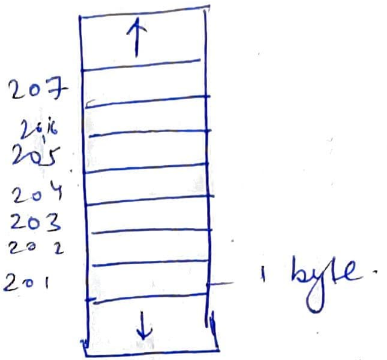
Pointers (C++)

memory (RAM)

int - 4 bytes of memory allocated.

char - 1 byte

float - 4 bytes.



address of bottomost byte = 0

int a;

a is stored at let's say [204 - 207].

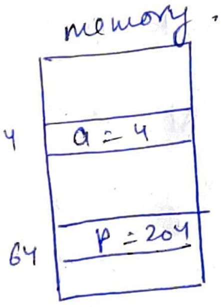
lookup table is maintained by computer.

a	int	204
c	char	209.

Now, we need to operate upon memory addresses using pointers.

Pointers - variables that store address of another variable.

memory.



int a; (variable a of type int)

int *p; (variable p of type pointer and points to the type int).

p = &a; (Now, p stores address of a variable a)

(& before a variable brings it's address)

a = 5;

print p // 204

print &a // 204

print &p // 64.

print *p // 5

✓ (putting a star in front of p gives value at address).

dereferencing

*p = 8

print a // 8.

p → address

*p → value at address p.

working with pointers

int a;

int *p; (pointer variable p is just defined like this. It has nothing to do with dereferencing done using *p afterwards into the program)..

int there means that p will store an int type variable.

p = &a (&a gives address of a and then stores it in pointer type p)

We can just consider addresses as a different datatype.

int b = 20;

int * p

*p = b ; (will the address in p change to point b?)

NO, we firstly need to explicitly allocate an address to p using ampersand (&).

So, in above case, value of a becomes 20 and p still points at a. To change the pointer we need

p = &b;

```
int a = 10
int *  p = &a;
```

(Remember, using a star before initialising a pointer and using star for dereferencing are different).

Pointer arithmetic

```
print p ;      // p is 2002 (say)
print (p+1) ;  // p is 2006
```

because int type has size of 4 bytes. To go to next integer, pointer value increases by 4.

Pointer types, Void pointers, pointer arithmetic

→ strict pointer types are needed for one type of variable.

int * → int
char * → char.

This is required we need to dereference data using a pointer.
↳ access / modify.

~~int a = 1025;~~
~~int * p = &a;~~

```
int a = 1025;
int * p;
  p = &a;
print (' size (int))
print ( ~~Address~~ p, *p )
print ( p+1 , *(p+1))

char * po;
po = (char *) p;      // typecasting

print ( size(char))
print ( Address → (po~~+1~~), value → (* po) )
print ( Address → (po+1) , value → ( *(po+1)) )
```

<u>Output</u>.

```
                                              201        200
                 1 byte.         202
               ⏞
→ 1025 =   00000000 .  00000000    00000100  00000001
                                                       ⏟
              203.                                let's say
                                                   address
                                                   here is
                                                     200.
```

size (int) → 4

p → 200

*p → 1025

p+1 → 204

*(p+1) → some garbage value stored in memory.

size (char) → 1

po → 200

* po → 1

po+1 → 201

*(po+1) → 4

} Now, pointer is char and thus it only moves forward 1 byte at a time.

// void poiwer - Genuic pointer

```
void * po;
po = p;
```

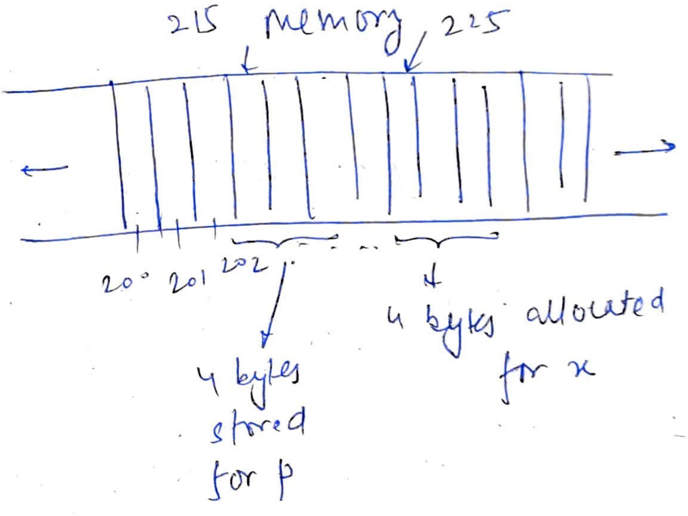
(No error here, we can store any type of pointer value in a void pointer variable).

However, void pointer cannot be dereferenced.

(po+1) → This also cannot be done.

Pointer to pointer.

```
int x = 5;
int* p;
p = &x;
```



Now, how can we store the address of pointer p in other variable.

p stores 225

let's say q is at 205

```
int** q;
```

↳ pointer to a pointer variable.

```
q = &p;
```

type of x → int

" " p → int*

" " q → int**

" " r → int***

We can go on like this.

```
int*** r;
r = &q; (valid)
```

r = &p (not valid)

Thus, r can store the address of a variable of type int ** only.

```
int x = 5;
int *p = &x;
*p = 6;
int** q = &p;
int *** r = &q;
```

205	215	225	230
215	225	6	205
q	p	x	r

```
print ( *p) ;              // 6
print ( *q) ;              // 225
print ( * (*q)) ;          // 6
print ( * (*r)) ;          // 225
print ( * (* (* r)) ;      // 6

*** r = 10;
print (x) ;                // x = 10 now.

**q = *p +2 ;              // x = 12 now.
```

Pointers as function arguments - call by reference

lets consider the code below.

```
void increment (int a):
{     a = a+1;
}

int main() {
    int a;
    a = 10;
    Increment(a);
    print(a)
}
```

This program prints a = 10 only because variable in increment is a local variable. and thus is different from variable in main.

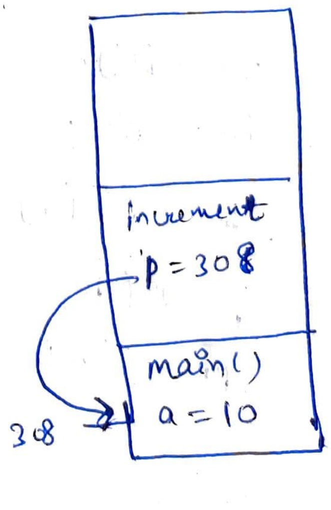
copying the value of a variable like this is called 'call by value'.

Now, what if we want to change the value of a in main when the function call to increment is made.

example

```
void increment (int* p)
{       *p = (*p)+1
   }.
int main () {
    int a;
    a = 10;
    Increment (&a);
    print (a)
}.
```

Here, output is 11..

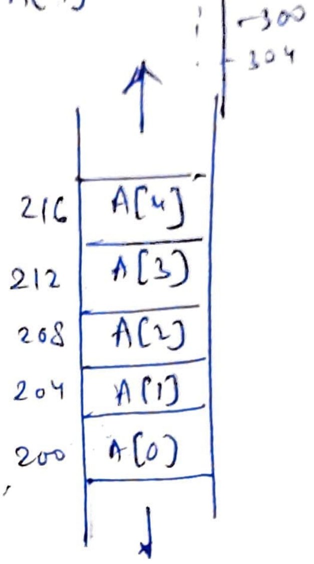


This is called call by reference.

Pointers and Arrays

int A[5]: → we create five integer spaces
A[0], A[1] ... A[4]

int x = 5; → let's say x is at 300.



```
int* p;
p = &x;
print p      // 300
print *p     // 5
```

~~print (p++)~~

```
p = p+1      // p increases by 4.
print (*p)   // Now, here we do not know what
                value this will give us.
```

However for integer array, we know the value of *(p+1) and so on.

A[0]	A[1]	A[2]	A[3]	A[4]
2	4	5	8	1

```
int A[5];
int *p;
p = &A[0].
print (p)     // 200
print (*p)    // 2
```

```
print (p+1) .     // 204
print (*(p+1))    // 4.
```

```
int A[5];
int* p;
p = A
print A      //200
print *A     // 2.
```

Now. here is a trick for arrays. just the name of an array is already a pointer for first element in C++.

```
int A[5];

print A     // 200   → prints address of first element of A.
print *A    // 2.
```

⟹ Element at index i

Address - &A[i] or (A+i)

value - A[i] or *(A+i).

Arrays as function arguments

```
int sum Of Elements (A[]) {
    int i, sum = 0;
    int size = sizeof(A) / sizeof(A[0]);
    print ( SOE, sizeof(A), sizeof(A[0]))        // 4, 4

    for (i=0; i<size ; i++){.
        sum += A[i].
    }.
    return sum
}.
```

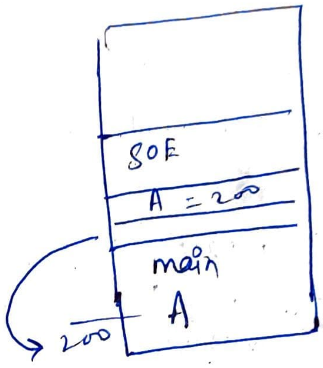
```
int main () {
    A[-] = {1,2,3,4,5};
    int total = Sum of Element (A)
    print ( sizeof (A), sizeof (A[0])      // 20, 4.
}
```

size of A in function is not 20 but 4

So, arrays are always called by reference in C++

This a pointer type.

```
Sum of Elements (int A[]) {
}.
```



This function is same as

Sum of Elements (int* A)

Thus, in C++, array is always given to a function along with it's size.

also A[i] would work inside the called function also.

A[i] is same as *(A+i). } dereferencing.

The values of array will change in main also because after all, we have just called it by reference.

Always give arrays like this

→ int Fun (int* A, int size).

Character arrays and pointers

string - group of characters.

eg. 'John' , "Hello world" , "I am felling lucky"

(i) How to store strings

size of array ≥ no of characters in a string + 1.

why the extra character ?

"JohN" size ≥ 5

char C[8];

0	1	2	3	4	5	6	7
J	O	H	N	\0	"	"	"

C[0] = J C[1] = O C[2] = H C[3] = N C[4] = \0 null characters

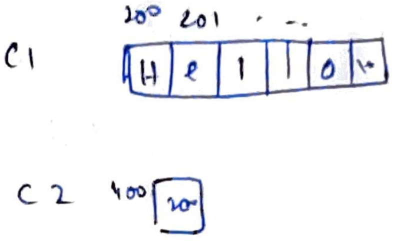
null character to indicate the termination of a string.

Rule: A string in C++ has to be null terminated.

(ii) Arrays and pointers are different types that are used in similar manner.

```
char C1[6] = "Hello"
char * C2;

C2 = C1 ;
```



→ This statement is valid.

```
print C2[1]    // e.
```

we can now use C2 similar to C1 and modify the position

```
C1 = C2;       X Not valid.
C1 = C1+1;     X Not valid.
C2++;          ✓ valid.
```

(iii) Array are always passed to function by reference.