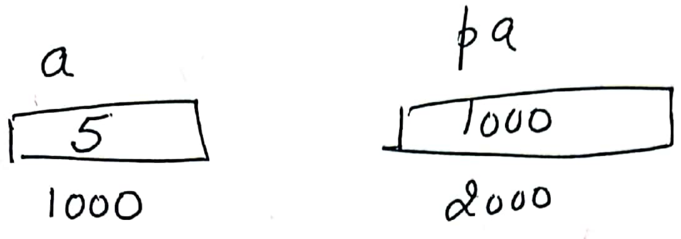
Pointers

Pointers are the variables that stores the address of a variable (int, Char, float, array or structure variable).
Basically, It points to the variable..



Suppose a is an integer variable - which has a value 5.
and It is stored at memory location 1000.
Then pa is pointer variable that points to a or
It stores the address of a.
Pointer variable can have its own address. (Here It is 2000)

How we will define or declare pointer variable

Egs:

```
int a = 5;
int * pa;
pa = &a;
```

This is Pointer to Int

```
float b = 25.5;
float *pb;
pb = &b;
```

Pointer to float

```
Char c = 'Z';
Char * pc;
pc = &c;
```

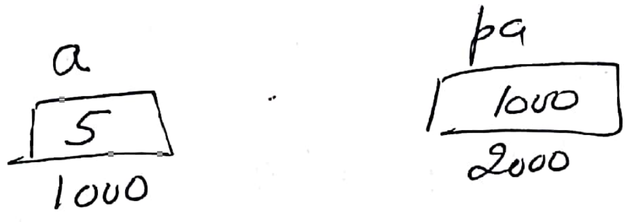
Pointer to Char.

```
int a [5];
int * p;
p = a;
```

Pointer to Array.

What ~~[illegible]~~

1) Eg



```
main()
{
 int a=5;
 int *pa;
 pa= &a;
printf("%d", a);          5
printf("%u", &a);         1000
printf("%u", pa);         1000
printf("%u", &pa);        2000
printf("%d", *pa);        5
}.
```

O/P

From above eg, we see value 5 is represented or pointed by a or *pa;

a = *pa;

* is indirection unary operator or "value at address" operator.

a → 5

*pa → (value at) 1000 → 5

What is the type of Pointer ?

~~Pointer~~ If we are writing

```
int *p ;
float *p ;
Char *p ;
```

(Unsigned Int)
%u

That doesn't mean pointer type is int, float or char.

Type of pointer is unsigned int

and it is represented by "%u"

unsigned means only +ve integers because addresses can never be negative.

Q Operations that are allowed on pointers

1. Pointers can be compared. (Comparison of 2 pointers are allowed) provided both pointers points to same object (int, Char or float).

```
int *pa, *pb;
if (pa == pb)  ✓
{
=
}
```

2. Subtraction of 2 pointers are allowed

```
int c = pa - pb
```

It will return the no. of integers between the 2 pointers.

eg int a [5] = {10, 20, 30, 40, 50};

int *pa;

pa = &a; (pointer to array)

int c = a [4] - a [2];

O/P will be 3 integers

not 50 - 30 = 20 ✗

pa = printf

---

③ Addition of a constant to pointer. or subtraction of a constant from pointer. is allowed

eg

```
pa = pa + 1
pa = pa + 2
pb = pb - 1;
pb = pb - 3;
```

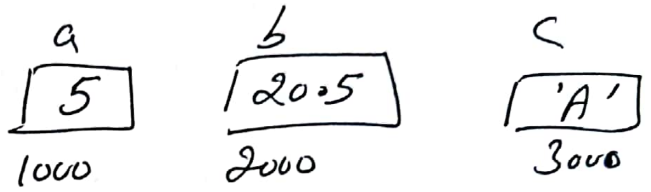
eg

```
int a [5] = {10, 20, 30, 40, 50};
int *pa;
pa = a;                              O/P
printf ("%d", *pa);         →        10
pa = pa + 2;                →   It will add 2 to pointer
                                Means it will jump 2 integers.
printf ("%d", *pa);         →   30
```

```
pa = pa-2;                    It will again move 2 integers
                              left
printf("%d", *pa);   ⟶   10.
}
```

④ <u>Pointers can be incremented or decremented</u>

```
pa++;
pb--;
```



```
eg   main()
     {
     int a = 5, *pa;
     float b = 20.5, *pb;
     char c = 'A', *pc;
     pa = &a;
     pb = &b;                                           o/p
     pc = &c;
     ~~print~~f("%d%f%c", a, b, c);          5      20.5      A
     a++;
     b++;                                    value will be incremented
     c++;
   printf("%d%f%c", a, b, c);                6      21.5      B
   printf(~~"~~"%u %u %u", pa, pb, pc);      1000   2000      3000
     pa++;
     pb++;
     pc++;
   printf("%u %u %u", pa, pb, pc);           1002   2004      3001
   }
```

From above e.g, we see

pa++; ∵ it is pointing to integer So it will jump 2 bytes So pointer will move from. 1000 becomes 1002

pb++; ∵ It is pointing to float, pointer will move from 2000 ~~to~~ 2004

pc++ ∵ Char takes 1 byte, So pointer will move from 3000 to 3001

⟹ So Pointer when incremented, always points to the immediate next location of its data type.

So Pointer will be incremented depending upon which value it's pointing to.

Operations that are not allowed on pointers.

① Addition of 2 pointers are not allowed

pa + pb ✗

② Multiplication of a constant by a pointer is not allowed

pa = pa * 2 ✗

③ Division of constant by pointer is not allowed

pa = pa/2; ✗

Differen

Funct

Scanned with CamS

Difference between Call by value / Call by Reference

Functions can be called by its value and by its address or reference.

Call by value	Call by reference
(1) In this, value is passed to a function.	(1) In this, address of a variable is passed to a function
(2) In this, photocopy of the variables are created	(2) In this, no photocopy is created.
(3) There is a local change	(3) There will be a global change.
(4) In this, if the formal parameters are altered, then actual arguments will not be affected.	(4) In this, if formal arguments are altered, then actual arguments will get affected.
(5) It is a slow process	(5) It is a fast process because searching by address is always fast rather than by name
(6) eg ↓	(6) eg ↓

~~void swap(int a, int b);~~ ~~void swap(int *a, int *b);~~

```
void swap(int, int);
void main()
  {
 int a, b;
 scanf("%d%d", &a, &b);
```

~~c = a;~~
~~a = b;~~
~~b = c;~~

```
      printf("%d%d", a,b);
  swap(a, b);
printf("%d%d", a, b);
  getch();
  }
void swap(int a, int b)
  {
  int c;
  c = a;
  a = b;
  b = c;
 printf("%d%d", a, b);
  }
```

```
void swap(int *, int *);
void main()
  {
 int a, b;
scanf("%d%d", &a, &b);

Swap(&a, &b);
printf("%d%d", a, b);
  getch();
  }
void swap(int *a, int *b)
 {
 int c;
 c = *a;
 *a = *b;
 *b = c;
printf("%d%d", *a, *b);
 }
```