04/10/2023
Wednesday

POINTERS

Pointers are the variables that store the address of another variable.

Pointers only store the address. Any value other than address can't be stored in it.

* Creation →

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

- `int *` → Pointer to integer data
- `ptr` → Variable or pointer name
- `&a` → Address of a

So, we can say, ptr is pointer to integer data.

- char * ch → ch is pointer to character data.
- bool * flag → flag is pointer to boolean data.

* **Access →**

Accessing value stored at address stored in pointer. That value can be accessed using de-reference ~~operation~~ operator. Its symbol is asterisk (*).

```
int a = 5;
int *ptr = &a;
cout << *ptr ;
```
⟶ Output will be the value at address stored in ptr i.e. 5

* **Difference b/w reference variable and pointer →**

Reference variable is the different name given to same memory location. It doesn't take extra space in memory while pointer is a variable that stores the address of another variable and it takes additional space.

Code →

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a = 5;
    int *ptr = &a;
    cout<<"value stored in ptr: "<<ptr<<endl;
    cout<<"address of ptr: "<<&ptr<<endl;
    cout<<"accessing: "<<*ptr<<endl;
}
```

O/P =

```
value stored in ptr : 0x61ff0c
address of ptr : 0x61ff08
accessing: 5
```

* **Size of pointer →**

Size of a pointer is architecture dependent. In computer memory organization, each byte of memory is assigned a unique address, and these addresses are used to access and manipulate data stored in memory. When we declare a pointer in C++, the size of the pointer variable is determined by the size of the the memory addresses used by the architecture.
Overall, size of pointer depends on variety of factors such as target platform's architecture, compiler implementation and memory organisation.

Code →

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a=5;
    int *ptr=&a;
    cout<<"size: "<<sizeof(ptr)<<endl;
    char ch='N';
    char *cptr=&ch;
    cout<<"size: "<<sizeof(cptr)<<endl;
    bool flag=1;
    bool *bptr=&flag;
    cout<<"size: "<<sizeof(bptr)<<endl;
}
```

Output →

```
size: 4
size: 4
size: 4
```

* **Pointer Declaration →**

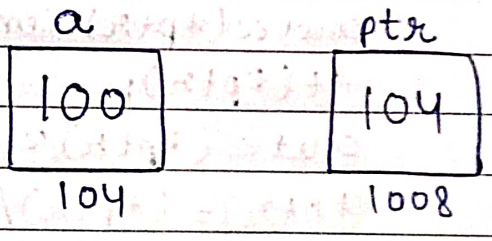
Bad Practice →
```
int *ptr;
cout << *ptr << endl;
```

Declaring pointer like this is a bad practice as it is trying to access a random memory which is not allowed. Generally, it gives run-time error or segmentation fault.

Good practice →
```
int *ptr = 0;
```
OR
```
int *ptr = nullptr;
```

Create a null pointer to avoid the above situation. This will also give run-time error or segmentation fault but this will help while debugging the code.

Q1 →
```
int a = 100;
int *ptr = &a;
a = a+1;
ptr = ptr+1;
cout<<a<<" "<<ptr;
```

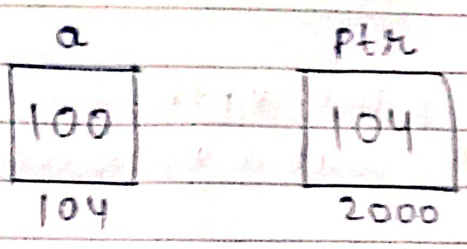


a = 101

ptr = 108 (As int is of 4 bytes, so, the next address after 104 would be 108.)

Q2 →

```
int a = 100;
int *ptr = &a;
a = a + 1;
cout << a << endl;
*ptr = *ptr + 1;
cout << *ptr << endl;
```

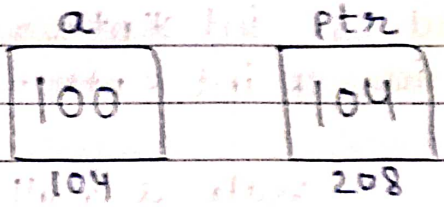


a = 101

*ptr = 102

Q3 →

```
int a = 100;
int *ptr = &a;
cout << a << endl; → 100
cout << &a << endl; → 104
cout << *a << endl; → error (integer can't be de-reference)
cout << *ptr << endl; → 100
cout << ptr << endl; → 104
cout << &ptr << endl; → 208
(*ptr)++;
cout << (*ptr) << endl; → 101
++(*ptr);
cout << (*ptr) << endl; → 102
(*ptr) = (*ptr)/2;
cout << (*ptr) << endl; → 51
*ptr = *ptr - 2;
cout << *ptr << endl; → 49
```



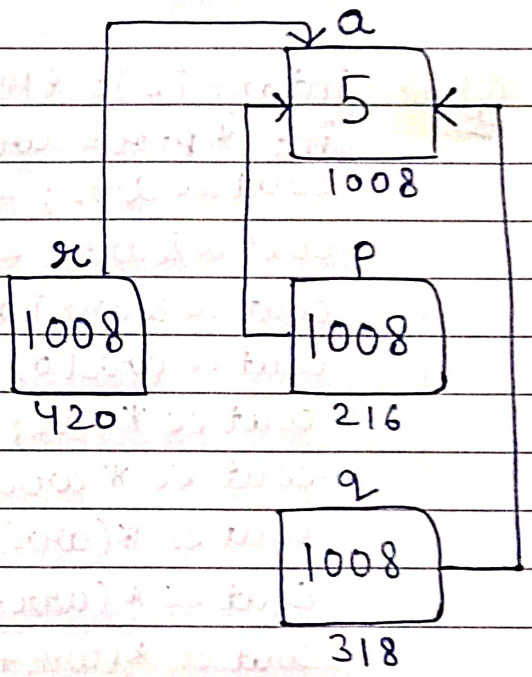
* Copying Pointer →

```
int a = 5;
int *p = &a;
int *q = p; // Pointer p is copied to q
cout << *p <<" "<<*q ; ⟶ 5  5
```

*p and *q both are referring to the value of a.

Q4 →

```
int a=5;
int *p= &a;
int *q = p;
cout<<a; ⟶ 5
cout << &a; ⟶ 1008
cout << *a; ⟶ error
cout << p ; ⟶ 1008
cout << &p; ⟶ 216
cout << *p ; ⟶ 5
cout << q ; ⟶ 1008
cout << &q; ⟶ 318
cout << *q; ⟶ 5
int *r = q;
cout << r ; ⟶ 1008
cout << &r; ⟶ 420
cout << *r; ⟶ 5
*p = *q+3;
 cout << *r; ⟶ 8
 *r = a/2;
 cout << a; ⟶ 4
```

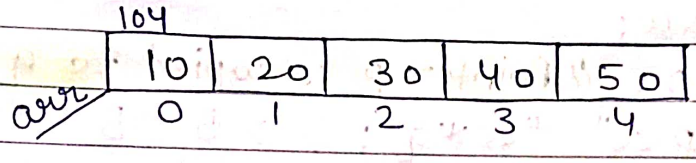


The value present at a can be manipulated by *p, *q and *r

__/__/____

* **Pointer with array →**

```
int arr[5] = {10, 20, 30, 40, 50};
```



Note →

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

Q1 →

```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr;
cout << arr ; → 104
cout << &arr; → 104
cout << &arr[0]; → 104
cout << arr[0]; → 10
cout << *arr; → 10
cout << *arr+1; → 11
cout << *(arr)+1; → 11
cout << *(arr+1); → 20
cout << *(arr+2); → 30
cout << *(arr+3); → 40
cout << *arr+0; → 10
cout << *(arr+0); → 10
```

{ These 3 statements prints the base address which is 104 here. }

Note →

```
*(arr+0) = arr[0]
*(arr+1) = arr[1]
*(arr+2) = arr[2]
```

Generalisation, ***(arr+i) = arr[i]**

OR ***(i+arr) = i[arr]**

Note → In case of integer arrays,

Correct { `int *ptr = arr;`
`int *ptr = &arr[0];`

Incorrect → `int *ptr = &arr;`

* **Why we can't do [arr = arr+1;] as [p = p+1;] →**

p is here pointer-to-integer data and arr is a name to an array.

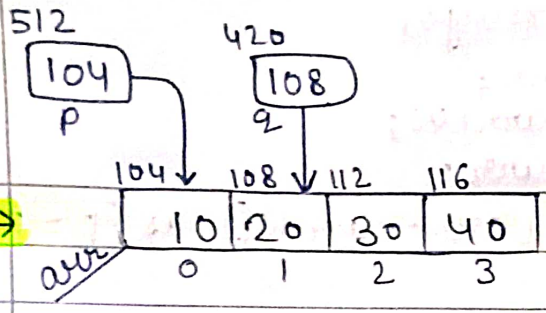
In C++, the name of array is actually a constant pointer to the first element of the array. This means that we can't modify the value of the array pointer itself (i.e. the address of the first element of array) by using pointer arithmetic.

So, if we try to do something like [arr = arr+1;], we will get a compilation error because here we are trying to modify a constant pointer. This is because the pointer arr is pointing to the memory location of first element of array and we can't change this address.

However, we can use pointer arithmetic to access other elements of array. for instance,

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr; // pointer to arr[0]
ptr++; // Now, points on arr[1]
cout << *ptr << endl; // prints 2
```

So, the summary is, we can't modify the address of first element of the array using pointer arithmetic, but we can use pointer arithmetic to access other elements in the array.



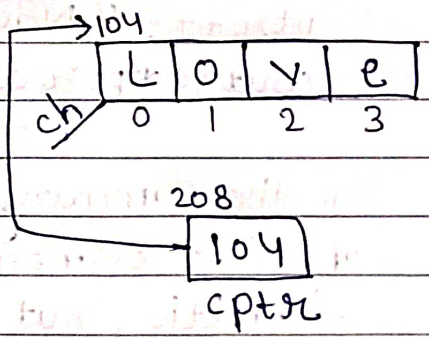
Q2 →

```
int arr[4]= {10,20,30,40};
int *p = arr;
int *q = arr+1;
cout<<p; ⟶ 104
cout<<&p; ⟶ 512
cout<<*p; ⟶ 10
cout<<q; ⟶ 108
cout<<&q; ⟶ 420
cout<<*q; ⟶ 20
cout<<*p+1<<endl; ⟶ 11
cout<<*(p)+2<<endl; ⟶ 12
cout<<*(q)+2<<endl; ⟶ 22
cout<<*(q+4)<<endl; ⟶ Garbage value
cout<<sizeof(arr); ⟶ 16
cout<<sizeof(p); ⟶ 4 (Pointer size is architecture dependent).
```

* **Pointer with char Array →**

Q1 →

```
char ch[50]="love";
char *cptr = ch;
cout<<ch; → love
cout<<&ch; → 104
cout<<*ch; → L
cout<<ch[0]; → L
cout<<&cptr; → 208
cout<<*cptr; → *(cptr+0) → cptr[0] → L
cout<<cptr; → love
```



Note → When we make a pointer for char-type data and tries to print that pointer, it will print the array characters rather than its address like in integer array.

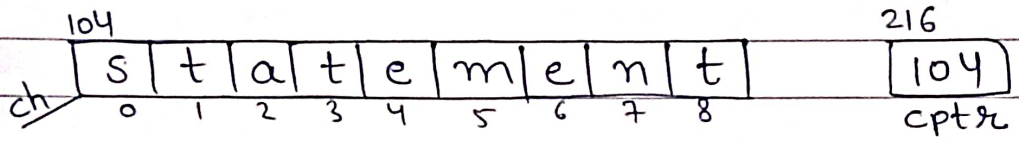
For instance,

```
int arr[3] = {1, 2, 3};
int *p = arr;
cout << p; → It will print address of first element of array.
```

While, in char array →

```
char ch[50] = "love";
char *cptr = ch;
cout << cptr; → It will print the whole string rather than its address.
```

Q2 →




```
Char ch[50] = "statement";
char *cptr = &ch[0];
cout << ch; → statement
cout << &ch; → 104
cout << *(ch+3); → t
cout << cptr; → statement
cout << &cptr; → 216
cout << *(cptr+3); → t
cout << cptr+2; → atement
cout << *cptr; → s
cout << cptr+8; → t
```

Note → In case of char arrays →

```
char *cptr = ch;      } Correct
char *cptr = &ch[0];  }
char *cptr = &ch; → Incorrect
```

Q3 →

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    char ch = 'A';
    char *cptr = &ch; → // Pointer to char data type
    cout<<cptr; → A◇|◇?
    cout<<*cptr; → A
```

Some random characters are printing here after A till we get null character