ARRAY POINTERS & POINTER ARITHMATIC

In C programming, if we want to access multiple memory locations of the same data type, we can use array pointers. We can simply access the data by **dereferencing** the pointers pointing to it. Suppose we have an array of length 3, and the elements are 1, 2, and 3.

```c
#include <stdio.h>

int main(){
    int x = 1;
    int y = 2;
    int z = 3;
    int *parr[3] = {&x, &y, &z};

    for(int i = 0; i < 3; i++){
        printf("Address of %d is %p\n", i+1, &parr[i]);

    }
    return 0;
}
```

The output will be

```
Address of 1 is 0061FF04
Address of 2 is 0061FF08
Address of 3 is 0061FF0C
```

Here, in the above example, we can see the memory location of each variable is different which is stored in an array.

The syntax of the array pointer is simple.

*Data_type (*name)[size_of_the_array]*

While naming the variable of the array we must use ().

One thing to note down that, the pointer that points the 0^{th} element of the array and the pointer that points the whole array are completely different.

```c
#include <stdio.h>

int main(){
    int arr[5] = {1,2,3,4,5};

    int *p, (*parr)[5];
    // here *p will points an integer whereas *parr points an whole array

    p = arr; //here p points to the 0th element

    parr = &arr; //parr points the whole array

    printf("address of p = %p   address of parr = %p\n", p,parr);

    p++;
    parr++;

    printf("address of p = %p    address of parr = %p", p,parr);
    return 0;
}
```

OUTPUT

```
address of p = 000000000061fe00   address of parr = 000000000061fe00
address of p = 000000000061fe04    address of parr = 000000000061fe14
```

Here, p is a base-type integer, which is 4 bytes, whereas the base-type parr is the size of the arr which is in this case 20 bytes.

```c
#include <stdio.h>

int main(){
    int arr[] = {1,2,3,9,10};

    int (*parr)[5];
    int *p = arr;
    parr = &arr;

    printf("sizeof(p) = %d, sizeof(*p) = %d\n", sizeof(p), sizeof(*p));
    printf("sizeof(parr) = %d, sizeof(*parr) = %d\n",  sizeof(parr), sizeof(*parr));

    return 0;
}
```

OUTPUT

```
sizeof(p) = 8, sizeof(*p) = 4
sizeof(parr) = 8, sizeof(*parr) = 20
```

So when we are using parr++, the pointer is shifted after 20 bytes.

Pointer Arithmetic

Pointer addition and subtraction

Suppose we have a pointer pt and an array of length 6. Now we want to point to the address of the first element of the array. So the memory address of the first element will be pt = &array[0].

```c
#include <stdio.h>

int main(){
    int arr[6] = {};
    int *pt = &arr[0];

    printf("%p", pt);

    return 0;
}
```

Now suppose we want to find out the memory location of the 4th element of the array. Now we can simply write pt = &arr[3], since the 4th element will be in the 3rd index.

```c
#include <stdio.h>

int main(){
    int arr[6] = {};
    int *pt = &arr[3];

    printf("%p", pt);

    return 0;
}
```

There is another way to do this as well which is just adding.

```c
#include <stdio.h>

int main(){
    int arr[6] = {};
    int *pt = &arr[3];

    printf("%p", pt);

    pt = &arr[0];
    pt = pt+3;// adding

    return 0;
}
```

OUTPUT

```
000000000061fe1c
000000000061fe1c
```

pt = pt + 3 refers the memory location of the 3rd index of the pointer. Here, (pt = pt+3) is equivalent to (pt = &arr[3]).

Here what is happening during the pointer addition is that it adds 4 bytes in each increment. And int type is 4 bytes so for each increment the pointer moves to 4 bytes of space.

```c
#include <stdio.h>

int main(){
    int arr[6] = {};
    int *pt = &arr[6];

    int x = sizeof(arr)/sizeof(arr[0]);
    for (int i = 0; i< x; i++){
        printf("The address of the %d index is %p\n",i,pt+i );
    }

    return 0;

}
```

OUTPUT

```
The address of the 0 index is 000000000061fe00
The address of the 1 index is 000000000061fe04
The address of the 2 index is 000000000061fe08
The address of the 3 index is 000000000061fe0c
The address of the 4 index is 000000000061fe10
The address of the 5 index is 000000000061fe14
```

Pointer subtraction is also similar to the addition. In addition, while adding we were accessing the pointer to the next element of the index after 4 bytes. Whereas in case of subtraction we just have to go back 4 bytes to access the previous memory location of the element of the array.

```c
#include <stdio.h>

int main(){
    int arr[6] = {};
    int *pt = &arr[6];

    int x = sizeof(arr)/sizeof(arr[0]);
    for (int i = x; i>= 0; i--){
        printf("The address of the %d index is %p\n",i,pt+i );
    }

    return 0;

}
```

OUTPUT

```
The address of the 6 index is 000000000061fe30
The address of the 5 index is 000000000061fe2c
The address of the 4 index is 000000000061fe28
The address of the 3 index is 000000000061fe24
The address of the 2 index is 000000000061fe20
The address of the 1 index is 000000000061fe1c
The address of the 0 index is 000000000061fe18
```

Subtraction of two pointer

Suppose, we want to know what the increment is between two elements of array through two pointers. The subtraction of two pointer basically gives the increment between the two pointers.

Note that the two pointers must be of same type.

```c
#include <stdio.h>

int main(){
    int arr[6] = {20,30,50,64,48,65};
    int *ptr1, *ptr2;

    ptr1 = &arr[0]; // pointing to the first element of the array
    ptr2 = &arr[5]; // pointing to the last element of the array

    int x = ptr2 - ptr1;

    printf("The increment is %d",x);

}
```

OUTPUT

```
The increment is 5
```

Here the increment between the first index and the last index is 5. What's happening here is that suppose the memory location of the 0^{th} index is 1000 so by the 5^{th} index the memory location will be 1020. And we put the memory location of the 0^{th} index in ptr1 and 5^{th} index in ptr2. After subtracting ptr1 from ptr2 we get 20 and since the size of int type is 4 bytes so 20/4 = 5. So the increment is 5.

Pointer increment and decrement

Let's start with the concept of **post increment.** Post increment is such that first the value will be assigned then the increment will be done.

```c
#include <stdio.h>

int main(){
    int arr[] = {2,16,45,8,6,87,65,49,897,6,87};
    int *pt = &arr[0]; //the address of first element

    printf("%d\n", *pt++);
    printf("%d", *pt);

    return 0;
}
```

OUTPUT

```
2
16
```

Here, we can see that the first output is 2, which is the first element. Let the base address be 1000. Now what pt++ is doing is first assigning the value of the address

1000, then incrementing it to the next element address, which is 1004, and then printing the value of that address.

Another thing is **pre-increment**. In pre-increment, the value is assigned after the increment is done.

```c
#include <stdio.h>

int main(){
    int arr[] = {2,16,45,8,6,87,65,49,897,6,87};
    int *pt = &arr[0]; //the address of first element

    printf("%d\n", *++pt);
    printf("%d", *pt);

    return 0;
}
```

OUTPUT

```
16
16
```

Similarly, the decrement also happens in this way. It also happens in post-decrement and pre decrement way.

```c
#include <stdio.h>

int main(){
    int arr[] = {2,16,45,8,6,87,65,49,897,6,87};
    int *pt = &arr[2]; //the address of third element

    printf("%d\n", *pt--);
    printf("%d", *pt);

    return 0;
}
```

OUTPUT

```
45
16
```

Here, the pointer points to the address of the 3rd element, and it first prints the 3rd element, and after that, it decrements and prints the 2nd element.

```c
#include <stdio.h>

int main(){
    int arr[] = {2,16,45,8,6,87,65,49,897,6,87};
    int *pt = &arr[2]; //the address of third element

    printf("%d\n", *--pt);
    printf("%d", *pt);

    return 0;
}
```

OUTPUT

```
16
16
```

POINTER COMPARISION

In C we can simply compare two pointer by using and implementing all the operators.

```c
#include <stdio.h>

int main(){
    int arr[] = {20,32,65,647,97,3,4,9,65};
    int *ptr1 = &arr[0];
    int *ptr2 = &arr[3];

    if (ptr1 < ptr2){
        printf("ptr1 is small");
    }
    else{
        printf("ptr2 is small");
    }
}
```

These pointer comparison can be used to sort an array or finding some user input numbers or character in an array.