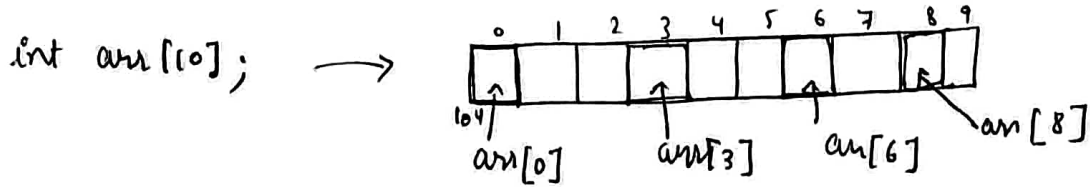


Pointers - 2



address of `arr[0]`

⇒ `&arr[0]` → 104

`&arr` → 104 ← base address

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

<code>cout << arr;</code>	address let = 104
<code>cout << arr[0];</code>	12
<code>cout << &arr;</code>	104
<code>cout << &arr[0];</code>	104

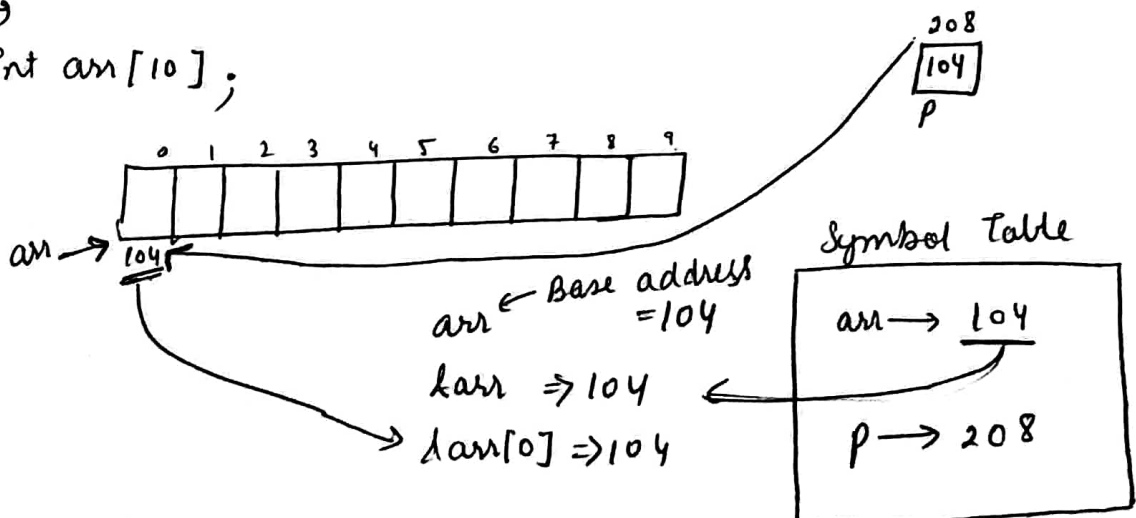
Catch → In case of arrays.

`arr` → Base Address
`&arr` →
`&arr[0]` →

⇒ `int *p = &arr;`
`cout << p;` ← `p` own address
`cout << &arr;` ← address of `arr` (base address)
 { different

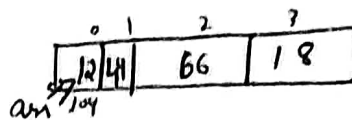
BIS →

`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 = 66

cout << *(arr + 3) ; o/p = 18

cout << arr[0] ; // 12

cout << arr[1] ; // 44

cout << arr[2] ; // 66

cout << arr[3] ; // 18

⇒ arr[i] ⇒ *(arr + i) ⇒ i[arr] same thing

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

int i = 0;

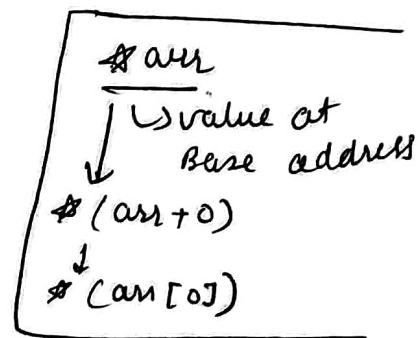
cout << arr[i] ; → o/p → 12

cout << i[arr] ; o/p → 12

cout << *(arr + i) ; o/p → 12

cout << *(i + arr) ; o/p → 12

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



int arr[10];



① Can we change the base address of the array?

arr = arr + 2 ; → not allowed.

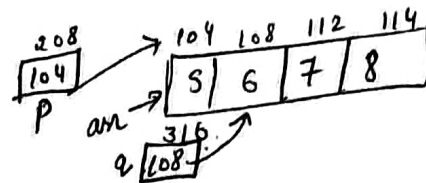
int *p = arr ;

*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` → 208
`&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.

① Difference between Pointer and Array →

1. size →

Array
`int arr[10]`
`sizeof(arr);` → 40 bytes

Pointer
`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. ⊗

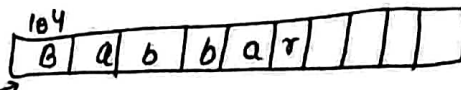
`arr = arr + 1` → X
 not possible

`p = p + 1`
 → possible

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

Character arrays

char ch[10] = "Babbar";



A horizontal array of 10 cells. The first six cells contain 'B', 'a', 'b', 'b', 'a', 'r'. The last four cells are empty. Above the first cell is the address '104'. An arrow labeled 'ch' points to the first cell.

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.

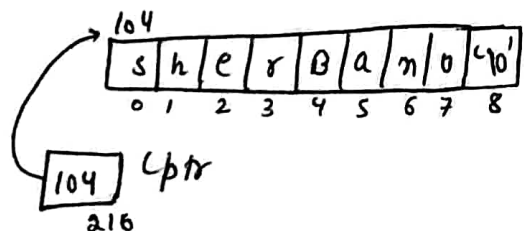
cout << ch; → Babbar
cout << &ch; → Base address → 104
cout << ch[0]; → B
cout << &c; → address of pointer
cout << *c; → B
cout << c; → Babbar.

*c = *(c+0)
= c[0] → B

char name[10] = "SherBano";

char *cptr = &name[0];

cout << name → SherBano
&name → 104



*(name+3) → r

cptr → ~~SherBano~~ SherBano ✓

&cptr → 216

*(cptr+3) → r

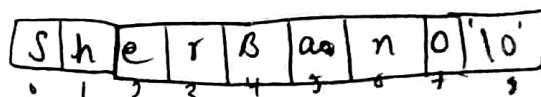
cptr+2 → erBano

*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



① Special case →

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

o/p → k A c d x ← Null character encountered.

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

① char name[10] = "Babbar"; ✓ allowed
cout << name; o/p → Babbar
char* c = "Babbar"; ✓ allowed but BAD Practice.
cout << c; o/p → Babbar.


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

2 Step process →

- ① temporary storage → "Babbar"
- ② memory change → copy to name array's storage.

char* c = "Babbar"; → BAD Practice

2 Step process →

- ① temp storage → "Babbar"
- ②  points to first address of temp storage.

① Pointer with function →

main() {

```
int arr[10] = { 1, 2, 3, 4 };
```

```
cout << "Size of array is " << sizeof(arr);
```

```
solve(arr);
```

```
}
```

```
void solve(int arr[10]) {
```

```
cout << "size" << sizeof(arr);
```

```
}
```

0	1	2	3	4	5	6	7	8	9
1	2	3	4						

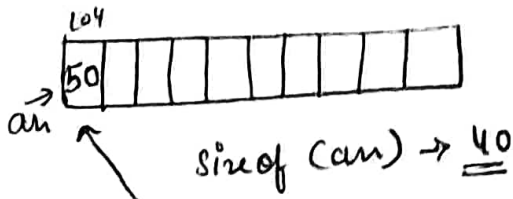
o/p → 40

o/p → 8

↑
because pointer is passed, not exact array

main()

int arr[10];



solve() and arr)

104

arr

size of (arr) -> 40

arr[0] = 50

→ This will update in actual array.

*arr

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

main()

int arr[10] = {1, 2, 3, 4};

cout << size of (arr);

for (int i=0; i<10; i++) {

cout << arr[i];

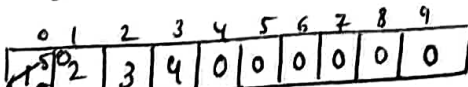
} cout << "calling solve fun";

solve(arr);

for (int i=0; i<10; i++) {

cout << arr[i];

}



arr

print -> 1, 2, 3, 4, 0, 0, 0, 0, 0, 0

print -> 50, 2, 3, 4, 0, 0, 0, 0, 0, 0

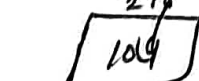
void solve(int arr[]) {

cout << "size" << size of (arr);

cout << arr; -> 104

cout << *arr; -> 216

arr[0] = 50;



arr

arr[0] = 50

*arr = arr[0] = 50

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

Ques -> main()

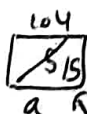
int a = 5;

int *ptr = &a;

cout << a;

solve(ptr);

cout << a; -> o/p -> 15



a

ptr

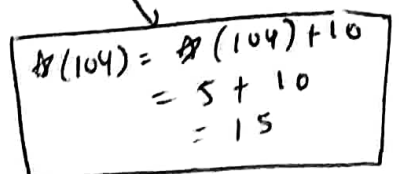
solve(int *p) {

*p = *p + 10;

400

104

p



Ques →

main() {

int an[4] = {10, 100, 200, 40};

int *p = &an[1];

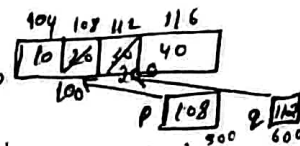
int *q = &an[2];

update(p, q);

// print entire array.

}

o/p → 10, 100, 200, 40

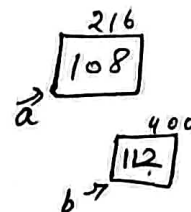


update(~~int~~ *a, int *b) {

*a = 100;

*b = 200;

}



*a = 100

↓

~~*a = 100~~

* (108) → 200

*b = 200

↓

* (112) → 200

H.W →

Q → Benefits of all pointer concepts.

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