Array level-1

25/09/23

What is Array?

- list of similar item
- collection of elements.
- Data structure.
- continuous memory block.

Array is a data structure which contain similar type of item.

only int → 1, 2, 3, 22 . . .

only char → 'a', 'b', 'c'

Why need Array?

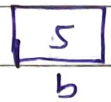
2 no → require to store ⇒ int a, b;

10 no → " " → int n_1 n_2 n_3 n_4 . . . n_{10} — 10 block

100000 →

for store large no. of data. we need Array.

int b = 5 ⇒



int a[10] ⇒

type of data which in store in this container.

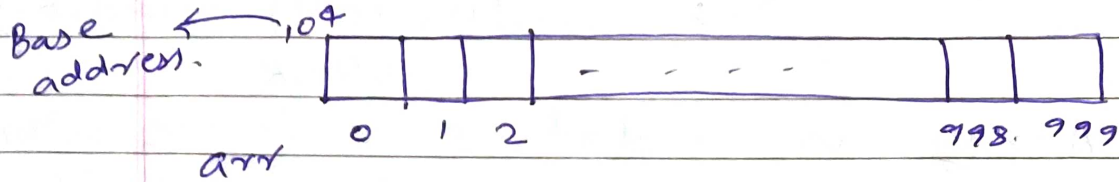
it take continuous memory to store similar type of data.

creation of array :→

int arr [1000]

- arr → array store array
- int → type data.
- name.
- [1000] → 1000 blocks.

int a = 5;
↓
4 byte.



- here it arr → have 4000 byte memory.
- It is static type allocation.

char alphabet [26];
↳ char type array.

int a = 5;

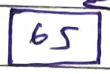
→ add. of this block = 104.

Symbol table	
Variable →	address
a	→ 104.
b	→ 1007

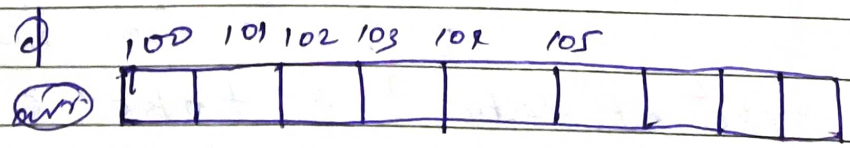
maped

char b = 'A';

1007 ↙ ASCII table.



int arr [10];



S.T

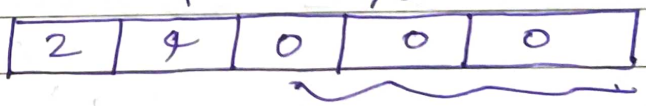
arr → 100

```
int a=5;
cout << "Address of a:" << &a << endl;

int arr[10];
cout << "Base address of arr is :" << &arr << endl;
cout << "Base add. of arr is:" << arr << endl;

cout << "size of arr : " << sizeof(arr) << endl;
```

same add.

Array Initialisation ⇒

```
int arr[] = { 1, 2, 3, 4, 5 };

int brr[5] = { 2, 1, 7, 9, 5 };

int crr[5] = { 2, 4 };
```



at remaining block it contain 0 → by default.

```
int drr[2] = { 2, 4, 9, 8, 7 };
```

→ gives error. because we try to insert more elements than its size.

```
int err[5];
```

all the block will contain garbage value by default.

```
int n;
cin >> n; → input (user)
int arr[n];
```

→ bad practice

because if user gives a big no. which is not available in our computer's memory then it gives error.

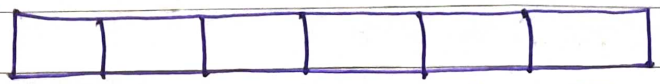
- it possible that memory are not available (continuos) as user demand for big size memory.
- To avoid this problem we use dynamic array allocation.

→ later study

Indexing in Array ⇒

To access a particular block of an array we require address of that block. for which we use index as we don't remember hexadecimal no. (add. of every block).

0- based index →

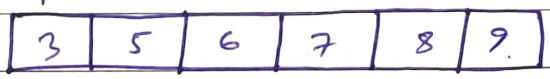


↳ 6 → size of array.

When we create n-size array then the index of last block will be (n-1).

Ex. Access block using index →.

int arr[6] = {3, 5, 6, 7, 8, 9}



arr[0] → 3
arr[1] → 5
arr[2] → 6
arr[3] → 7
arr[4] → 8
arr[5] → 9

Code ⇒

```
int arr[5] = {5, 8, 7, 12, 13};
```

output.

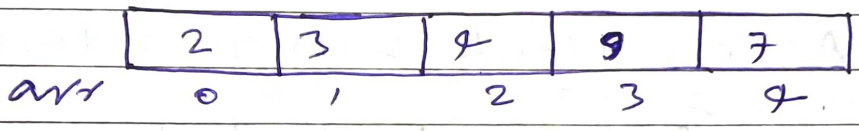
```
cout << arr[0] << endl; → 5
cout << arr[1] << endl; → 8
cout << arr[2] << endl; → 7
cout << arr[3] << endl; → 12
cout << arr[4] << endl; → 13
```

↳ it ~~would be~~ boring for large no. so, we use loop.

code ⇒ printing an array.

```
int arr[50] = {2, 3, 4, 9, 7};
    int n = 5;
    for (int i = 0; i < n; i++){
    cout << arr[i] << endl;
    }
```

dry run :→

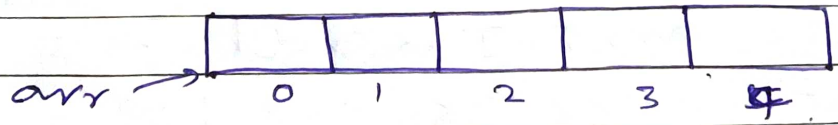


condition check n = 5

i++ ↳ i = 0 0 < 5 → true → print arr[0]
i++ ↳ i = 1 1 < 5 → true → print arr[1]
i++ ↳ i = 2 2 < 5 → true → print arr[2]
i++ ↳ i = 3 3 < 5 → true → print arr[3]
i++ ↳ i = 4 4 < 5 → true → print arr[4]
i++ ↳ i = 5 5 < 5 → false → stop → terminate loop.

Taking input in an array ⇒

```
int arr[5];
```




```
cin >> arr[0];
cin >> arr[1];
cin >> arr[2];
cin >> arr[3];
cin >> arr[4];
```

this will be tough for big size array so, we use loop.

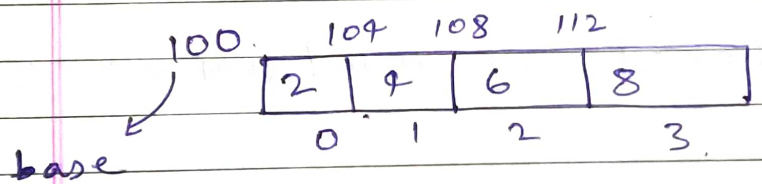
```
int arr[5];
int n=5;

for(int i=0; i<5; i++){
    cin >> arr[i];
}
```

similar dry run as we see for print element.

arr[i] ⇒ means ?

→ value at (base add. + size of datatype * index)

arr[2] = 6.



arr[2] ⇒ 100 + 4 × 2

value at → = 108.

Problem ⇒

① create 10 size array.
→ int arr[10];

② take i/p in that array.

③ double-up each value of that array

```
int n=10
for(int i=0; i<10; i++){
    cin>>arr[i];
}
```

```
for(int i=0; i<10; i++){
    arr[i] = 2*arr[i];
}
```

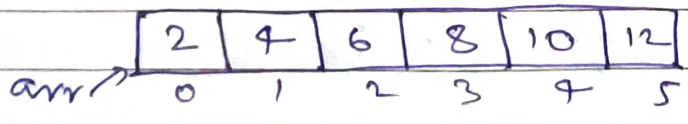
Problem ⇒

① Create 5 size array.
② take i/p in that array.
③ print total sum.

```
int arr[5];
 int n=5;

for(int i=0; i<n; i++){
      cin>> arr[i];
   }
int totalsum=0;
for(int i=0; i<n; i++){
    totalsum= totalsum +arr[i];
}
cout << totalsum;
```

Linear search in an Array :-

① find the target, in given array.



target = 10.

- traverse on every element of array and compare with target after traversing whole array if it not equal to target then return not found.

code ⇒

```
int arr[6] = {2, 4, 6, 8, 10, 12};
int target = 10;
int n = 5;
bool flag = 0;
for (int i = 0; i < n; i++){

    if (arr[i] == target){
    flag = 1; // found.
    // cout << "target found" << endl;
    break;
    }

}

if (flag == 1){
  cout << "target found" << endl;
}
if (flag == 0){
  cout << " target not found" << endl;
}
```

0 → not found.
1 → found.

no need to compare further if we found target.

Array & function ⇒.

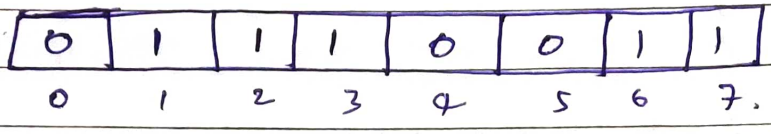
```
void solve (int arr[], int n ){
    for (int i=0; i<n; i++){
        cout << arr[i] << " ";
    }
}

int main (){
    int arr[5];
    int n=5;
    solve (arr, size);
}
```

⊙ whenever we send array into function we also pass its size.

```
// present → true
// absent → false.
bool linearsearch (int arr[], int n,
                   int target){
    for (int i=0; i<n; i++){
        if (arr[i] == target){
            return true;
        }
    }
}
```

```
    // not found.
    return false;

}

int main (){
    int arr[5] ={2, 3, 4, 5, 6};
    int target = 5;
    int n = 5

    bool ans = linearsearch(arr, n, target);

    if (ans == 1){
        cout << "target found" << endl;
    }
    else{
        cout << "target not found" << endl;
    }
    return 0;
}
```

Q count 0's and 1's in array.
In array only 0 & 1.



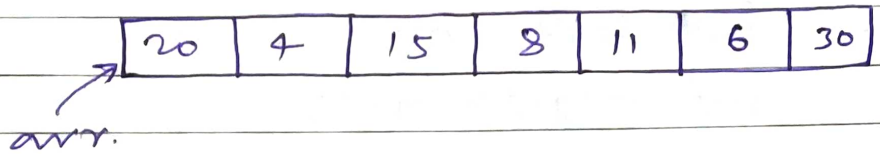

```
int main() {
    int arr[8] = {0, 1, 1, 1, 0, 0, 1, 1};

    // zero => 3
    // one => 5
    int size = 8;
    countzeroone(arr, size);

}
```

function

```
void countzeroOne(int arr[], int n){
    int zerocount = 0;
    int oneCount = 0;

    for(int i=0; i<n; i++){
        if(arr[i] == 0){
            zeroCount++;
        }
        if(arr[i] == 1){
            Onecount++;
        }
    }
```

```
cout << " No. of zero: " << zeroCount << endl;
cout << "No. of one: " << OneCount << endl;

}
```

Minimum no. in an array ⇒



① initilize as minAns = INT_MAX;

② traverse on array & compare with minAns.

```
for(int i=0 ; i<n; i++){
    if(arr[i] < minAns){
        minAns = arr[i];
    }
}
```

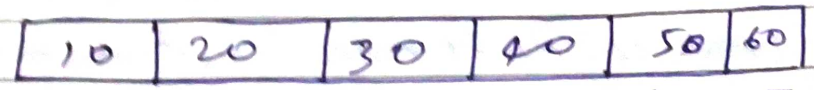
③ cout << minAns << endl;.

(*) (i) for minimum initilization → INT_MAX.

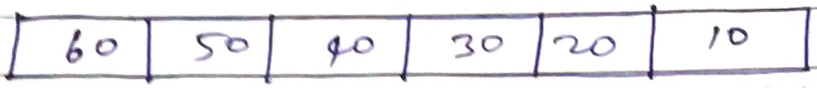
(ii) for maximum → INT_MIN.

Reverse an Array ⇒

i/p →

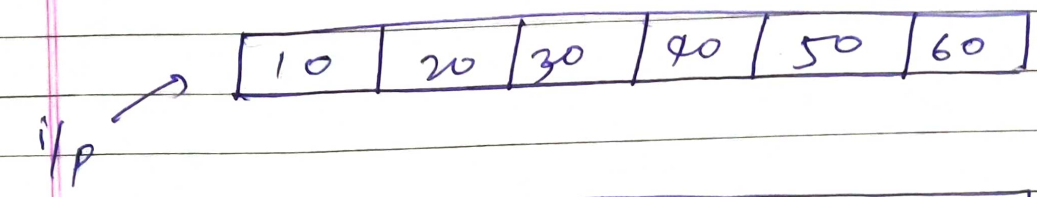


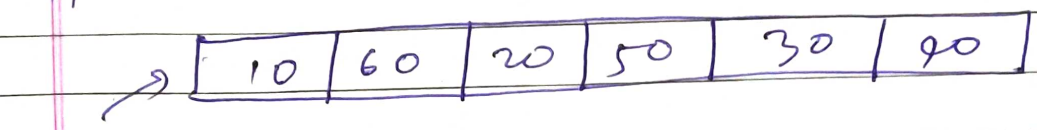
o/p →




```
void ReverseArray(int arr[], int n){
    int left = 0;
    int right = n-1;

    while(left <= right){
        swap(arr[left], arr[right]);
        left++;
        right--;
    }
}

int main(){
    int arr[] = {10,8,9,2,1,4,12};
    int n = 7;
    ReverseArray(arr, n);
// Array has been reversed we
    can print it.
}
```

Extreme print in an array ⇒






```
int left = 0;
int right = n-1;
while (left <= right){
cout << arr[left] << " " << arr[right]
     << " ";
left ++;
right --;
}
```

In case of odd.

i/p ⇒ 10, 20, 30, 40, 50

o/p ⇒ 10, 50, 20, 40, 30, 30

(X) because left & right on same index.

```
while (left <= right){
if (left == right){
cout << arr[left] << endl;
}
else{
print both left & right.
}
```