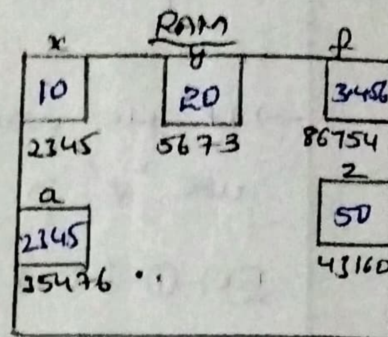
5-12-2022

UNIT - V. POINTERS.

* pointer:

→ 'pointer' is a variable which can stores the address of another variable.

→ It can stores only one variable's address at a time.



Ex: ① int x=10, y=20, z=50;

int *a=&x;
(or)
int *a=&y;
(or)
int *a=&z;
} pointers.

(ii) float f=3.456;

float *fp=&f; - pointer

(iii) char c='n';

char *cp=&c; - pointer.

Ex: pointer to pointer

① int x=10;

int *a=&x; - pointer

int ** ap=&a; - pointer to pointer

② float f=3.456;

float *fp=&f; - pointer

float ** fpp=&fp; - pointer to pointer.

③ char c='n';

char *cp=&c; - pointer

char ** cpp=&cp; - pointer to pointer.

→ we can also do like this.

(i) int x, *xp;

x=10;

xp=&x; — No need of '*' again.

(ii) char ch='A';

char * cp;

cp=&ch; — No need of '*' again.

→ If we want 'value' of pointer, we definetly use '*' in printf.

Ex:- ① int x=10;

int *xp=&x;

printf("x=%d", *xp); — 10.

② int x=10, y=20, z=30.

int *xp=&x;, *yp=&y, *zp=&z;

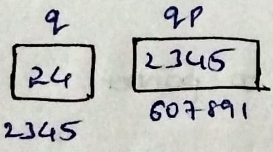
int sum;

sum = *xp + *yp + *zp; (10, 20, 30)

printf("%d", sum); — 60.

printf("%d", xp+yp+zp); — collapse (sum of addresses).

*



printf("address=%u"; qp);

printf("value of q=%d", *qp);

*qp=24;

* Note:

→ we cannot do add, sub, mul, div on any two addresses.

* **Arithmetic operations:**

① **Addition:**

int a=5, b=10, s;

int *ap=&a; *bp=&b, *sp=&s;

*sp = *ap + *bp

*sp = 5+10.

② **Multiplication:**

*mp = *ap * *bp

→ we need to give space

③ **Division:**

*dp = *ap / *bp

* **Increment / Decrement:**

→ bp++ → Address Increases

→ *bp++ → value Increases after address

→ (*bp)++ → Value Increases

→ 'ap+1' [i.e, we can add constant to address]. & substraction also but not Mul & Division].

'ap+1' means ap+1 * 2 bytes for 'int'

Ex: int a=10

int *ap=&a;

a	ap
$\boxed{10}$	$\boxed{123}$
123	1345

ap+1 = 123+1 * 2

= 123 + 2

= 125

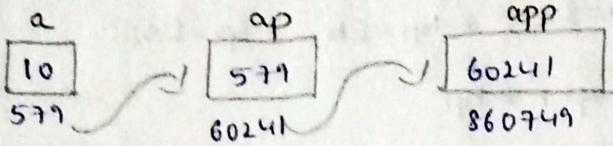
↓

New address.

* **Note:** We can also do relational operators [>, <, =] on pointers.

* Pointers to pointers :-

Ex: int a=10, *ap=&a, **app=≈



printf("a=%d", **app); — 10

printf("a=%d", *ap); — 10

printf("&ap=%u", app); — 60241

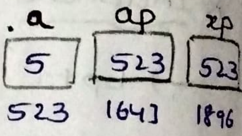
printf("&a=%u", ap); — 579.

* Note :-

→ we can assign variable address to many pointers.

Ex: int a=5 ; *ap=&a ;

int *xp=&a;



Here *ap = *xp = &a = 523

* Null pointer :-

→ 'Null' is a 'predefined constant' stored in the stdio.h, stdlib.h & string.h Header files.

Ex: ① int *ap; → stores garbage value

int *ap = NULL; → stores 0.

② char *ap = NULL;

③ float *ap = NULL;

→ 'Null' represents that it doesnot storing any variable address. It is an empty pointer stored with "zero".

* Generic pointer :

→ If we want to store the address of any variable. we need to declare that as void. then it is called 'Generic pointer'.

→ It is for just "retriving the value"

Ex:-

```
void *ptr; → stores anytype.
ptr = &a;
ptr = &f;
printf("a=%d", *((int *) ptr));     } we can't do both
printf("f=%f", *((float *)ptr));    } at a time.
```

* Note :-

→ we can allocate memory 'dynamically'.

→ 'Pointer' can store address of 'one variable only at a time'.

→ we need to do 'typecasting' to 'retrive' the value stored in the 'Generic pointer'.

Ex: ①
```
char c ='A';
void *ptr;
ptr = &c;
printf("c=%c", *((char *) ptr));
```

②
```
float f=7.6;
int a= 6;
void *ptr;
ptr= &a;
printf("a=%d", *((int *) ptr));
ptr = &f;
printf("f=%f", *((float *) ptr));
```

* Call by Mechanism in Functions:-

→ Two types:-

1. "call by value"
2. "call by reference"

1. call by value :-

```
void fun (int a, int b)
{
    x++;
    y--;
}   return x;

main ( )
{
    int a=10, b=20;
    a=fun(a, b);
    printf("a=%d b=%d", a,b)        — 10, 20.
}
```

2. call by reference :-

```
void fun ( int *a, int *b)
{
    (*x)++;
    (*y)--;
}
main ( )
{
    int a=10, b=20;
    fun (&a, &b);
    printf("a=%d, b=%d", *a,*b)        — 11, 19.
}
```

→ It is used without 'return'.

* Array of pointers :-

Ex-1 :-

```
int x=10, y=20, z=30;
int *ptr[3];
ptr[0] = &x;
ptr[1] = &y;
ptr[2] = &z;
```

Ex-2 :-

```
int x[5], y[5], z[5];
int *ptr[3] = { x, y, z };
```

store Base Addresses

Note :-

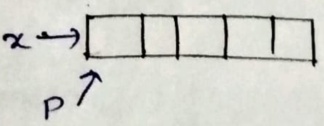
```
int x[5];
int * ptr = &x; || x; || &x[0];
int *ptr = &x; || x; || &x[0];
```

All three are same

Two dimensional Array :-

```
int x[5][5];
int (*ptr)[5] = x;
```

Accesing one-d :-



By x :- x[0], x[1], x[2] - By Index

	For Address	For value
By P :-	ptr+0	*ptr+0
	ptr+1	*ptr+1
	ptr+2	*ptr+2

```
→ for (i=0; i<5; i++)
  { printf("%d", (ptr+i));   - for Address
  }
  {
    printf("%d", (*ptr+i));   - for values.
  }
```

Accessing in 2-d :-

For ~~value~~ Address :-

```
for (i=0 ; i<5 ; i++)
{ for (j=0; j<5; j++)
  {
    printf ("%d" (*(ptr+i)+j));
  }
}
```

For values :-

```
for (i=0; i<5; i++)
{ for (j=0; j<5; j++)
  {
    printf ("%d" *(*(ptr+i)+j));
  }
}
```

Accessing in 3-d :-

```
int x[3][2][3].
int (*ptr)[2][3] = x;
for (i=0 ; i<5; i++)
{
 for (j=0; j<5; j++)
  {
   for (k=0; k<5; k++)
   {
     printf("%d" (*(*(ptr+i)+j)+k));   - for Address
     printf("%d" *(*(*(ptr+i)+j)+k));  - for values.
   }
  }
}
```

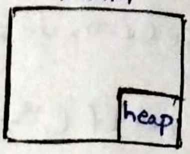
* Dynamic Memory Allocation:-

For ex;

10	20	30	40	50

int a[10];

RAM



But we need to store 3 elements only - Memory waste

But if we need to store 11 elements then - Memory Insufficient

→ Then 'heap memory' is useful to store at runtime.

→ ① Malloc ( )
② calloc ( )
③ Free ( )
④ realloc ( )

} 'Functions' (or) 'routines'.

① Malloc ( ):-

void * malloc (size_t size)
↓
unsigned int

→ We need to type cast.

→ malloc (3 * size of (int)) → for 3 elements.

Syntax:-

$\boxed{\text{ptr=(int *) malloc (3* size of (int))}}$

→ In the case of 'failure', it returns 'NULL'.

Ex:-
After program

```
if(ptr == NULL)
{
    printf("Memory not allocated");
}
```

→ After program, 'free (ptr)' - We need to free the memory

* **Example :-**

```
int main ( )
{
 int n, i, *ptr;
 printf ("Enter total no. of values");
 scanf ("%d", &n);
 ptr =(int *) malloc (n * size of (int));
 printf ("Enter values: ");
 for (i=0; i<n; i++)
 {
    scanf ("%d", (ptr+i));
 }
 printf ("The entered values are : ");
 for(i=0; i<n; i++)
 {
   printf ("%d", *(ptr+i));
 }
 free (ptr);
}
```

② Calloc ( ):-

→ Full form is 'contiguous allocation'.

→ It is a built-in function in 'stdlib.h'.

→ used to dynamically allocate 'multiple blocks' of memory & each block is of same size.

Syntax:-

```
void * calloc (size_t n, size_t size);
```

n ↓ No. of blocks

size ↓ size of each block

Ex:- calloc (5, size of (int));

5×4=20 Bytes.

0	0	0	0	0

←4→←4→←4→←4→←4→

```
int * ptr;
⇒ ptr = (int *) calloc (5, size of (int));
        ⇓
ptr = (int *) malloc (5 * size of(int));
```

20 bytes

Note:-

→ By default, 'calloc' stores '0'

→ By default, 'Malloc' stores 'Garbage values'

③ free ( ):-

→ It will release the dynamically allocated memory

→ It is built-in- function in 'stdlib.h'

→ $\boxed{\text{void free (pointer).}}$

Ex:-

```
ptr = (int *) malloc (3 x size of(int));
free (ptr);
ptr = NULL;
```

Heap

10	2	5

1000

ptr 1000

→ It doesnot erase previous data.

printf("%d", *(ptr+1))

→ If we print previous data. — Undefined Behaviour.

May be we get 2 || 0 || garbage value

(4) realloc :-

-) To extend, we can use realloc.

Ex:-

```
{pf ("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 ("%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

* constant to pointers :-

```
int a=10, b=20;
int *ptr;
    ptr=&a
    printf("%d", ptr);
    printf("%d", *ptr);

    ptr=&b
    ptr → address  }
    ptr → value    } accessing
```

→ int * const ptr=&a;

*ptr=20; → We can change address but not value.

→ const int *ptr=&a;

*ptr=30; ×

ptr=&b; → We can change value but not address.

→ const int * const ptr=&a;

We can change value & address also.