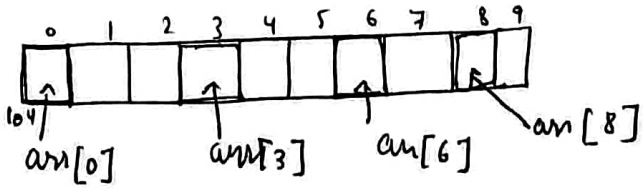
Pointers - 2

int arr[10]; →



address of arr[0]

⇒ &arr[0] ⟶ 104

&arr ⟶ 104 ← base address

```
int arr[] = { 12, 14, 16, 18 };

cout << arr;
cout << arr[0];
cout << &arr;
cout << &arr[0];
```

address let = 104

12

104

104

Catch → In case of arrays.

arr → Base Address

&arr → Base Address

&arr[0] → Base Address

⇒ int * p = &arr;

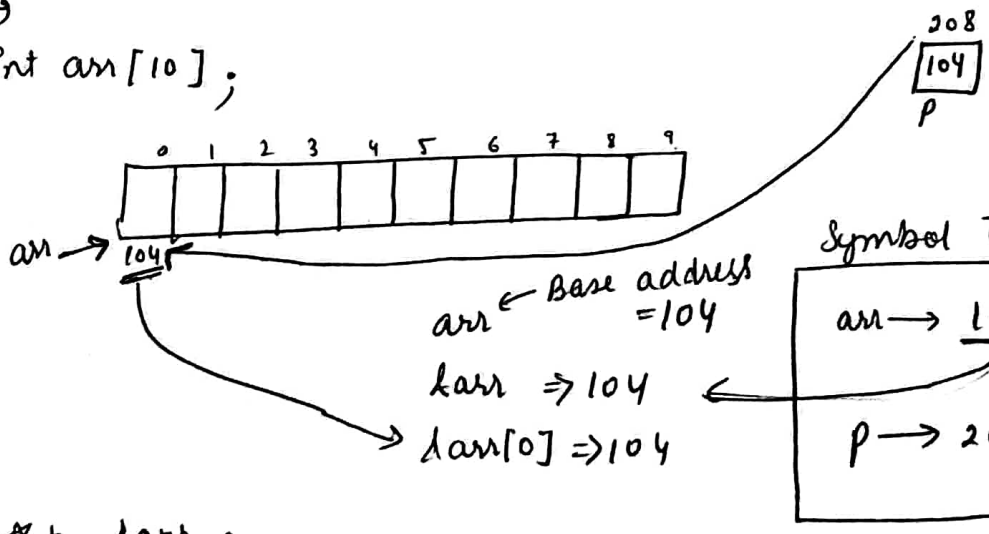
cout << p; ← p ~~arr~~ address

cout << &arr; } different

↖ address of arr (base address)

BTS →

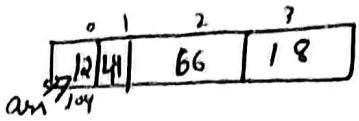
int arr[10];



int *p = &arr;

cout << &p; ⟶ 208

cout << p; ⟶ 104




```
cout << *arr ;
```

↳ o/p → 12

```
cout << *arr + 1 ;
```

→ *arr = 12 + 1 = 13

```
cout << *(arr) + 1;
```

*(arr) = 12 + 1 = 13

```
cout << *(arr+1);
```

*(arr+1) = 44

```
cout << *(arr+2);
```

o/p = ~~85~~ 66

```
cout << *(arr+3);
```

o/p = 18.

```
cout << arr[0]; // 12
cout << arr[1]; // 44
cout << arr[2]; // 66
cout << arr[3]; // 18
```

⇒ $\boxed{arr[i] \Rightarrow *(arr+i) \Rightarrow i[arr]}$ same thing

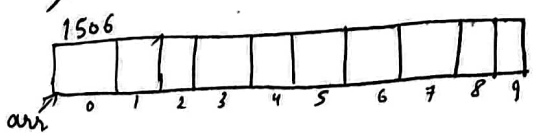
internally → *(i+arr) same as *(arr+i).

```
int i = 0;
cout << arr[i];      ⇒ o/p → 12
cout << i[arr];        o/p → 12
cout << *(arr+i);      o/p → 12
cout << *(i+arr);      o/p → 12
```

*arr
↳ value at Base address
↓
*(arr+0)
↓
*(arr[0])

⇒ arr is the pointer to the first location of the array.

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



⊙ Can we change the base address of the array?

```
arr = arr + 2;
```

→ not allowed.

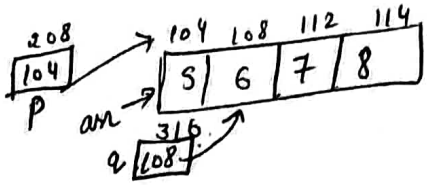
```
int *p = arr;
p = p+1;
```

→ allowed.

So we can do this by pointers. (access subpart of an array).

Why we were not able to change by arr = arr + 2;
The address of base address is already mapped ~~with~~ in symbol table, and we can't change entry in pointer. That's why we are not able to change it.

→ int arr[4] = {5, 6, 7, 8};
int *p = arr;
int *q = arr+1;



→ cout →

arr → 104
&arr → 104
arr[0] → 5
&arr[0] → 104
p → 104
&p → 208
*p → 5
q → 108

&q → 316
*q → 6
*(p)+1 → 5+1 = 6
*(p)+2 → 5+2 = 7
*(q)+3 → 6+3 = 9
*(q+4) → Segmentation fault/ garbage value/ error.

Difference between Pointer and Array →

	Array	Pointer
1. size →	int arr[10] sizeof(arr); → 40 bytes	1. int *p = &arr; sizeof(p); → 8 bytes
	cout << sizeof(*p) << endl; → size of integer → 4/2 according to system architecture.	
	Total space taken by array.	Total space taken by pointer.
2.	arr = arr+1 → ✗ not possible	2. p = p+1 → possible

arr = 104 cout << arr+1 108	int *p = arr+1; p 108	arr = arr+1; not allowed. error because arr is a constant pointer. You can't change arr's in symbol table.

Character arrays →

char ch[10] = "Babbar"; ch →

104									
B	a	b	b	a	r				

char * c = ch; → no difference in ch and &ch.

cout << c; → Babbar → why not 104? (The address)

So we observed that cout's behaviour is different for character pointers.

cout << ch; → Babbar
cout << &ch; → Base address → 104
cout << ch[0]; → B
cout << &c; → address of pointer
cout << *c; → B
cout << c; → Babbar.

*c = *(c+0)
= c[0] → B

char name[10] = "SherBano";
char* cptr = &name[0];

104								
s	h	e	r	B	a	n	o	'\0'
0	1	2	3	4	5	6	7	8

104 (216) cptr

cout →
name → SherBano
&name → 104
*(name+3) → r
cptr → ~~SherBano~~ SherBano ✓
&cptr → 216
*(cptr+3) → r
cptr+2 → erBano
*cptr → S ⇒ *cptr = *(cptr+0) = cptr[0] = S
cptr+8 → '\0' ← Null character.

c → whole string (0 to 7 index

c+2 → 0+2 to 7th index

S	h	e	r	B	a	n	o	'\0'
0	1	2	3	4	5	6	7	8

Special case →

```
char ch = 'k';
char* cptr = &ch;
cout << cptr;
```

o/p → kAcdx ← Null character encountered.

k will be printed and other garbage value will be printed till it encounters a Null character.

```
char name[10] = "Babbar";
cout << name;
```

✓ allowed

o/p → Babbar

```
char* c = "Babbar";
cout << c;
```

✓ allowed but BAD Practice.

o/p → Babbar.

BTS. `char name[10] = "Babbar";`

2 Step process →

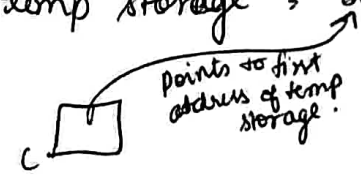
① temparory storage → "Babbar"

② Memory change → copy to name array's storage.

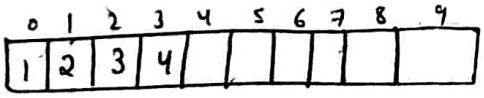
`char* c = "Babbar";` → BAD Practice

2 Step process →

① temp storage → "Babbar"

② 

Pointer with function →




```
main(){
    int arr[10] = { 1, 2, 3, 4 };
    cout << "Size of array is " << sizeof(arr);
    solve(arr);
}
void solve(int arr[]){
    cout << "Size " << sizeof(arr);
}
```

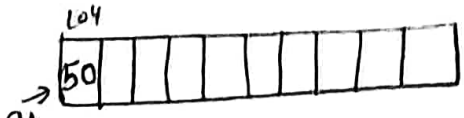
// o/p → 40

o/p → 8

because pointer is passed, not exact array

main()

int arr[10];



sizeof (arr) → 40

solve () int arr[ ])

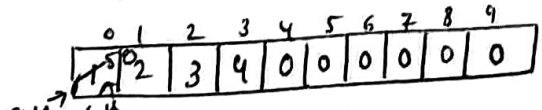
sizeof (arr) → 8

arr[0] = 50 → This will update in actual array.

*(arr+0)

so that's why array is pass by reference (because pointer is passed).

```
main(){
    int arr[10] = {1, 2, 3, 4};
    cout << sizeof(arr);
    for (int i=0; i<10; i++){
        cout << arr[i];
    } cout << "Calling solve fun";
    solve(arr);
    for (int i=0; i<10; i++){
        cout << arr[i];
    }
}
```



print → 1, 2, 3, 4, 0, 0, 0, 0, 0, 0

print → 50, 2, 3, 4, 0, 0, 0, 0, 0, 0

```
void solve(int arr[]){
    cout << "size" << sizeof(arr);   → 8
    cout << arr;   → 104
    cout << &arr;   → 216
    arr[0] = 50;
}
```

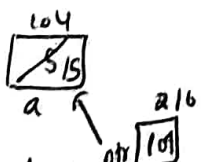
arr[0] = 50

*(arr+0) = arr[0] → 50

Yes, we can have two memory location with same name. (scope is changed.)

Ques→

```
main(){
    int a = 5;
    int *ptr = &a;
    cout << a;   → o/p → 5
    solve(ptr);
    cout << a;   → o/p → 15
}
```

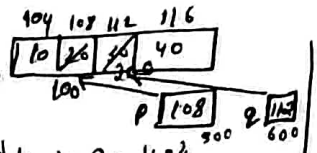



```
solve(int *p){
    *p = *p + 10;
}
```

*(104) = *(104) + 10
= 5 + 10
= 15

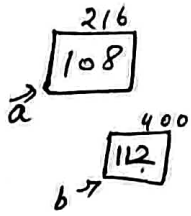
Ques →

```
main ( ){
    int an[4] = {10,20,30,40};
    int *p = &an[1];
    int *q = &an[2];
    update (p, q);
    //print entire array.
}
```



o/p → 10, 100, 200, 40

```
update ( int *a, int *b){
    *a = 100;
    *b = 200;
}
```



*a = 100 ↓ *(108) → 100

*b = 200 ↓ *(112) → 200

H.W →

Q → Benefits of all pointer concepts.

Q → Pointer to function topic, why do we need this?