POINTERS

* C-Language:

How to create statements – 1st unit

Iteration [How many times] – 2nd unit.

collection of similar elements – 3rd unit. [Arrays, strings]

Functions - for Reusable code – 4th unit.

Ex:

```
int sum (int a, int b)
{
    int c;
    c = a+b;
    return c;
}
```

Body of function.

```
int r = sum(5,6);
printf("%d", r);        - 11
```

a = 5, b = 6, c = 11

```
int v1, v2;
int r = sum(v1, v2);
```

} By user (dynamic)

→ All the above parameters – 'call by value'

→ passing address of v_1 & v_2 – 'call by reference'

```
int r = sum(&v1, &v2);
```

* Pointers:

```
f(&a);   ⟶  f(int *p)
```

Address — pointer.

→ 'Pointer' is a 'variable' which stores 'address' of another 'variable'.

Syntax:

```
datatype *v;
```

Here '*' is 'Indirection operator' / 'Dereference operator'

Ex:

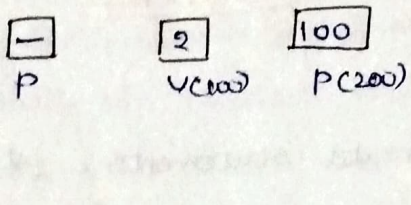
int *p; p is a pointer – stores int datatype Address

float *k; k is a pointer – stores float data Address.

char *c; c is a pointer – stores character Address.

* Ex:-

```
int *p;
int v=2;
p=&v;
```



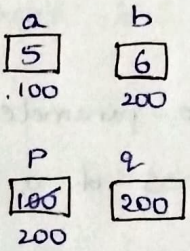

```
pf("%d",v);   -2
pf("%d",*p);  -2.
```

* Note:-

→ Any kind of 'pointer variable' they can allocate same memory size - 'e 8 bytes'

* Ex:-

```
int a=5, b=6;
int *p, *q;
p=&a;
q=&b;
p=q;
pf("%d",*p); →6
pf("%d",*q); →6
```



* Note:-

→ We can perform the following on pointers.

① 'e Pointer Assignment'

② 'e comparison'

③ 'Inc/Decr' [p++, ++p, --p, p--]

④ 'Able to add some constants' [+, −, /]

p+1, p-1, p/1;

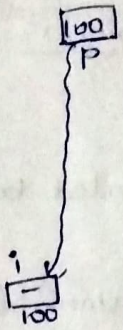
→ Array:- a+1; ⇒ a+1 x size.

pointer:- p+1; ⇒ p+1 x 8 (size)

→ We can't able to perform addition on two pointers (addresses).

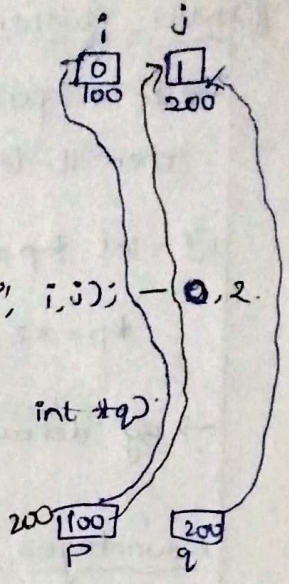
* Ex-1

```
void set (int *p)
{
   *p = 25;
}
main ( )
{
   int i;
   set (&i);
   pf ("%d", i);  -25
}
```



Ex-2:

```
int i=0, j=1;
main ( )
{
  f (&i, &j);
  pf ("%d %d", i, j);
}
void f (int *p, int *q)
{
  p = q;
  *p = 2;
}
```



* Ex-3:

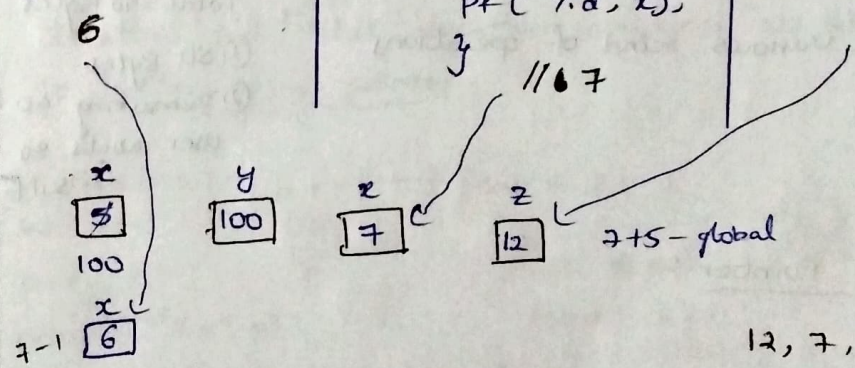
```
main ( )
{
  x = 5;
  p (&x);
  pf ("%d", x);
}
```

```
void p (int *y)
{
  int x = *y + 2;
  q (x);
  *y = x - 1;
  pf ("%d", x);
}
```

// 7

```
void q (int z)
{
  z = z + 2;
  pf ("%d", z);
}
```

// 12



12, 7, 6.

* Types of Pointers:-

1. 'NULL Pointer'
2. 'void Pointer'
3. 'wild Pointer'
4. 'Dangling pointer'

① NULL Pointer:

→ If a pointer not pointed to any memory location, then it is called 'NULL Pointer'.

Ex: int *p = NULL;

*p = 32; //error/unexpected behaviour.

→ By default NULL pointer variable has value '0'.

Advantages of Pointers:

→ Creating Dynamic Memory Allocation.

Use of NULL Pointer:

→ Whenever we 'request for memory' the response can be type of either 'Address' or 'NULL'.

```
if (P==NULL)
{
    == X
}
else
{
  various kind of operations
}
```

Memory

10 bytes per each	100	200	300
	400	500	600
	700	800	900

Total = 90 Bytes

① 50 Bytes

② Remaining 40 bytes, User wants 80 Bytes (Insufficient)

② Void Pointer:

Ex:

```
float a = 5.0;
int *ap;                    Invalid
ap = &a;
pf("value of a is: %d", *ap)  — Unexpected behaviour
```

→ 'void pointer' can hold 'address of any datatype'.

```
float a = 5.0;
void *ap;                   Valid
ap = &a;
pf("value of a is: %f", *(float *ap));  — 5.0
```

→ we need to use 'type casting'.

→ If you would like to accept the address of any kind of variable during runtime then go for 'void pointer'.

⑤ Wild Pointer :-

→ Wild pointer going to behave abnormally.

Ex:-

```
int *ptr;
*ptr=12;
```

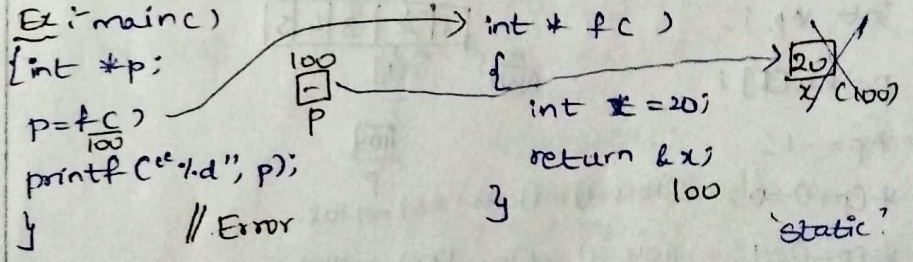
→ We created pointer & didn't mention to which address I need to point.

→ Garbage value replaced by 12.

→ so recommended to initialize pointer variables.

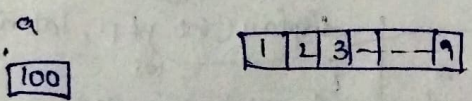
④ Dangling pointer :-

→ If any kind of pointer, pointed to certain memory location but when that location contains 'No data' then it is 'Dangling pointer'.



* Arrays with pointers :-

a[ ] = {1, 2, ——, 9}.



```
main()
{ a[] = { ... };
  f(100 a) a[]//2
} f(or &a);

f(int b[])
{ int *k
  //process array elements
}
```

b: 100 → b[]//2

Array,	Receiving	Pointer
a[],	b[],	k[]
i[a],	i[b],	i[k]
*(a+i*2)	*(b+i*2)	*(b+i)

→ a[i] = *(a+i*size)

i^th ⇒ (a+i*s) ⇒ Address

a[i] ⇒ value ⇒ *(a+i*size)

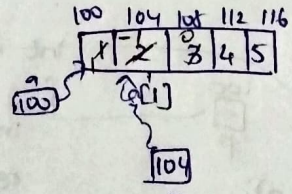
Ex:

```
int a[] = {1, 2, 3, ____, 9};
int b[] = a;
int *k = a;
```

a[] ≡ i[a] ≡ b[] ≡ i[b] ≡ k[] ≡ *[k+i]

Ex:

```
int a[] = {1,2,3,4,5}
int *p, i;
p = &a[1];
*p = -1;
*(p+1) = 0;
*(p-1) = 1;
```



(104+1) ⇒ (104+1*4) ⇒ 108

(104-1) ⇒ (104-1*4) ⇒ 100

Find result of a?

// 1, -1, 0, 4, 5

Ex:

```
main()
{ int a[] = {10,20,30,40,50}
  display(100 a, 5);
  display(a+2, 5);
}
```

100, 104, 108, 112, 116

108

```
display(int *p, int n)
{ for(i=0; i<n; i++)
  { pf("%d", p[i]);
  }
}
```

100

108

// 10, 20, 30, 40, 50

// 30, 40, 50, gr, gr

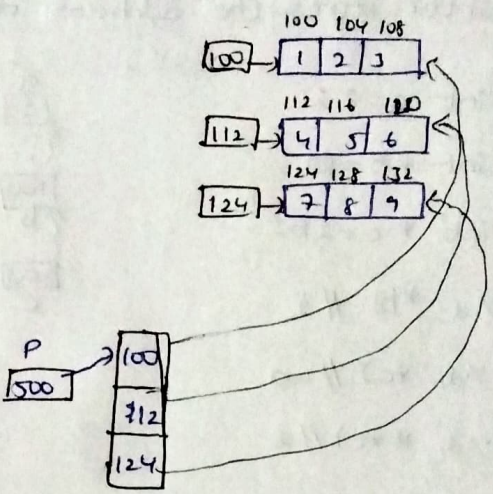
* Passing Two-Dim Arrays:

Ex-1

```
int a[ ] = {1, 2, 3};
int b[ ] = {4, 5, 6};
int c[ ] = {7, 8, 9};
```

2-D

1	2	3
4	5	6
7	8	9



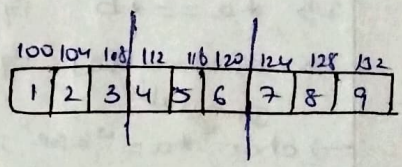

```
int *p[3];

p[0] = &a; (or) a
p[1] = b;
p[2] = c;
```

Accessing:

```
A[1][1] ≡ P[1][1] ≡ *(*(P+1)+1)   // 5
A[2][1] ≡ P[2][1] = *(*(P+2)+1)   // 8.
```

Ex-2:

```
main()
{ int a[ ][3] = {1,2,3,4,5,6,7,8,9};
  f(a);
}
f(int b[ ][3])
{
  for(for c)
    b[i][i]
}
```



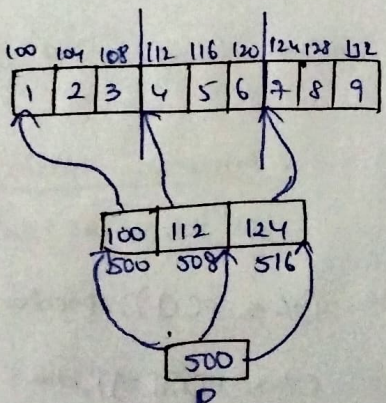

```
f(int (*P)[3])   // P is a pointer + '3' elements.
{
  P[1][1]; 5
  P[2][1]; 8
    (or)
  *(*(p+1)+1);
  *(*(p+2)+1);
}
```

P++;

P+1×12

⇒100+12=112

```
*(*(516)+1)
  *(124+1)
  *(124+1×4)
  *(128) = 8.
```



* Double pointer (or) Pointer to pointer:-

→ Pointer stores the address of another pointer.

Ex:- int a=2;

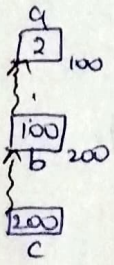
int *b=&a;

int **c=&b;

(%d, *b) // 2

(%d, *c) // 100

(%d, **c) // 2

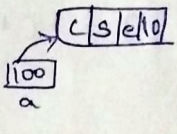


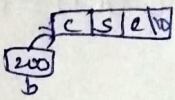
* strings & pointers:-

Ex:- → mutable

char a[ ]= "cse";

char b[ ]= "cse";





Is a==b? No. [Diff Addresses]

Is *a==*b? Equal

Is a[0]==b[0]? Equal

→ Immutable

char *a="cse";

char *b="cse";

is (a==b) ? Equal

is *a==*b ? Equal

is a[0]==b[0] ? Equal

2-d:-

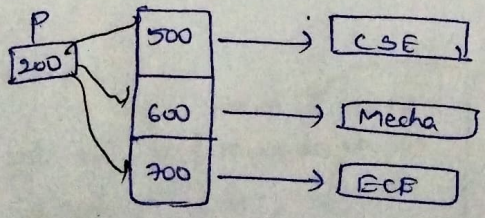
char a[3][8] ={"cse", "Mecha", "ECE"};

sizeof(a) // 24 bytes.

~~char (*p)[8] = {}~~

↓

char *p[8]={



(%s, *p+2); E

(%s, *(*p+1)+2); NO o/p

(%s, p[1]); Mecha

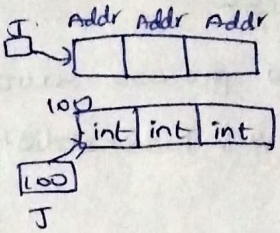
(%s, p[1][2]); cha

(%c, p[1][2]); c

(%s, *(p+2)); ECE.

int *J[3];

int (*J)[3];



* Pointers to functions :-

f(int)

f(array)

f(function)

Ex:-

```
int sum(int a, int b)
{
   ==========
}
```

sum 100

Ex:-

f(a) ⇒ f(int a)

f(arr) ⇒ f(int a[] or ___>

f(Jk) ⇒ f(int (*p)(int))

```
int Jk(int b)
{
   ==========
}
```

```
main
{
  f(Jk);
}
```

Steps:-

① Declare function prototype

int (*fp)();

② Initialization, fp = Jk;

③ call, fp(5), (*fp)(5);

* Pointer constant :-

int *const p;

p = &a;

p = &b; X

fixing Address

constant to pointer :-

const int *p;

p = &a;

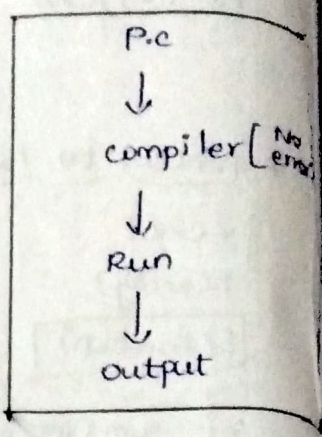
*p = 7; X

fixing value.

a: 2 (100) b: 3 (200)

* Preprocessor Directives:-

→ We are going to process source code before the compilation. # include <stdio.h>



Rules:-

① starts with '#'

② Doesn't end with ';'

Some of Preprocessor Directives:-

#define
#ifdef
#ifndef
#if
#else
#endif
#error

} useful to create user defined functions

Build in:-

__DATE__
__TIME__

} produces string as output [%s]

① #define:-

→ To define 'constants'

→ Before compilation phase

Ex:-

```
#define MAX 5
#define S(x) x*x.
```

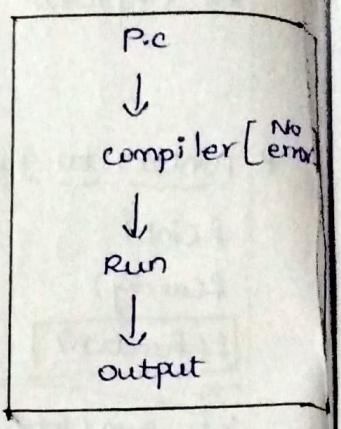
Code:-

```
#include <stdio.h>.
#define MAX 5
#define S(x) x*x
void main()
{ const int a=7;
  printf("value of max is %d", MAX); //7
  printf("mult of 5+2 two times is %d", S(5+2)); //17
  printf("value of S(5) is %d", S(5)); //25
}
```

5+2*5
5+10+2
17
BOOM!!

* Preprocessor Directives:-

→ We are going to process source code before the compilation. # include <stdio.h>



Rules:-

① starts with '#'

② Doesn't end with ';'

Some of Preprocessor Directives:-

#define
#ifdef
#Ifndef
#if
#else
#endif
#error

} useful to create user defined functions.

Build in :-

__DATE__
__TIME__

} produces string as output [%s]

① #define :-

→ To define 'constants'

→ Before compilation phase.

Ex:-

```
#define MAX 5
#define S(x) x*x.
```

code:-

```
#include <stdio.h>.
#define MAX 5
#define S(x) x*x
void main()
{ const int a=7;
  printf("value of max is %d", MAX); //7
  printf("Mult of 5+2 two times is %d", S(5+2)); //17
  printf("value of S(5) is %d", S(5)); //25
}
```

5+2*5+2 → 5+10+2 → 17 BOOMS

② #ifdef :-

→ It gives result in 'true' or 'false' format.

Ex:-

```
#include <stdio.h>
#define VERSION5
void main( )
{
  #ifdef VERSION5
  printf("version 1 code");
  #endif
} //version 1 code.
```

③ #ifndef :-

→ It is quite opposite to ifdef.

→ If it is Not available returns 'True' and available returns 'false'.

④ #if :-

→ same as if statement.

Ex:-

```
#include <stdio.h>
#define MAX 5.
void main( )
{
  #if MAX>2
    printf("max is >2");
  #else MAX<2
    printf("max is <2");
  #endif
}
```

⑤ #error :-

```
#else
  #error "MAX is less than 2.
```

* Build in :-

(1) __DATE__ and __TIME__

Ex:-

```
void main()
{
  pf("%s", __DATE__);   // Jan 27 2023
  pf("%s", __TIME__);
}
```

* Dynamic Memory Allocation :-

→ If memory is created at compile time – static memory allocation,

→ If memory is going to create at run time – Dynamic memory allocation.

Ex:-

int Arr[50]; 20 Bytes

for 2 students → 8 Bytes – Memory Wasting

for 100 students → 400 Bytes – Insufficient

→ We can overcome this by 'Dynamic Memory Allocation'

→ Data will store in 'Heap Memory'.

Ex:-

```
#include<stdio.h>
int main()
{
  int a[5]; i;
  for(i=0; i<5; i++)
  {
    a[i]=i;
  }
  for(i=0; i<5; i++)
  {
    pf("%d\n", a[i]);
  }
}
```

static allocation.

```
#include<stdio.h>
int main()
{
  int i, n; a[5];
  pf("Enter value of n");
  sf("%d", &n);
  int b[n];
  for(i=0; i<n; i++)
  {
    b[i]=i;
  }
  for(i=0; i<n; i++)
  {
    pf("%d", b[i])
  }
}
```

Dynamic Allocation.

* Types :-

1. malloc ( )
2. calloc ( )
3. free ( )
4. realloc ( )

} all returns void data.

① malloc ( ) :-

int *p = (int *) malloc (n x size of (datatype))
int here

→ If memory is sufficient then it returns 'address'
else returns 'NULL'

Ex :-

```
#include <stdlib.h>   <- Note
void main()
{
  int n;
  pf("Enter n value: ");
  sf("%d", &n);
  int *fp = (int*) malloc (n * sizeof (int));
  if (fp == NULL)
  { pf("memory not created"); }
  else
    pf("Memory created");
  for (i=0; i<n; i++)
  {
    fp[i] = i;
  }
  for (i=0; i<n; i++)
  { pf("%d\n", fp[i]);
  }
}
```

O/p :-

```
Enter n value : 8
0
1
2
3
4
5
6
7
```

② calloc ():

→ It allocates memory 'contiguously'

→ By default calloc initialize with 'zero' whereas Malloc with 'garbage values'

→ calloc takes 'two parameters'

① 'No. of elements' you want to store.

② 'size' of each element.

→ 'Malloc' takes only 'one parameter'

Ex:-

```
#include <stdlib.h>
void main()
{
  int *cp=(int*)calloc(n, sizeof(int));
  if(cp==NULL)
    pf("Memory not created");
  else
    pf("memory created");
}
```

③ free():

```
  else
    pf("memory created");
    for(i=0; i<n; i++)
    {
      cp[i]=i+1;
    }
    for(i=0; i<n; i++)
    {
      pf("%d", cp[i]);
    }
    free(cp);
    pf("memory deleted");
    pf("after deletion: %d", cp[1]);
}
```

0.

By default

Because memory freed

④ realloc :-

→ To extend, we can use realloc

Ex:-

```
{pf C"Enter no of students");
sf ("%d", &n);
realloc (cp, n*sizeof(int));
for (i=6; i<n; i++)
  cp[i]= i+1;
for (i=0; i<n; i++)
  pf C"%d\n", cp[i]);
}
```

O/P

```
n=6
1
2
3
4
5
6
n=10
1
2
3
4
5
6
7
8
9
10
```

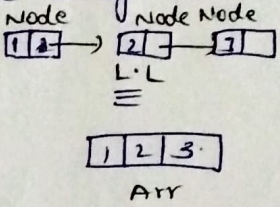
* Applications of Dynamic memory allocation:-

① In Linked Lists → To overcome arrays drawbacks.

② In Heaps → In Data structures

③ In stacks → FILO/LIFO

④ In Queues → FIFO/LILO etc



* command line Arguments:-

→ If inputs is given by command prompt then they are called "command line Arguments"

Ex:-

```
#include <stdio.h>
void main (int argc, char *argS[ ])
{
  pf C"NO. of arguments : %d", argc);
  pf ("first argument : %s", argS[0]); # File Name
  if (argc==3)
  {
    argS-E
    pf C"first argument : %s", argS[1]); // First element's value
    pf C"second argument : %s", argS[2]); // value of second element
  } pf C"%d", atoi(argS[1])+atoi (argS[2]));
  else
  { pf C"Enter valid no of arguments");
  }
}
```