* What is Complexity ?

Ans ← Complexity refers to how complicated a program's design, structure or code it. Complexity means how hard or easy it is to understand, modify, and maintain a piece of software.

Lower complexity means code that is straightforward & simple to work with, while higher complexity often implies code this is hard to manage.

* Difference between Array & Linked list ?

Array	Linked list
• An array stores elements in a contiguous memory location.	• Linked lists store elements randomly at any address in the memory.
• In an array, memory size is fixed while declaration.	• Linked list utilize dynamic memory

Array	Linked list
• Memory size cannot be altered at run time.	• Memory size can be altered at run time.
• An array is a data structure that stores elements of the same data type.	• Linked lists consist of nodes where each node contains a data field & a reference to the next node in the list
• Arrays are index-based data structures where each data element is associated with an index	• Each node consists of two fields: one field stores data, & the 2nd field (pointer), stores an address that keeps a reference to the next node.

* Why Linked list is called Non-homogenous?

Sol:- In linked list, various types of data can be stored. For this reason, linked list is called non-homogenous.

* Why Array is called homogenous?

Sol:- In Array, one type of data can be stored. For this reason, ~~time~~ Array is called homogenous.

* Difference between Call by value & Call by reference?

Call by value	Call by reference
• In the case of calling a function, we pass the values of variables directly to a function. Thus, it has it's name as call by value.	• Whenever calling a function, rather than passing the variable's values, we pass it's address to the function. Thus, it has it's name as call by reference
• It copies the ~~=~~ value of a passed Parameter into the function's argument. Thus, any changes that occur in any argument inside the function have no reflection on the Passed Parameter.	• Both the passed Parameter & the argument refer to a similar location. Tus, any changes occuring in any argument inside the function also reflects in the Passed Parameter.

• The method of call by value passes a copy of the variable.	• The method of call by reference passes the variable itself.
• Languages like C++, C#, etc provide support to the call by value.	• Only the JAVA language supports the call by reference method.
• In the call by value method, there is no modification in the original value.	• In the call by reference method, there is a modification in the original value.

* difference between Referencing & Dereferencing?

Referencing	Dereferencing
• Referencing means taking the address of an existing variable (using &) to set a pointer variable.	• Dereferencing means using the * operator to retrieve the value from the memory address that is pointed by the pointer.

• '&' is the reference operator – it gives you a reference (pointer) to some object.	• '*' is the dereference operator – it takes a reference (pointer) & gives you back the referred to object.
• Example → int i = 5; int *p; p = &i; [&i returns the memory address of the variable i]	• Example → int i = 5; int *p; p = &i; *p = 7 [*p returns the variable now i is 7
• Referencing means making a pointer point to a variable	• Dereferencing is accessing the value of the variable that pointer points to.

*Difference between Linear & Non-linear DS

Linear DS	Non-linear DS
• A type of data structure that arranges the data items in an orderly manner where the elements are attached adjacently.	• A type of data structure that arranges data in sorted order, creating a relationship among the data elements.
• Memory utilization is inefficient.	• Memory utilization is efficient.
• Single level	• Multi level
• Easier to implement	• Difficult to implement
• Ex → Array, Linked list, queue, stack	• Ex → tree, graph

* Difference between Singly & Doubly Linked list

Singly L.L	Doubly L.L
• SLL is a set of nodes where each node contains 2 field – 'data' & next 'link'	• DLL node contains 3 fields – data, Previous link & a next link
• In SLL, the traversal can be done using the next node link only. ~~Thus~~, traversal is possible in one direction only.	• In DLL, the traversal can be done using the previous node link or the next node link. So, traversal is possible in both direction.
• The SLL occupies less memory than DLL as it has only 2 fields.	• The DLL occupies more memory than SLL as it has 3 fields
• We mostly prefer to use SLL for the execution of stacks.	• We can use a DLL to execute stacks, binary trees.
• SLL is ~~as~~ less efficient.	• DLL is more efficient.

Computational Complexity

Input size

- Number of elements in the input.
- Size of an array

Running time

- The number of Primitive operations (steps) executed before termination.

Fact → Running time grows with the size of the input

order (rate) of growth

- The leading term of the formula.

Computational Complexity

(i) Space Complexity

(ii) Time Complexity

Space Complexity

- It is the amount of memory which is needed by the Program to run to completion.
- We can measure the space by finding out that how much memory will be consumed by the instructions & the variables ~~used~~ used.

✓ [Code লিখার সময় যত variable ~~& space~~ use করার যায় তা দ্বারা space পরিমাপ করতে পারি] variable কমত কম হবে space কম হ

Example

Suppose we want to add 2 ~~integer~~ integer numbers.

Algorithm 1

step 1 → input A
step 2 → input B
step 3 → set C = A+B
step 4 → 'Sum is', C
step 5 → Exit

Algorithm 2

step 1 → input A
step 2 → input B
step 3 → 'Sum is', A+B
step 4 → Exit

✓ Both Algorithm Produce same result. But first takes 6 bytes and second takes 4 byte (2 bytes for each integer). first has more instruction than the 2nd one. So, we'll choose the 2nd one as it takes less space than the first one.

Time Complexity

- It is the amount of time which is needed by the algorithm (program) to run to completion.
- We can measure the time by finding out the compilation time and run time.
  - The compilation time is taken by the compiler to compile the program. It depends on the compiler & differs from compiler to compiler. So we ignore the compilation time & only consider run time
  - The run time is taken to execute the program. We can measure the run time on the basis of number of instructions in the program.

Example :-

Algorithm 1

step 1 → input A
step 2 → input B
step 3 → C = A+B
step 4 → 'SUM is', C
step 5 → Exit

Algorithm 2

step 1 → input A
step 2 → input B
step 3 → 'sum is', A+B
step 4 → Exit

Suppose second is required to execute one instruction. So, 1st will take 4 sec & 2nd one will take 3 sec for execution. So, we will choose the second one as it will take less time.

Algorithm Complexity

I) Worst Case Complexity :-

The function defined by the maximum number of steps taken on any instance of size n.

[কোড রান করার পর যদি desired output শেষের দিকে / Last step-এ পাই তাহলে এতে পুরো কোড-এর সকল step compile হয় এবং এতে যে কোড বড় সময় বুঝায়]

II) Best Case Complexity :-

The function defined by the minimum number of steps taken on any instance of size n

[কোড রান করার পর যদি desired output প্রথমেই / first step-এ পাই তাহলে। এতে পুরো code এর সকল step compile করার প্রয়োজন হয় না]

Average Case Complexity

the function defined by the average number of steps taken on any instance of size n.

[কোড-টুকরা-প্রোগ্রাম পাই যদি desired output তাড়াতাড়ি চলে পাই তাহলে]

- Best case depends on the input
- Average case is difficult to compute
- So we use Worst case Analysis
  - Easier to compute
  - Usually close to the actual running time.
    [মোট Last step পর্যন্ত যায়]

Asymptotic Notation

3 types →

1) Big O (oh) Notation → Used in Worst case
2) Big Ω (Omega) Notation → " " Best case
3) Big Θ (Theta) Notation → " " Average case

Finding Time Complexity :

* Algorithms → 1

```
a = 4;                  ⟶ O(1)      Simple line (executed only 1)
b = a + z;              ⟶ O(1)      "
for i = 1 to n;         ⟶ O(n+1)    executed (n+1) time
    SUM = a + b;        ⟶ O(n)      executed n time
```

Number of operations required by this loop are →

$(1+1+n+1+n)$

$= 2n+3$ ⟶ constant num. 2 & 3 का फिर

$= O(n)$

* Algorithms → 2

```
for i = 1 to n;         ⟶ O(n)
    for j = 1 to n;     ⟶ O(n)      [As it's Nested loop
        SUM = a + b;
```

Number of ~~of~~ operations required by this loop →

$O(n) \times O(n)$

$= O(n^2)$

Strategy–1

Increment / Decrement loop → $O(n)$

Example–

```
for (i=0; i<n; i++);      → O(n)
for (i=0; i<n; i+=10);    → O(n/10) = O(n)
for (i=n; i<>n; i--);     → O(n)
```

Note ← '+' কিংবা '–' এর ক্ষেত্রে loop-এ $O(n)$ হবে।

Strategy–2

Multiply / Divide loop → $O(\log_K n)$

Example–

```
for (i=1; i<n; i*=2)      → O(log2 n)
for (i=1; i<n; i*=3)      → O(log3 n)
for (i=n; i>1; i/=4)      → O(log4 n)
```

Strategy – 3

Double loop ⟶ $O(n^2)$

যেহেতু এক লুপের মধ্যে আরেক লুপ আছে তাই ১ম টি সূত্র হিসেবে হবে $O(n)$, ২য় টি সূত্র হিসেবে হবে $O(n) \times O(n) = O(n^2)$, ৩য় টি হবে $O(n^2) \times O(n) = O(n^3)$। আগেরটির সাথে $O(n)$ গুণ হয়ে যাবে।

Example:-

```
for (i=0 ; i<n; i++)        ⟶ O(n)
    for(j=0 ; j<n; j++)     ⟶ O(n²) → O(n)×O(n)
        for (m=0 ; m<n ; m++) ⟶ O(n³) → O(n²)×O(n)
```

$O(n + n^2 + n^3) = O(n^3)$

↳ এখানের সবচেয়ে বড় value টি

Strategy – 4

Loop condition based on external value ⟶ $O(\sqrt{n})$

Example

```
P = 0                       ⟶ O(1)
for (i=1 ; P<=n ; i++)      ⟶ O(√n)
    statements...
    P = P+1
```

Like as, যেহেতু P external value (P=0) এর উপর loop depend করছে তাই-

```
for (i=1, i*i <=n; i++)        ⟶ O(√n)
    statements...
```

যখনই loop এর মধ্যে (i*1) হয়েছে

Strategy → 5

```
p=0
for (i=1; i<n; i*=2)           ⟶ O(log n)
    p++;
for (j=1; j<p; j*=2)           ⟶ O(log p)
    statements...
```

As, p = O(log n)

Complexity = O(log p)

= O(log log n)

like as,

```
for (i=1; i<n; i++)            ⟶ O(n)
    for (j=n; j=1; j/=2)       ⟶ O(log n)
        statements...
```

Complexity = O(n log n)

Example

```
for (i=1 ; i<n ; i++){          ⟶ O(n)
    for (j=1 ; j<m ; j++){      ⟶ O(m)  O(mn)
    }
}
```

O(m) × O(n) = O(mn)

Complexity: O(mn + n)

```
for (i=1 ; i<n ; i++){          ⟶ O(n)
}
```

Complexity = O(mn + n + n)

= O(2n + mn)

= O(n + mn) or, O(mn)

Rate of growth

Let n, be the size of input.

* Lower Bound (Best Case)

$1 < \log n < \sqrt{n} < n$

* Average case (Exact Bound)

n

* Upper Bound (Worst case)

$n < n\log n < n^2 < n^3$

Exercise

```
int n=100, x=3, i, j;          ⟶ O(1)
int Arr[n]                     ⟶ O(1)
for(i=0; i<n; i++)             ⟶ O(n)
    Arr[i] = x;
    x++;
for(j=i-1; j>0; j/=2)          ⟶ O(log2n)
    Printf("Div Lo = %d", Arr[j];
    i = Arr[j];
    While(i*i<=x)              ⟶ O(√x × log2n)
        Printf("%d", Arr[i]);
        i++;
    Printf("\n");
```

Complexity = $O(1+1+n+\log_2 n+\sqrt{x}\log_2 n)$

$= O(2+n+\log_2 n+\sqrt{x}\log_2 n)$

$= O(n+\log n+\sqrt{x}\log n)$ ↘ n এর Rate of Growth বেশি

$= O(n)$

Array

→ Set of data

→ Homogeneous

→ fixed size

C- Pointer

→ A Pointer is a variable that stores the memory address of another variable.

→ Some C Programming tasks are performed more easily with Pointers, & other tasks, such as dynamic memory allocation, cannot be performed without using Pointers.

→ Every variable is a memory location and every memory location has it's address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

Example

```
int main ()
{  int var1;
   char var2[10];
   Printf ("Address of var1 is %x\n", &var1);
   Printf ("Address of var2 is %x\n", &var2);
}
```

Address Hexa-decimal-এ আসে তাই %x use করা হয়েছে

Address ছোট করার জন্য

Output :- Address of var1 is bff5a400

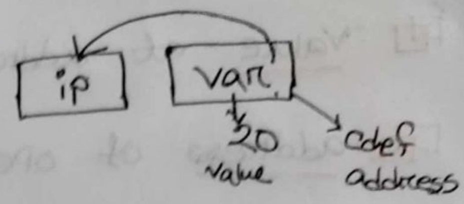
" " var2 is bff5a3f6

Declaration of Pointer

```
data-type * Var_name ;
int *p
char *p
double *p
float *p
```

Example

```c
int main ()
{
    int var = 20;
    int *ip;          → Pointer variable declare
    ip = &var;        → Pointer variable এর মধ্যে 'var' variable এর address রাখা হয়েছে
    Printf("Address of var variable :%x\n", &var);   → 'var' variable address
    Printf("Address of ip variable : %x\n", &ip);    ↗ ip variable এর নিজের address (&ip)
    Printf("Address store in ip variable: %x\n", ip);  ↗ ip variable এ যে address store. অর্থাৎ 'var' variable এর address
    Printf(" Value of ip variable: %d\n", *ip);    ↘ ip variable এ যে address store আছে সেই variable এর value (*ip)
}
```



এখন এ var declare করেছি, যেহেতু ip এর মধ্যে var variable এর address ও রাখা হয়েছে তাই এর address ও value ip তে চলে আসছে

Output :-

```
Address of var variable : cdef
   "    of ip      "    : aefg
   "    store in ip "   : cdef
Value of  ip       "    : 20
```

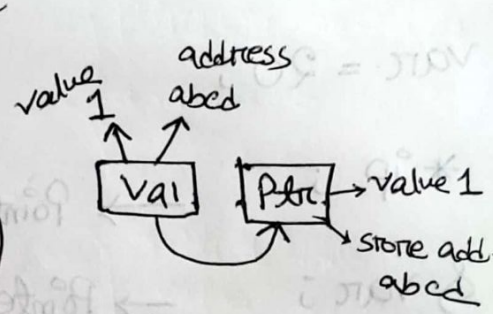
Note :- Pointer declare করার সময় variable এর নামের পূর্বে * দিতে হয়। নিজের address বের করার জন্য & sign use করা হয় (&var). নিজের value বের করার জন্য * use করা হয় (*var). variable এ অন্য variable এর store করা address জানার জন্য (var) use করা হয়। & sign লাগে না।

Value at address (*)

address of operators (&)

Example

```
{
    int *Ptr;      → Pointer declare
    int Val = 1;
    Ptr = &Val;
    int deref = *Ptr;
    Printf ("%d\n", deref);
}
```



Ptr variable এর মধ্যে 'Val' variable এর address রাখা হয়েছে।

deref = Ptr এর ~~value~~ ~~val~~ মধ্যে থাকা value(*Ptr)

Output :- 1

Ptr এর নির্দেশ ~~val~~ address কে বুঝাতে হলে (&Ptr), Ptr এর মধ্যে থাকা address কে বুঝাতে হলে (Ptr)

NULL Pointer

→ A Pointer that is assigned NULL is called a NULL Pointer.

→ The NULL Pointer is a constant with a value of zero defined in several standard libraries.

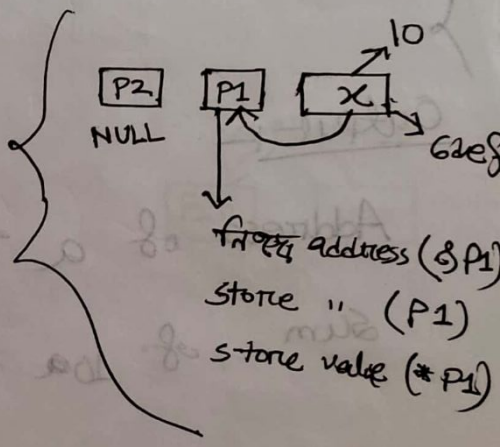
Example :-

```
#include<stdio.h>
void main()
{
    int *Ptr = NULL;
    Printf("The value of Ptr is: %x\n", Ptr);
}
```

Output :- The value of Ptr is 0

Example :-

```
int x = 10;
int *P1, *P2 = NULL;
P1 = &x;
Printf("Address of x : %x\n", &x);
Printf("Address of P1 : %x\n", P1);
Printf("Address of P2 : %x\n", P2);
```

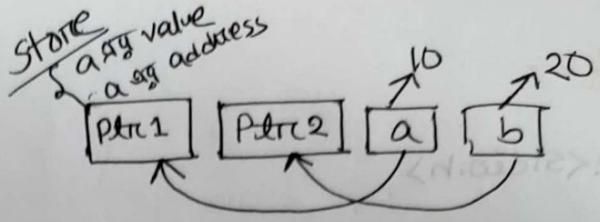


Output :-

Address of x : 6def
" of P1 : 6def
" of P2 : 0

As *P2 is a NULL Pointer.

Example




```
int main()
{
    int *Ptrc1, *Ptrc2, a, b;
    int a = 10, b = 20;
    Ptrc1 = &a;
    Ptrc2 = &b;
    Printf("Address of a is %x & b is %x \n", Ptrc1, Ptrc2);
    Printf(" Sum of %d & %d is %d \n", a, b, *Ptrc1 + *Ptrc2);
}
```

Ptrc1 এর মধ্যে a এর address & Ptrc2 এর মধ্যে b এর address

*Ptrc1 দ্বারা a এর value. *Ptrc2 দ্বারা b এর value

Output :-

Address of a is 61fe0c & b is 61fe08

Sum of 10 & 20 is 30

C-Pointer Arithmetic

→ C-Pointer is an address, which is a numeric value

→ 4 arithmetic operators → ++, --, +, -

Example

Let, Ptrc is an integer pointer which address is 1000

then, ++ Ptrc.

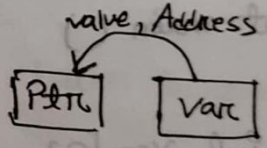
Now, Ptr will Point to the location 1004.
Because each time Ptr is incremented, it'll Point next integer location which is 4 bytes.

* If Ptr is a character whose address is 1000. then above operation will Point to the location 1001 because next character will be available at 1001

যেহেতু Ptr একটি integer we know, integer memory তে 4 byte দখল করে। যখন Ptr 1 বাড়বে তখন next memory তে 4 byte দখল করবে। তাই 1000 থেকে 1004 হবে।

যেহেতু character memory তে 1 byte দখল করে তাই Ptr 1 বাড়লে 1000 থেকে 1001 আসে

Increment Pointer




```
#include <stdio.h>
int main()
{
    int var[3]={10,20,30};   → Array declare
    int i, *Ptr;             → Pointer declare
    Ptr = &var;              → Ptr এর মধ্যে 'var' store করা
    for(i=0; i<3; i++){
        Printf("Address of var[%d]= %x\n", i, Ptr);   → Ptr এর মধ্যে 'var' এর address
        Printf("Value of var[%d]= %d\n", i, *Ptr);    → Ptr এর মধ্যে থাকা value
        Ptr ++;
    }
```

→ Address বাড়বে. যেহেতু Ptr integer type তাই 4টি memory দখল করবে। Address 4 করে বাড়বে।

Output :-

Address of var[0] = 61fe04 → Address 4 করে বাড়বে কারণ integer type.

value of var[0] = 10

Address of var[1] = 61fe08

value of var[1] = 20

Address of var[2] = 61fe0c →12

Value of var[2] = 30

প্রথমে i-এর condition check করে loop-এ ঢুকবে পরে loop এর কাজ শেষ করে Ptr এর address বাড়বে পরে i এর value ও বাড়বে আবার condition check করে loop-এ ঢুকবে

Decrement Pointer

```
int main ()
{
   int var[3] = {10,20,30};
   int i, *Ptr;
   Ptr = & var[3-1];
   for (i=3; i>0; i--){
      Printf("Address of var[%d]=%x\n", i, Ptr);
      Printf("Value of var[%d]=%d\n", i, *Ptr);
      Ptr--;
}
```

Ptr = & var[3-1]; → Ptr এর মধ্যে var store করা

প্রথমে i এর value 3, আর 3>0 তাই loop এ ঢুকবে ও address print করবে। পরে Ptr-- মানে address 4 byte কমে যাবে যেহেতু int type. পরে আবার loop এ এসে i-- মানে i এর value হবে loop চলবে যেহেতু ৩টি index (0,1,2) তাই var[3-1] লিখছি

Ptr-- ; → Ptr এর মধ্যে থাকা address 4 byte কমে যাবে।

Output :-

Address of var[3] = 61fe0c

value of var[3] = 30

Address of value[2] = 61fe08

value of var[2] = 20

Address of var[1] = 61fe04

value of var[1] = 10

Pointer * AND ++

- *p++ or, (*p)++ → *p before increment, later increment *p
- *++p or, *(++p) → increment p first, *p after increment
- ++*p or, ++(*p) → increment *p first, after increment *p

↓

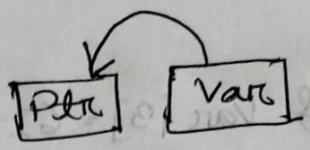
Pointer than Pointer (*p) increment

Pointer Comparison

⇒ Pointers Compared by using relational operators (==, <, >)

Example :-

```
int main()
{
    int var[3]={10,20,30};
    int i, *ptr;
    ptr = &var;
    i = 0;
    while(ptr<=&var[3-1]){
        Printf("Add. of var[%d] is %x\n",i,ptr);
        Printf("Value of var[%d] is %x\n",i,*ptr);
        ptr++;
        i++;
    }
}
```



Ptr দ্বারা যদি 'var' variable address 3 value store করা। যেহেতু Ptr এ যদি var store করা আছে, 0 index এ আমরা loop এ প্রবেশ করছি। তাহলে কত address বাড়বে (ptr++) 4 byte বাড়বে। এবং i এর value 0 থেকে 1 এ আসবে same Ptr এর Value.

Output → Add :- --
Value - - [same output increment]

Array of Pointers

```
int main()
{
    int var[3]={10,100,200};
    int i;
    for(i=0;i<3;i++){
        Printf("value of var[%d] = %d\n",i,var[i]);
    }
}
```

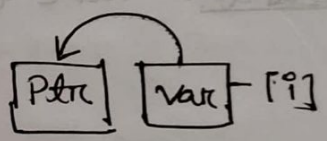
i এর value 0→2 পর্যন্ত যাবে এবং loop check করে loop-এ প্রবেশ করে loop এর কাজ শেষ করবে,

Output:- Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

int *Ptr[ ]

This declares Ptr as an array of int pointers. Thus, each elements in Ptr, now holds a pointer to an int value

```
int main()
{
    int var[3] = {10, 100, 200};
    int i, *Ptr[3];
    for (i=0; i<3; i++){
        Ptr[i] = &var[i]
    Printf("Value of var[%d] = %d\n", i, *Ptr[i]);
    Printf("Address of var[%d] = %x\n", i, Ptr[i]);
}
```



i=0 হলে Ptr[0] = &var[0]
0 তম index এ address
Ptr[0] এ গিয়ে store হবে,
Ptr[0] index এ যে address
সেটির value (*Ptr[0])
Print করে

Output:- Value of var[0] = 10 | Address of var[0] = 61fe10
Value of var[1] = 100 | " " var[1] = 61fe14
" of var[2] = 200 | " " var[2] = 61fe18

Character type

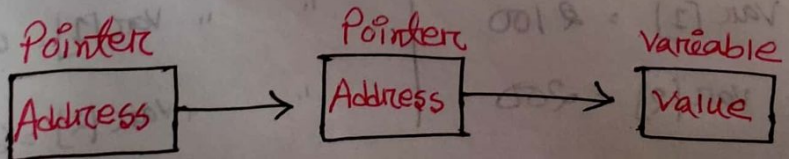
```
int main()
{
    Char * names[] = {"NURA", "PRIMU", "SUBI", "NISHU"};
    int i;
    for (i=0; i<4; i++){
    }   Printf("Value of names[%d] = %s\n",i,names[i]);
}
```

Output :- Value of names[0] = NURA
" [1] = PRIMU
" [2] = SUBI
" [3] = NISHU

Pointer to Pointer

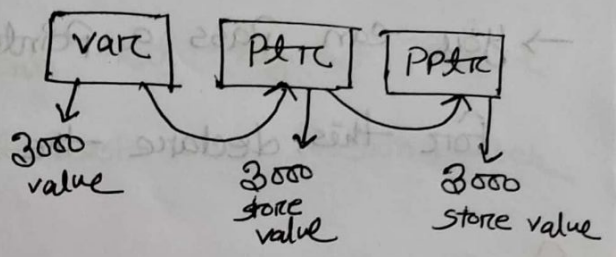
→ A Pointer Contains the address of a variable

→ When we define a Pointer to a Pointer, the first Pointer contains the address of the 2nd Pointer, which Points to the location that contains the actual value



```
int **Var;
```

Example :-



*Ptr → Ptr पर रखी हुई value

**Ptr → *Ptr पर रखी हुई value

```
int main ()
{
    int var;
    int *Ptr;
    int **PPtr;
    var = 3000;
    Ptr = &var;
    PPtr = &Ptr;
    Printf("Value of var : %d\n", var);
    Printf("Value of *Ptr : %d\n", *Ptr);
    Printf("Value of **Ptr : %d\n", **Ptr);
}
```

Output :-

Value of var : 3000

" *Ptr : 3000

" **Ptr : 3000

Passing Pointers to function

→ You can pass a pointer to a function

for this, declare the function parameter as a pointer type.

Example :-

```
#include <stdio.h>

double A (int *arr, int size)   → Return type_ func name _(arguments)
{
    int i, sum=0
    double avg ;
    for(i=0 ; i< size ; i++){
        sum = sum + A[i] ;
    }
    avg = sum/size ;
    return avg ;
}
int main()
{
    int valence[5]={10,20,30,40,50} ;
    double avg ;
    avg = A (valence, 5);
    Printf(" Average is : %lf\n ", avg);
}
```

0 < size = 5

sum = 0 + A[0]
= 0 + 10 = 10

sum = 10 + A[1]
= 10 + 20 = 30

sum = 30 + A[2] = 30 + 30
= 960

sum = 60 + A[3] = 60 + 40
= 100

sum = 100 + A[4] = 100 + 50
= 150

then 5 < size 5 ;
loop finish

Output :-

Average is 30.000

C-Function

→ A function is a group of statements that together perform a task. Every C Program has at least one function, which is main().

→ A function declaration tells the compiler about a function's name, return type, and Parameters.

```
return type_function name (parameters)
{
    statements ;
}
```

→ A function definition Provides the actual body of the function.

→ A function is known with various name like a method or a procedure etc.

Uses

→ Functions are used to break up large programs into named sections.

→ A function already been using, which is main function.

→ Functions are often used when the same piece of code has to run multiple time.

→ In this case, you can put this piece of code in a function & give that function a name. When the piece of code is required you just have to call by it's name.

→ If you want to use function you just have to call the function.

Example:-

```
#include <stdio.h>
void MyPrint()          ⟶ function এর মধ্যে statement লিখব
{
    printf("Printing from a function");
}
void main()
{
    MyPrint();          ⟶ function কে call করা হচ্ছে
}
```

output:- Printing from a function

A function definition in C language consist of

1) function header
2) function body

Parts of a ~~function~~ function

→ **Function name** :- actual name of function. The function name & the parameter list together constitute the function signature.

→ **Parameters** :- A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. Parameter are optional, a function may contain no parameters.

→ **Return type** :- A function may return a value. The return type is the data type of the value the function returns. If you don't want to return a result from a function, you can use void return type. In this case, the return type is the keyword void.

→ **Function body** → The function body contains a collection of statements that define what the function does.

Example

```
int max (int num1, int num2)
{
    int result;
    if (num > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

argument

Function Declarations

- A function declaration tells the compiler about a function name & how to call the function.
- A function declaration has the following parts:

```
return type function_name (Arguments/parameters);
int max (int num1, int num2);
```

- Parameters names are not important. Only their type is required. So, this is also valid →

```
int max (int, int);
```

- Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a function

- To use a function, you will have to call that function to perform the defined task.
- When a program calls a function, program control is transferred to the called function. A called function performs defined task & when it's return statement is executed, it returns program control back to the main program
- To call a function, you need a to pass the required parameters along with function name, and if function returns a value, then you can store returned value

Example :-

```
int square (int n1)
{
    int x = n1 * n1 ;
    return (x) ;
}

Void main ()
{
    int num ;
    num = square(4) ;
    Printf (" %d ", num) ;
}
```

ধাপ-1 - main function এখান শুরু করবে। main function এ num variable square(4) call করবে। এর square function এর n1 এর মান যাবে। পরে n1 গুণ করবে। $x = n_1 \times n_1 = 4 \times 4 = 16$ function 16 return করবে main function এর num এ, পরে num Print হবে।

Output :- 16

Example :- Calling a function →

```
#include <stdio.h>
int max (int num1 , int num2)
{
    int result ;
    if (num1 > num2)
        result = num1 ;
```

→ max function (argument)

→ function এর body

```
    else
        result = num2;
    return result;
}
void main()
{
    int a = 10,
    int b = 20;
    int total = max(a,b);
    Printf ("Max is : %d\n", total);
}
```

→ result value return करके main function में आएगा

→ main function

→ function को call करके हमारे a यानि num1, b यानि num2 result value total में आएगा

→ total value print हो जाएगी।

* Function Arguments

- C functions can accept unlimited Number of Parameters.
- If a function is to use arguments, it must declare variables that accept the values of the arguments. These are called formal Parameters.
- Parameter →

  Actual Parameter :- This is the argument which is used in function call.

  Formal Parameter :- This is the argument which is used in function definition.

Variables →

Local variable :-

- A local variable is a variable that is declared inside a function.
- A local variable can only be used in the function where it is declared

Global variable :-

- A global variable is a variable that is declared outside all functions.
- A global variable can be used in all functions.

Example

```
#include <stdio.h>
int A;
int B;
int Add()
{
    return A+B
}
int main()
{ A=5;
  B=5;
  int answer;
```

Here,
A, B → Global variable
These variables can be used in main() and Add() in two function.

Here,
answer → local variable
it can only be used in main()

```
answer = Add()
 printf("%d\n", answer);
}
```

Output :- 10

Global variable হচ্ছে —
ফাংশনের বাইরে থাকে —

* Type of function call

While calling a function there're two ways that arguments can be passed to a function.

1) **Call by value** → Copies the actual value of an argument into the formal parameter of the function. Changes have no effect on the argument.

ii) **Call by reference** → Copies the address of an argument into the formal parameter. Changes made to the parameter affect the argument.

Call by value

- In this method, different memory is allocated for both actual and formal parameters. Because value of actual parameter is copied to formal parameter.

• C Programming language uses call by value method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

Example :-

```c
#include <stdio.h>
Void value (int x)
{
    x = 5;
}
int main ()
{
    int z = 27;
    value (z);
    Printf ("z is now %d\n", z);
}
```

Output :- 27

z = 27
value (z) মানে value function কে call করে z এর মান x receive করে তখন x = 27 হয়।
x এর মান value main function এ চলে আসলে। তখন z = 27

[x] [z] → 27

z = 27 হলো এই value টি x receive করে এবং x = 27. এরপর value ফিরে call করেছে এই func এর মধ্যে x এর value change হওয়া এর z এর মধ্যে কোনো change আসে না।

Example → swap() function

call by value

```
#include <stdio.h>
void swap (int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
void main ()
{
    int a = 100;
    int b = 200;
    Printf("before swap: a is %d\n", a);
    Printf("before swap: b is %d\n", b);
    Swap (a,b);
    Printf("After swap: a is %d\n", a);
    Printf("After swap: b is %d\n", b);
}
```

→ temp এর মধ্যে x এর value রাখলাম

→ x এর মধ্যে y এর value

→ y এর মধ্যে temp এর value

local variable

value পাঠানো call করার [swap(a,b)] এর swap function এর মধ্যকার x এর value change a এর b তে কোনো প্রভাব না, এর a এর value swap হয় নাই।

Output:- Before swap: a is 100
" : b is 200
After swap: a is ~~200~~ 100
" : b is ~~100~~ 200

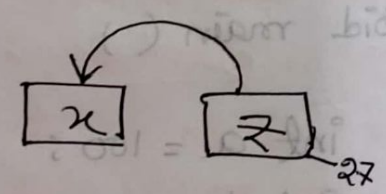
Output shows that there's no change in the values cause it changed inside the function.

Call by Reference

- Inside the function, the address is used to access the actual argument used in the call. That's mean, changes made to the parameter affect the passed argument.
- To pass the value by call by reference, argument pointers are passed to the functions.

Example :-

```
#include <stdio.h>
void value (int *x)
{
    *x = 5 ;
}
int main ()
{
    int z = 27 ;
    value (&z) ;
    printf ("z is now %d\n", z);
    return 0 ;
}
```



z এর address x receive করেছে। এই এখানে *x মানে z। value function এর মধ্যে *x = 5, এই z এর মান change হয়ে 5 হবে এবং main function এ আসবে

Output :- z is now 5

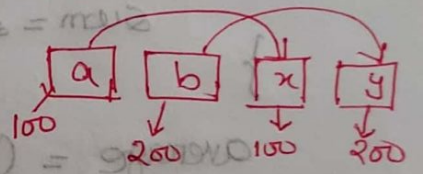
swap() function

Call by Reference

```
#include <stdio.h>
void swap (int *x, int *y)
{ int temp;
  temp = x;
  x = y;
  y = temp;
}
void main()
{
  int a = 100;
  int b = 200;
  Printf("before swap: a is %d\n", a);
  Printf("before swap: b is %d\n", b);
  swap(&a, &b);
  Printf("After swap: a is %d\n", a);
  Printf("After swap: b is %d\n", b);
}
```

→ function call

swap(&a, &b), means a variable ও address x pointer receive করবে, b variable ও address y pointer receive করবে।



then function এর কাজ শুরু হবে।

Output :- Before swap: a is 100
" : b is 200
After swap: a is 200
" : b is 100

Passing an array to a function

```
#include <stdio.h>
int average (int age[])
{
    int i;
    int average, sum=0;
    for (i=0; i<6 ; i++){
        sum = sum+ age [i];
    }
    average = (sum/6);
    return average;
}
int main ()
{
    int age[] ={20,25,30,35,40,15};
    int avg;
    avg = average(age);
    Printf("Sum is %d\n", avg);
}
```

[returntype_function name (array type - array name)]

যখন i=0, i<6 তার loop এ যাবে;

sum = 0 + age[0]
= 0 + 20 = 20

then i = 1,

sum = 20 + age[1]
= 20 + 25 = 45

→ main function এ average value return হবে।

→ array initialize

→ function call
avarge function এ যাবে
age[] → array এর element গুলো যাবে।

Output →

Sum is 27

Passing two dimensional array to function

```
#include<stdio.h>
void display(num[2][2])
{
    int i, j;  Printf(" Displaying:\n");
    for(i=0; i<2 ; i++){
        for(j=0; j<2 ; j++){
            Printf("%d ", num[i][j]);
        }
        Printf("\n");
    }
}
int main()
{
    int i,j;
    int num[2][2] ;
    for(i=0; i<2; i++){
        for(j=0; j<2; j++){
            Printf("num[%d][%d]", i, j);
            scanf("%d", &num[i][j];
        }
    }
    display(num);
}
```

output :-

```
num[0][0] = 1
num[0][1] = 2
num[1][0] = 3
num[1][1] = 4
Displaying:
1 2
3 4
```

Structures

- Structure is the collection of variables of different types under a single name for better handling.
- In C programming, structures allows you to combine data items of different kinds.
- Structures are used to represent a record, suppose you want to keep track of your books in a library.

Defining a structure

- To define a structure, you must use struct statement, The struct statement defines a new data type, with more than one member of your program.

```
struct [structure tag]
{
    member definition;
    member definition;
} [one or more structure variables];
```

- The structure tag is optional and each member definition is a normal variable definition, such as int i; float f

Example → 1

```
struct Books          ← structure tag
{
    char title[50];   ┐
    char author[50];  ├ member definition
    char subject[100];┘
    int id;
} book;
```

book ← structure variables

Example → 2

```
struct person
{
    char name[50];
    int id;
    float salary;
};
```

With the declaration of the structure you have created a new type, called Books.

Structure variable declaration

Ex → 1

```
struct person
{
    char name[50];
    int id;
    float salary;
};

struct Person P1, P2, P[20];   ← Inside main function
```

Ex → 2

```
struct Person
{
    char name[50];
    int id;
    float salary;
} P1, P2, P[20];
```

In both cases, 2 variables P1, P2 and array P having 20 elements of type struct Person are created.

Difference between C variable, Array & structure

- A normal C variable can hold one data of one data type at a time.

Ex → int a = 20
char b = 'Z'

Here, a & b are variable

- An array can hold group of data of same data type.

Ex → int a[3];
a[0] = 10;
a[1] = 20
a[2] = 30
char b[10] = "Hello"

Here, a[3], b[10] → array

- A structure can hold group of data of different data types.

Ex →
```
struct student
{
    int a;
    char b[10];
}

a = 10;
b = "Hello";
```

- Data types can be int, float, double, ~~etc~~ char etc.

Concept

Using Normal Variable

```
struct student
{
    int marks;
    float average;
};

int main ()
{
    struct student report = {100, 80.765};
    Printf(" marks = %d\n", report.marks);
    Printf ("Average = %.2f\n", report.average);
}
```

structure tag

syntax

→ initialize structure

Accessing structure members

[As normal variable এর dot (.) ব্যবহার করা]

output ← marks = 100

Average = 80.76

```
struct student
{
    int marks;
    char name[50];
    float average;
};
int main()
{
    struct student report;
    report.marks = 80;
    strcpy(report.name, "Nura");
    report.average = 75.478;
    Printf("mark = %d\n", report.marks);
    Printf("Name = %s\n", report.name);
    Printf("Average = %.2f\n", report.average);
}
```

→ initialize declare structure

→ যারই string আই strcpy use করুন হয়ে। কারণ array তে সরাসরি string নেয়া যায় না

Output:

mark = 80

Name = Nura

Average = 75.47

* Accessing members of a structure

* Structure can be accessed in 2 ways. They are
  1) Using Normal structure variable
  2) Using Pointer variable

* There're two types of operators
  1) Member operator (.) dot
  2) structure pointer operator (->) Arrow

* Dot (.) operator is used to access the data using normal structure variable.

* Arrow (->) is used to Access the data using Pointer variable.

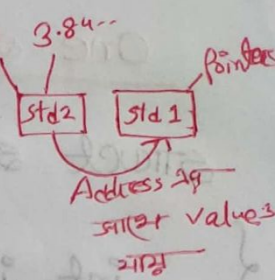
Pointers to structures

```
struct student
{
    int age;
    float cgpa;
};

int main ()
```

```
{
    struct student *std1;
    struct student std2 = {30, 3.8495};
    std1 = &std2;
    Printf ("Age = %d\n", std1->age);
    Printf ("cgpa = %.2f\n", std1->cgpa);
}
```

→ struct Pointer



Output :- age = 30

cgpa = 3.84

One student data

```
struct student
{
    int id;
    char name[50];
    float cgpa;
};
struct student std1;
int main()
{
    struct student std1;
    std1.id = 1;
    strcpy(std1.name, "Nura");
    std1.cgpa = 3.8493;
    Printf("Id is = %d\n", std1.id);
    Printf("Name is = %s\n", std1.name);
    Printf("cgpa is = %.2f\n", std1.cgpa);
}
```

Output:-

```
Id is = 1
Name is = Nura
Cgpa is = 3.84
```

3 student data

```
struct student
{
    int id ;
    char name [50] ;
    float cgpa ;
}
int main()
{
    struct student record [3] ;
    int i ;
    record [0].id = 1 ;
    Record [
    strcpy (record [0].name = "Nura");
    record [0].cgpa = 3.8495 ;
    record [1].id = 2 ;
    strcpy (record [1].name = "Prima");
    record [1].cgpa = 3.509 ;
    record [2].id = 3 ;
    strcpy (record [2].name = "Subi");
    record [2].cgpa = 3.487 ;
    for(i=0; i<3 ; i++) {
        Printf(" Record of student : %d \n", i+1) ;
```

→ record [0],
record [1],
record [2],

```
Printf("ID is : %d\n", record[i].id);
Printf("Name is : %s\n", record[i].name;
Printf("Cgpa is : %.2f\n", record[i].cgpa);
}}
```

Output :-

Record of student : 1
ID is : 1
Name is : Nura
Cgpa is : 3.84
Record of student : 2
ID is : 2
Name is : Prima
Cgpa is : 3.50
Record of student : 3
ID is : 3
Name is : Subi
Cgpa is : 3.48

Input structure data

```
#include<stdio.h>
Struct student
{
    int id;
    char name [50];
    float cgpa;
};
int main ()
{
    int i;
    struct student record [5];
    for (i=0; i<5 ; i++) {
        Printf ("Enter records of %d\n",i+1);
        Printf ("Enter id: ");
        Scanf ("%d", & record [i].id);
        Printf (" Enter name: ");
        fflush (stdin);
        gets (record [i].name);
        Printf("Enter cgpa:");
        scanf ("%f", record [i].cgpa);
}
```

```
for (i=0; i<5; i++){
    Printf ("Record of student : %d\n", i+1);
    Printf ("id = %d\n", record[i].id);
    Printf ("Name=%s\n", record[i].name);
    Printf ("cgpa = %.2f\n", record[i].cgpa);
}
```

output :- ⊗ 5 student data enter

Add two distance by user

```
#include <stdio.h>
struct distance
{
    int feet;
    float inch;
}d1, d2, sum;
int main()
{
    Printf ("1st distance :\n");
    Printf ("Enter feet :");
    scanf ("%d", &d1.feet);
    Printf ("Enter inch :");
    scanf ("%f", d1.inch);
```

```
Printf("2nd distance :\n");
Printf ("Enter feet : ");
scanf ("%d", &d2.feet);
Printf ("Enter inch : ");
scanf ("%f", &d2.inch);
sum.feet = d1.feet + d2.feet;
sum.inch = d1.inch + d2.inch;
if (sum.inch >12) {
    sum.feet++;
    sum.inch = sum.inch-12;
}
Printf("Sum of distance = %d + %.1f ", sum.feet, sum.inch);
}
```

Output :-

```
1st distance :
Enter feet : 5
"  inch : 13.4
2nd distance :
Enter feet : 4
"  inch : 6.89
Sum of distance = 10 + 8.3
```

Passing structure to function

It can be ~~be~~ done in below 3 ways →

1) Passing structure to a function by value
2) " " " " by address (reference)
3) NO need to Pass a structure – Declare structure variable as global.

By value [user থেকে input]

```
#include <stdio.h>
struct student
{
    char name [50];
    int roll;
};
void display (struct student std1)
{
    Printf ("Name = %s\n", std1.name);
    Printf ("Roll = %d\n", std1.roll);
}
void main ()
{
    struct student info;
    Printf (" Enter name: ");
    scanf ("%s", &info.name);
```

→ main function এর info std1 receive করছে

scanf এর পরিবর্তে লেখা যায়
gets(info.name);
scanf দিলে প্রথম একটি শব্দ Print হয়, gets দিলে Name এর সব word Print হয়

```
Printf(" Enter roll : ");
scanf ("%d", &info.roll);
display (info);          ⟶ display function কে call করা হয়েছে।
}
```

Output :- Enter name : Nura
" Roll : 110

Name = Nura
Roll = 110

By Value [called function]

```
#include <stdio.h>
#include <string.h>
Struct student
{
    int id;
    char name [50];
    float cgpa;
};
Void main ()
{
    Struct student record;
    record.id = 110;
    strcpy (record.name, "NURA");
    record.cgpa = 3.8493;
    display (record);
}
```

```
Void display (struct student *info)
{
    Printf ("id = %d\n", *info.id);
    Printf ("Name = %s\n", *info.name);
    Printf ("cgpa = %.2f\n", *info.cgpa);
}
```

Output:- id = 110
Name = NURA
cgpa = 3.84

By value

```
#include <stdio.h>
#include <string.h>
struct books
{
    char title[50];
    char author[50];
    int id;
};
Void main()
{
    struct books book1, book2;
    strcpy(book1.title, "New");
    strcpy(book1.author, "NURA");
    book1.id = 110;
```

```
    strcpy ("book2.title, "Old");
    strcpy (" book2 *author, "Prima");
    book2. id = 120;
    display (book1);
    display (book2);
}
void display (struct books info)
{
    Printf ("Title = %s\n", info.title);
    Printf (" Author = %s\n", info.author);
    Printf (" Id = %d\n", info.id);
}
```

Output :-

Title = ~~NURA~~ New

Author = NURA

id = 110

Title = Old

Author = Prima

id = 120

DMA

In C Programming,

there are 4 library function under <stdlib.h>

1) malloc ()
2) Calloc () } memory allocate
3) free () → Memory free/remove
4) realloc () → নতুনভাবে memory size realloc/ new size

malloc ()

'malloc' stands for memory allocation. malloc () reserves a single large block of memory with specified size and return a pointer of type void.

[malloc () দিয়ে single block of memory নিতে পারি- যার size specific দেয়া- থাকবে-]

Syntax

Ptr = (cast type*) malloc (byte-size);

Void* malloc (size in byte);

Example

```
int *Ptr ;
Ptr = (int *) malloc (100* sizeof (int)) ;
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first bytes of memory.

Calloc ()

'Calloc' or Contiguous allocation allocates the specified number of blocks of memory of the specified type.

malloc() ও Calloc() দুটোর কাজ একই, memory allocation করা, but দুটোর type different. malloc() একটি Parameter নেয় (কত byte size পরকার সেটি) আর- Calloc() দুটি Parameter নেয় (double, float, int-এর size, আর কতটি element দরকার হবে) (কতটি block বানাতে হবে, each block এর size)

Syntax

```
Ptr = (cast-type *) calloc (no.of blocks, size of each block in bytes);
Void * Calloc (n, block-size) ;
```

Example

```
float *ptr;
ptr = (float*) calloc (25, size of (float));
```

25 elements each of size of float 4 bytes.

free()

- 'free' method in C is used to dynamically de-allocate the memory.
- The Programmer must use free() to release space taken from Heap

Syntax

```
void free (Pointer to heap memory);
```

Example

```
free (ptr);
```

realloc()

- "realloc()" or 're-allocation' changes the size of a block of memory that was previously allocated with malloc() or calloc()
- If the previously allocated memory is insufficient or more

than sufficient. Then you can change memory size using realloc()

Syntax

```
Void *realloc (Void *Ptr, newsize);
```

example ↓

```
Ptr = realloc (Ptr, newsize);
```

Code example of malloc()

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int *Ptr;                                   ——— Pointer declare
    Ptr = (int*) malloc (5 * sizeof (int));
    if (Ptr==NULL){
        Printf ("Memory not allocated");
    }
    else
        Printf ("Memory allocation successful");
}
```

Output ⇒ Memory allocation successful.

malloc এর মাধ্যমে memory allocate. As আমরা 5 টি integer ডাটা রাখতে চাচ্ছি। আমরা যানি, যে, int এর size 4 byte তাই 5 টি int ডাটা রাখতে (5×4) = 20 byte দরকার তাই আমরা malloc এর মাধ্যমে। malloc 20 byte এর address Ptr এর মধ্যে assign করবে। (যেহেতু Ptr হচ্ছে int type তাই type cast ও হবে int type

memory তে address আছে কি না তা check করার জন্য

Example :-

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int *Ptr;
    int n, i;
    Printf("How many elements: ");
    scanf ("%d", &n);
    Ptr = (int*) malloc (n * sizeof(int));
    if (Ptr == NULL){
        Printf ("NO memory allocated");
    }
    int sum=0;
    for(i=0; i<n; i++){
        scanf ("%d", Ptr+i);
        sum = sum+ *(Ptr+i);
    }
    Printf ("Sum is %d", sum);
}
```

Pointer declare

Ptr এর জন্য 'n' টি integer এর জন্য memory বরাদ্দ (bytes)

যদি memory allocate হয়ে না Ptr এ তাহলে এটি আসবে

যদি i=0; 0<3 [n=3]
তাহলে, input নিবে, (Ptr+0) এর
sum = 0 + (Ptr+0) এর value
= 0 + 1 = 1
then i=1; 1<3
sum = 1 + (Ptr+1) এর value
= 1 + 2
= 3
then i=2; 2<3
sum = 3 + (Ptr+2) এর value
= 3 + 3 = 6
sum is 6

Output :-

```
How many elements: 3
1
2
3
sum is 6
```

Code example of Calloc() & free()

```
#include<stdio.h>
#include <stdlib.h>
int main ()
{
    int *Ptr , n ;
    Printf("How many elements);
    Scanf("%d", &n);
    Ptr = (int *) Calloc(n, sizeof(int));
    if (Ptr = NULL) {
        Printf("Memory allocated zero");
    }
    else
    Printf("Memory allocation successfully");
    free(Ptr)
    Printf("Memory freed successfully");
}
```

→ same malloc এর মতো,

Calloc (n, size of (int))

n → কতটি element

size of (int) → 1 byte

5, size of(int) = 20 byte

free (Ptr) → memory free হয়ে যাবে / elements থাকবে না।

Output:

```
How many elements : 5
Memory allocation successfully
Memory freed successfully
```

Code example for realloc()

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int *Ptr1, *Ptr2;
    Ptr1 = (int*) malloc (5 * sizeof(int));
    Ptr2 = (int*) calloc (5, sizeof (int));
    if (Ptr1 == NULL || Ptr2 == NULL){
        Printf (" Memory not allocated.\n");
    }
    else
        Printf("Memory Allocation successful\n");
        Ptr1 = realloc (Ptr1, 7 * sizeof (int));
        Printf ("Memory reallocation successfully\n");
}
```

malloc দ্বারা memory allocate

calloc দ্বারা memory allocate

Ptr1 এ - malloc দিয়ে 5 * 4 byte = 20 byte memory allocate করেছি, এখন realloc দ্বারা memory size change করে 7 * 4 byte = 28 byte memory allocate করেছি.

Output:

Memory allocation successful
Memory reallocation successfully

Example

```c
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int *Ptr, n1, n2 ;
    Printf(" Size of Previous Array:");
    Scanf ("%d", &n1);
    Ptr = (int*) malloc (n1 * sizeof(int));
    for (i=0; i<n1; i++){
        Printf ("%x\n", Ptr+i);
    }
    Printf(" Enter Size of new Array: ");
    Scanf ("%d", &n2);
    Ptr = realloc (Ptr, n2);
    for(i=0; i<n2; i++) {
        Printf ("%x\n", Ptr+i);
    }
}
```

Notes:

- array-এর size (n_1)
- malloc এর মাধ্যমে ($n_1 * 4$ byte) মেমরি ... Ptr এর মাধ্যমে ক্রমানুসারে ... [$n_1 = 2$]
- i=0, 0<2 (Address Print হবে), Ptr+0 then 1<2 Address, Ptr+1
- New Array এর size (n_2)
- Ptr এর মাধ্যমে realloc এর মাধ্যমে new array size নেওয়া হবে। এখন Ptr এর মাধ্যমে আগের memory ($n_1 * 4$ byte) change হয়ে নতুন ($n_2 * 4$ byte) তৈরি হবে। পরে loop এর মাধ্যমে address print হবে

Output:–

```
Size of Previous array:2
161430
161434
Size of new Array: 3
161430
161434  161438
```

1-D Array

Array declaration :-

```
data_type array_name[array_size];
int marks [100];
```

Array :- Collection of variable of same data type

```
int marks [5];  ⟶ Array size 5
```

marks [0], marks [1], marks [2]
marks [3], marks [4]

Array initialize :- ⟶ `int marks [5] = {80, 70, 60, 65, 82};`

Array Print :-

```
Printf ("%d", marks [0]);
for (i=0; i<=4; i++)
{
    Printf ("%d", marks [i]);
}
```

User input :-

```
int mark [5];
scanf ("%d", &mark [0]);
```

```
for (i=0; i<=4 ; i++)
{
    scanf ("%d", &mark [i]);
}
```

Types of Array

1) one dimensional (1-D) Array

Ex- int marks [10]

↙
Value - 1 to 3rd bracket

2) Two dimensional (2-D) Array

Ex- int marks [2] [3]

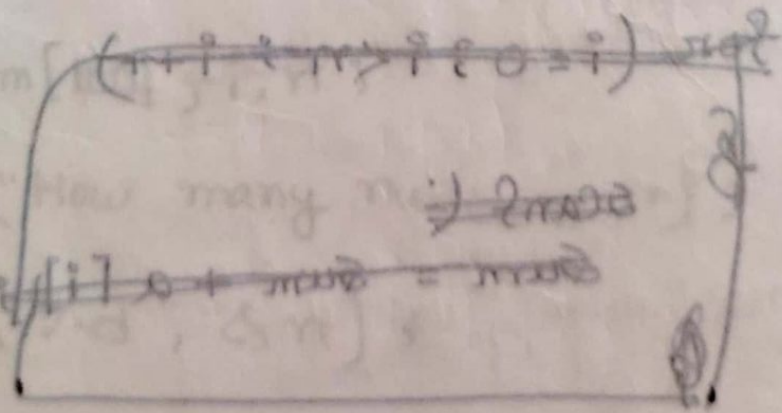
↙
2 bracket.

Array sum ↵

```
#include <stdio.h>
int main()
{
    int digit[5] = {10,20,30,40,50};
    Printf("%d\n", digit[1]);
    Printf("%d\n", digit[3]);
    int sum = digit[0] + digit[1] + digit[2] + digit[4];
    Printf("The sum is %d\n", sum);
    return 0;
}
```

Output ↵

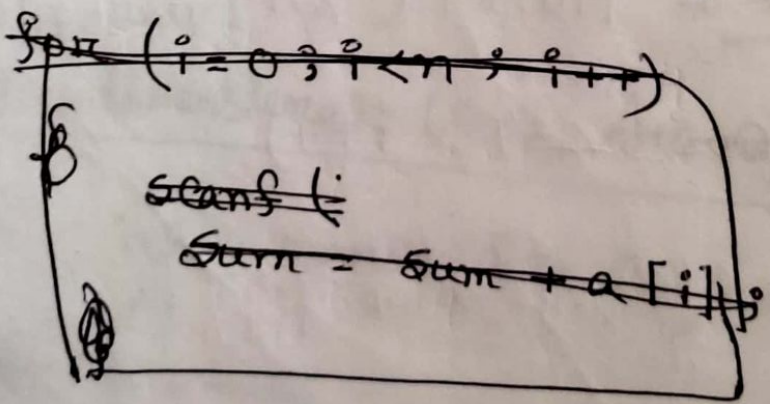
20
40
The sum is 150



Array sum & Average

Getting user input

```
#include <stdio.h>
int main()
{
    int a[100], sum=0, i, n;
    Printf("How many numbers: ");
    scanf("%d", &n);
    for (i=0; i<n; i++)
    {
        scanf("%d", &a[i]);
    }   sum = sum + a[i];
```




```
    Printf("the sum is %d\n", sum);
    Printf("the average is %d\n", sum/n);
    return 0;
}
```

Output:-

how many numbers : 4

1 2 3 4

the sum is 10

the average is 2

Maximum & minimum of Array

// write a Program that can take some numb
and display maximum.

Sol:-

```
#include <stdio.h>
int main ()
{
    int num[100] ; i, n ;
    Printf ("How many numbers : ") ;
    Scanf ("%d", &n) ;
    for (i = 0 ; i < n ; i++)
    {
        scanf("%d", & num [i]);
    }
```

```
int max = num[0]
for (i = 1; i < n; i++)
{
    if (max < num[i])
        max = num[i]
}
Printf("maximum = %d\n");
Printf("maximum = %d\n", max);
return 0;
}
```

Output :-

How many numbers : 4

1 2 3 4

maximum is 4

Minimum of Array

```
#include <stdio.h>
int main()
{
    int digit[100], i, n;
    printf("How many numbers: ");
    scanf("%d", &n);
    for (i=0; i<n; i++)
    {
        scanf("%d", &digit[i]);
    }
    int min = digit[0];
    for (i=1; i<n; i++)
    {
        if (min > digit[i])
            min = digit[i];
    }
    printf("Minimum = %d\n", min);
    return 0;
}
```

Output :- How many numbers : 4

10 20 30 40

minimum = 10

Linear search

* Array থেকে কোন element search করাকে linear search.

```
#include <stdio.h>
int main ()
{
    int num [ ] = {5, 10, 20, 15, 25, 30}
    int value, pos = -1; i ;
    Printf (" Enter the value : ");
    scanf ("%d", & value );
    for (i=0 ; i<6 ; i++){
        if ( value == num[i]){
            pos = i + 1 ;
            break ;
        }
```

```
}
if (pos==-1){
    printf("Not found");
}
else{
    Printf(" the position of the value is :%d\n", pos);
}
return 0;
}
```

Output :- Enter the value : 15
the position of the value is : 4

Enter the value : 23
Not found

2-D Array / Matrix Array

Declaration

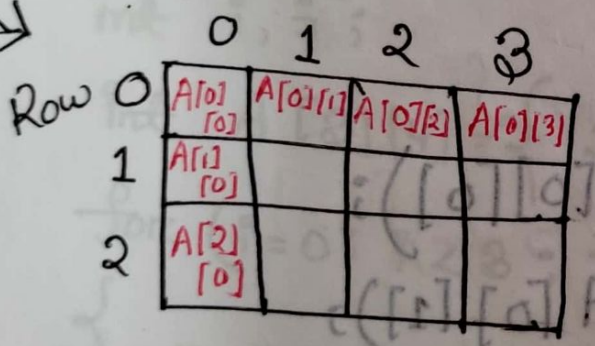
Data_type array_name [row size] [col size] ;

int A [3] [4] ;

↓

Number of elements = row x Columns

= 3 x 4 = 12



Array initialization

- Initializing 1st row →

```
A [0] [0] = 5 ;
A [0] [1] = 6 ;
A [0] [2] = 7 ;
A [0] [3] = 8 ;
```

2nd row →

```
A [1] [0] = 15
A [1] [1] = 16
"
```

3rd row →

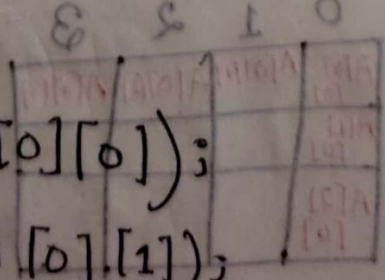
```
A [2] [0] = 25
A [2] [1] = 26
"
```

Or,

```
int A [3] [4] = {
                {5, 6, 7, 8},
                {15, 16, 17, 18},
                {25, 26, 27, 28}
                }
```

Array Printing

```
Printf (" %d", A [0] [0]);
Printf (" %d", A [0] [1]);
"
```



Or,

```
for (i=0; i< no_row ; i++)
{
   for (j=0; j < no_col ; i++)
      {
      Printf ("%d", A[i][j]);
      }
   Printf ("\n");
}
```

2-D Array Print

```
#include <stdio.h>
int main ()
{
   int i, j;
   int A [3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
   for (i=0; i<3; i++)
   {
      for (j=0; j<4; j++){
         Printf ("%d", A [i][j]);
      }
```

```
        Printf ("\n");
    }
    return 0;
}
```

output ←

1 2 3 4
5 6 7 8
9 10 11 12

Getting input for 2D Array

```
#include <stdio.h>
int main ()
{
    int A[3][4], i, j;
    for (i=0; i<3; i++){
        for (j=0; j<4; j++){
            Printf ("A [%d][%d] = ", i, j);
            Scanf ("%d", &A[i][j];
        }
    Printf ("\n");
}
```

input A matrix

```
for (i=0; i<3; i++){
    for (j=0; j<4; j++){
        printf("%d", A[i][j]);
    }
    printf("\n");
}
return 0;
}
```

Output :-

```
A[0][0] = 1
A[0][1] = 2
A[0][2] = 3
 :   :
 :   :
1  2  3  4
5  6  7  8
9  10 11 12
```

Simple matrix

```
#include <stdio.h>
int main ()
{
    int i, j;
    int A[3][4] , int B[3][4];
    Printf (" Element of matrix A : \n ");
    for(i = 0; i<3 ; i++){
        for (j=0; j<4 ; j++){
            Printf ("A [%d][%d] = ", i, j);
            scanf ("%d ", &A[i][j]);
        }
    Printf ("\n");
}
Printf ("A = \n ");
for(i = 0; i <3 ; i++){
    for (j=0; j<4 ; j++){
        Printf ("%d", A[i][j]);
    }
    Printf ("\n");
}
```

//input matrix A

// Print matrix A

```
int B[3][4]
Printf("Elements of matrix B: \n");
for (i=0; i<3; i++){
    for (j=0; j<4; j++){
        Printf("B[%d][%d] = ", i, j);
        scanf("%d", &B[i][j]);
    }
    Printf("\n");
}
Printf("B = \n");
for (i=0; i<3; i++){
    for (j=0; j<4; j++){
        Printf("%d", B[i][j]);
    }
    Printf("\n");
}
return 0;
}
```

// input matrix B

// Print matrix B

Output:- Element of matrix A :

A [0] [0] = 1

A [0] [1] = 2

A =

1	2	3	4
5	6	7	8
9	10	11	12

Element of matrix B :

B [0] [0] = 10

B [0] [1] = 11

B [0] [2] = 12

B =

10	11	12	13
14	15	16	17
18	19	20	21

```
#include <stdio.h>
int main()
{
    int i, j, numberOfRows, numberOfCols;
    int A[10][10], B[10][10], C[10][10];
    printf("Enter number of Rows & Cols : ");
    scanf("%d %d", &numberOfRows, &numberOfCols);
    printf("Elements of matrix A :\n");
    for(i=0; i<numOfRows; i++){
        for(j=0; j<numOfCols; j++){
            printf("A[%d][%d] = ", i, j);
            scanf("%d", &A[i][j]);
        }
        printf("\n");
    }
    printf("A =\n");
    for(i=0; i<numOfRows; i++){
        for(j=0; j<numOfCols; j++){
            printf("%d ", A[i][j]);
        }
        printf("\n");
    }
```

// input elements of matrix A

// Print elements of matrix A

```
Printf("Elements of matrix B:\n");
for(i=0; i<numOfRows; i++){                    //input elements of matrix B
    for(j=0; j<numOfCols; j++){
        Printf("B[%d][%d] = ", i, j);
        Scanf("%d", &B[i][j]);
    }
}
Printf("\n");
}
Printf("B = \n");
for(i=0; i<numOfRows; i++){                    //Print element of matrix B
    for(j=0; j<numOfCols; j++){
        Printf("%d ", B[i][j]);
    }
    Printf("\n");
}
for(i=0; i<numOfRows; i++){                    //Addition of matrix A & B
    for(j=0; j<numOfCols; j++){
        C[i][j] = A[i][j] + B[i][j];
    }
}
Printf("A+B = \n");
```

```
    for(i=0 ; i < numOfRows ; i++){
        for(j=0 ; j < numOfCols ; j++){
            Printf("%d  ", C[i][j]);
        }
        Printf("\n");
    }
    return 0;
}
```

Output :- Enter element of Rows & Cols : 2 2

Elements of matrix A :

A[0][0] = 1

A[0][1] = 2

A[1][0] = 3

A[1][1] = 4

A =

1 2

3 4

Elements of matrix B :

B[0][0] = 5

B =

5 6

7 8

A+B =

6 8

10 12