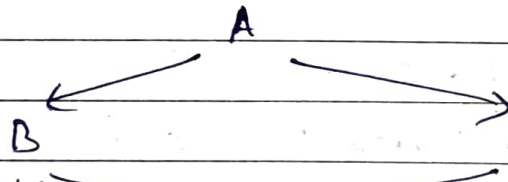
```
class Base1 {
  public:
    void greet() { // greet in Base1
        cout<<"How are You?"<<endl;
    }
};
class Base2 {
  public:
    void greet() { // greet in Base2
        cout<<"Kaise ho?"<<endl;
    }
};
class Derived: public Base1, public Base2 {
  public:
    void greet() {
        Base2::greet(); // Ambiguity is Resolved by declearing which of the Ambiguous fxns is to be used.
    }
};
```

Ambiguity as same fxn is defined in Both Base classes.

Virtual Base Class.



* Consider ~~of~~ classes A, B, C, D where classes B & C are inherited from A & D is inherited from B & C then there is ambiguity b/w the Properties of A from B & Properties of A from C in D. Then which should it follow
* To solve this ambiguity ~~default~~ class A can be declared 'virtual' during inheritance.




```
class B : virtual public A{
    // code
};
```

```
class C : virtual public A{
    // code
};
```

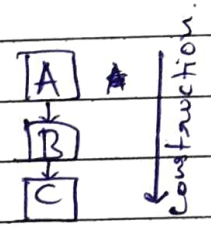
Constructors in Derived Class.

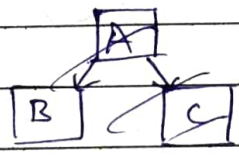
* If base class constructor doesn't have any arguments, there is no need of any Constructor in derived class.
* If base class has arguments, derived class need to pass of arguments to the base class Constructor.

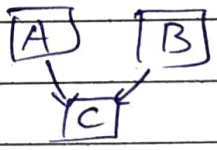
* If both base & derived classes have Constructors, base class ~~[illegible]~~ Constructor will execute first.

Constructors in Multiple & Multilevel Inheritance

* In multiple inheritance, base classes are constructed in order in which they appear in the class declaration.
* In multilevel inheritance, the constructor are executed in the order of inheritance.



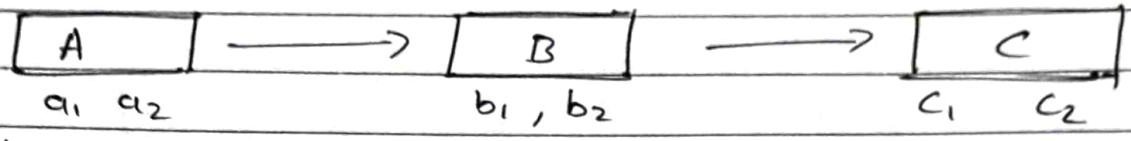




Construction

class c : public A, public B

Example




```
class A{
  A(a1,a2){
    // code.
  }
};
```

```
class B{
  B(b1,b2){
    // code.
  }
};
```

Same

```
class A{
  int A1, A2;
  public:
    A(int a1, int a2){
      A1 = a1;
      A2 = a2;
    }
};
```

```
class B{
  int B1, B2;
  public
    B(int b1, int b2){
      B1 = b1;
      B2 = b2;
    }
};
```

```
class c: public A, public B{
  int c1, c2;
  public
  C(int a1, int a2, int b1, int b2, int c1, int c2) :
      A(a1, a2), B(b1, b2), C(c1, c2){
        // code.
      }
};
```

Special Case of Virtual Base Class

- The Constructor for Virtual base classes are invoked b4 an non-virtual base class.
- If there are multiple virtual base classes, they are invoked in the order declared.
- Any non-virtual base class are then constructed b4 the derived class cont constructor is executed.

Initialization list in Constructors in Cpp.

Syntax

```
Constructor (argument-list) : initilization-section{

  assignment + other code;
}
```

Example :- (for Inheritance)

```
class test{
  int a, b;
  public:
    test(int x, int y){
      a = x;
      b = y;
    }
};
class demo{
  int c, d;
  public
    demo(int P, int q){
      c = P
      d = q } };
```

```
class derived : public test, public demo {
    int e, t;
public:
    derived(int p, int q, int r, int s, int t, int u) :
Initialization list → test(p, q), demo(r, s) {
        e = t;
        t = u;
    }
};
```

Example for a simple class.

```
class test {
    int a, b;
public:
    test(int i, int j) : a(i), b(j) {
        // code for constructor
    }
};
```

a is declared 1st so it should initialized // a = i & b = j

Initialization list.

Note.

* the varible which is declared 1st, it should be initialized 1st. if not done so it will not be initialized and will have garbage value.

New & delete Keyword.

Using Pointers With Classes & objects.

Dynamic Memory Allocation:-

new Keyword.

It is used to dynamically initialize and use a pointer.

Syntax:-

```
int *p = new int(40);
    or
float *p = new float(40.7);
    or
char *p = new char("A");
    or
    etc,
    or
int *arr = new int[3];
 arr[0] = 10;
 arr[1] = 10;
 arr[2] = 100;
```

delete Keyword.

it is used to dynamically free the allocated space.

Syntax:-

```
delete p;   // to free a single block

delete arr[]; // to free continuous memory blocks.
```

Using Pointer with objects.

```
class test{
    int a, b;
    public:
        void setdata (x, y){
            a=x.
            b=y.
        }
};

int main(){

test object;
test *ptr = &object;
(*ptr).setdata (2,3);
```

We can use pointers with objects of any class just like we use it with other standard datatypes.

Arrow Operator (->)

```
ptr -> setdata (2, 3);
```

This is same as above statement

int *ptr = new int [34];

The statement is asking Compiler to dynamically allocate 34 continuous integer memory blocks pointed by "ptr", i.e., "ptr" is pointing the 1st memory block of that 34 memory blocks.

this Pointer

* Every object in c++ has access to its own address through an important pointer called "this" pointer.
* Inside a member fxn, "this" may be used to refer to the invoking object
* its an implicit parameter of all member function.

```
class demo {
   // code
};
int main() {
   demo A; // for A ('this' points to A)
   demo B; // for B ('this' points to B).
```

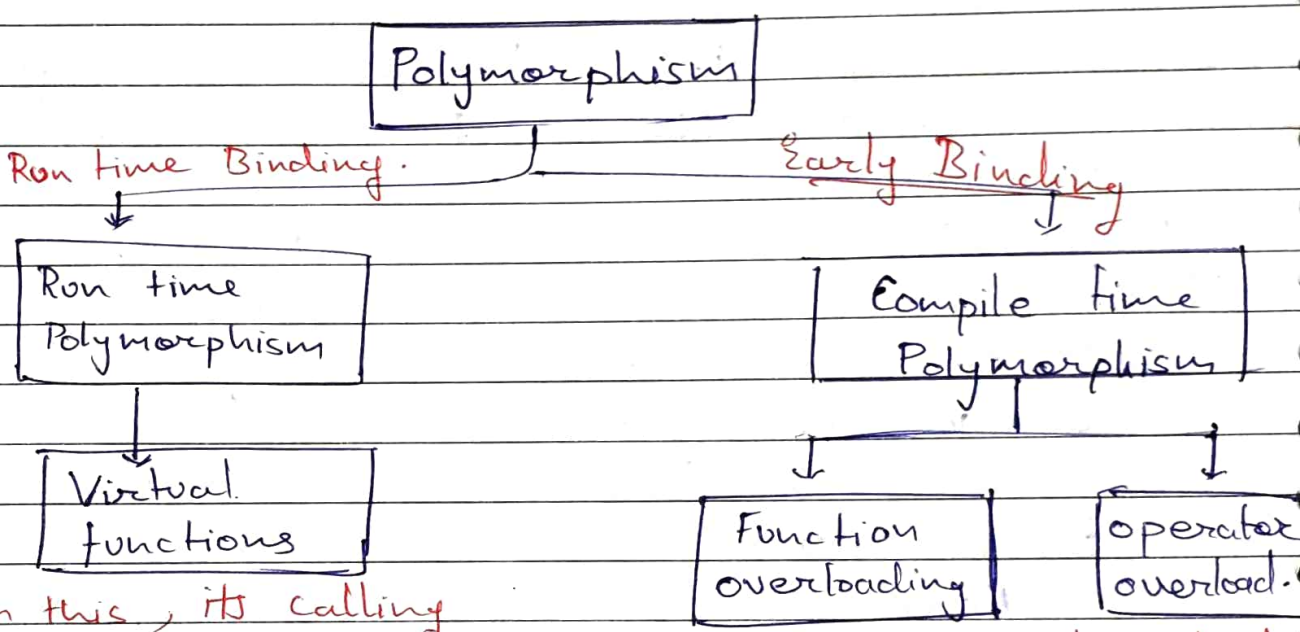
let -> 54

Polymorphyism in C++

(One name and multiple forms)

Example :-

1) Function overloading
2) operator overloading
3) Constructor overloading
4) Virtual functions.



In this, its calling A order of calling of the function is not decided by Compiler at compiling & linking, insted its decided at the time of execution. (i.e., Run time)

In this, its decided that which function or operator is calling and in which order at the time of Compiling & linking.

Pointers to derived Class.

- A Pointer of any base class can point the object of derived Class in C++
- If any base class pointer is pointing any object of derived Class, the pointer can only access the members of base class which are inherited in the derived class.
- A Pointer of any derived ~~cat~~ class Can NOT point the ~~of~~ object of derived Class in C++.

Example:-

```
class base {
  public:
    base_var;
void display() { cout<<"base:" << base_var ; }
};
class derived : public base {
  public:
    derived_var;
    void display() {
      cout << "base: "<< base_var<<endl
           <<"derived: "<< derived_var<<endl;
    }
int main() {
base* base_ptr;
derived derived_obj;
base_ptr = &derived_obj;
base_ptr-> derived_var = 10; //Valid
// base_ptr -> base_var = 10; // Valid.
base_ptr -> display(); // it will call display() of base class
```

multiple display() definations so it will be decided @ Runtime that will display() is called.

LATE Binding (Binding will done @ Runtime)

Virtual functions

~~class base{~~
~~int a = 1;~~
~~public: void~~
~~virtual display();~~

```
class base {
    int a = 1;
public:
    virtual void display() {
        cout << " base: " << a;
    }
};
class derived {
    int b = 2;
public
    void display() {
        cout << "base: " << a << endl
             << "derived: " << b << endl;
    }
};
int main() {
base* base_ptr;
derived derived_obj;
base_ptr = &derived_obj;
base_ptr -> display();
return 0;
}
```

it can't be called, it can only be inherited.

There are multiple display() definations but base::display() is vertual. ∴ derived::display() is called.

(Runtime Polymorphysim)
Late Binding

Rules for virtual functions.

1) They ~~are~~ cannot be static
2) They are accessed by object pointers
3) Virtual functions can be a friend of another class
4) a virtual function in base class might not be used.
5) If a virtual function is defined in a base class, there is no necessity of redefining it in the derived class.

Abstract base Class & Pure Virtual functions

virtual void function() = 0; // do-nothing fxn
// its a pure virtual function.
// there must be a redefination of this fxn in every derived class

Abstract base class:-

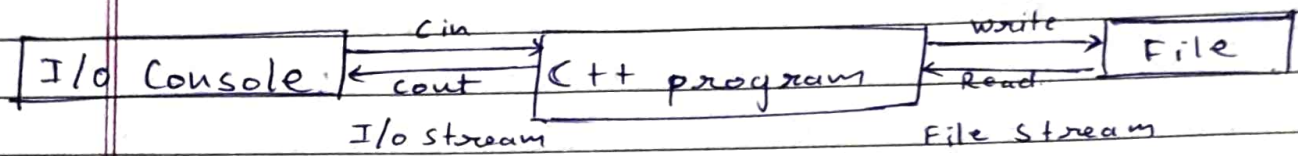
A base class containing at least one pure virtual function is called Abstract base class.

An Abstract class can't be used directly, it can only be inherited that's why its called Abstract 'base' class.

→ but pointers of this class can be made, if you want base classptr to point derived objects.

→ However you can't create objects of it.

File I/O in C++



library for File I/O

~~#include <fstream.h>~~
#include <fstream>
#include <ifstream>
~~#include <oftst~~

#include <fstream>

class.

<fstream>
1) fstream
2) ofstream
3) ifstream

} derived from fstream.

Write operation C using ofstream Constructor

```
ofstream out ("Sample.txt");
out << "Hello";
        or
string s = "Hello";
~~out << "Hello";~~
out << s;
```

Reading operation (using ifstream constructor)

```
string s;
ifstream in("sample.txt");
in >> s; // It will stream only 1st word
         (till 1st whitespace) to s.

getline(in, str2); // It will stream 1st line
                   in to s.
```

.close(); is used to close output or input file stream.

Using .open() function to read or write

Writing to file.

```
ofstream fout;
fout.open("sample.txt");
fout << "Hello, this is Atharva" << endl;
fout.close(); // after closeing, another
              file can be opened.
```

Reading from file.

```
string s;
instream fin;
fin.open("sample.txt");
fin >> s;
cout << s;
fin.close(); // after closeing, another file
             can be opened.
```

.eof() function.

.eof() can be used to find the end of file.

.eof() is a bool fxn which returns 1 if file is ended and 0 if file is not ended.

Templates.

Why Templates :-

1. DRY (Don't Repeate Yourself)
2. Generic Programming.

Syntax for templates.

```
template <class T>
class vector{
        T* arr;
        Public:
            vector (T* arr) {
                // code.
            }
        // other code.
};
int main(){
 vector <int> myVec (Ptr);
 vector <float> myfVec (Ptr);
}
```