character arrays and pointers

```
void print ( char* p) {

   while (*p != '\0')

     { printf ("%c", *p);

        p++;

     }

   printf ("\n");

}

int main () {

   char c[20] = "Hello";

   print (c)

 }
```

Here, p = c when the function call is made.

p[0] = H
p[1] = e.

output of this code will be.

Hello.

Remember that p stores the value of address of character string. that is the only important thing.

Also, if we want to only read from the array and not modify it., use. const.

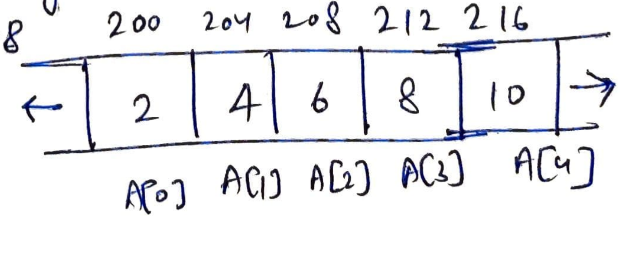
void (const char* p).

Pointers and multidimensional arrays

Firstly, we need to understand how a multidimensional array is stored in memory.

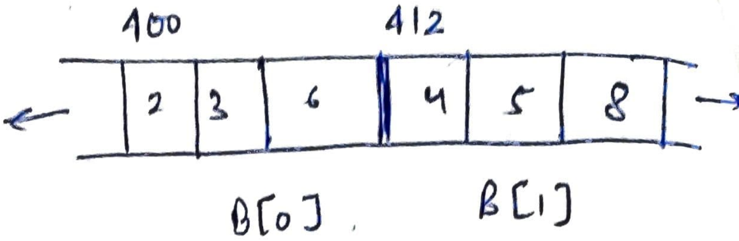
```
int A[5];
int * p = A;
print p // 200
print *p // 2
```

```
print (p+2) // 208
print *(p+2)
    // 6.
```



2 dimensional arrays.

Int B[2][3]B ;



B[0]
B[1] } → 1D arrays of 3 integers each.

int* P = B; X wrong statement because. B will return a pointer to a 1D array of 3 integers and not just an integer

The type of the pointer matters., So, we need to define the pointer as 1D array of 3 integers.

int (*P)[3] = B; ✓

print B or &B[0] ; // 400

print *B or B[0] ; // 400

Remember B[0] is also an array.

B stores the address of B[0].

print (B+1) . // 412. → Now B gives next 1D array of 3integers.
or
&B[1].

print *(B+1) // 412 → B+1 is a pointer to array B[1], Dereferencing gives value of B[1] i.e 412 print *(B+1) + 2 // 420

↳ Integer pointer to first element of B[1].

print ~~B~~ *(*B + 1) // 3.

↳ pointer to B[0] → itself an array pointer.

Thus, For 2D array.

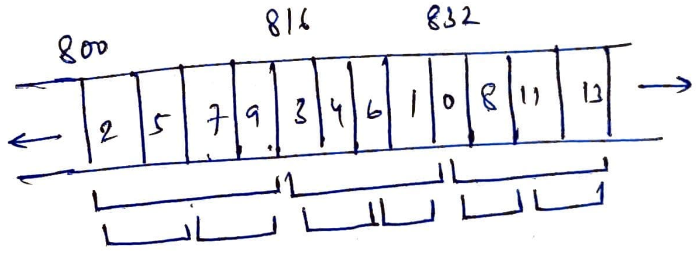
B[i][j] = *(B[i] + j)

= *(*(B+i)+j).

Pointers and multidimensional arrays

Consider previous array only.

int c[3][2][2];



Here similar to previous case

int (*P) [2] [2] = c;

print c // 800

print *c or c[0] // 800.

c[i][j][k] = *(c[i][j] + k) = *(*(c[i]+j) + k)

= *(*(*(c+i))+j) +k))

print *(c[0][1] + 1) // 9.

print *(c[1] +1) // .824.

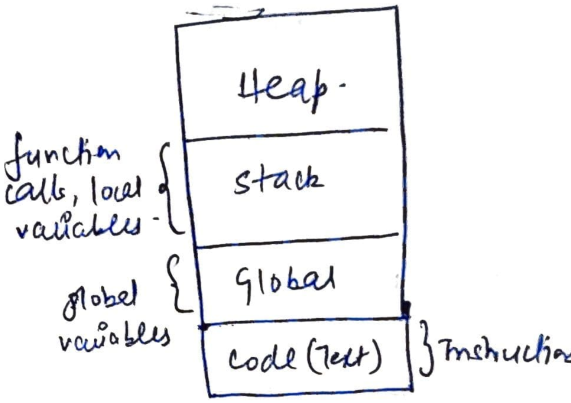
To pass let's say a 2 dimensional array in a function, the argument passed should be changed as.

```
int B[2][3];    // B returns int (*)[3]

void Func (int (*A)[3]) {
```

Pointers and Dynamic memory

Functions are called in a stack. one function over another.



stack overflow → memory is filled with function calls.

Stack allocated memory is fixed.

Heap memory is not fixed and we can keep increasing the memory required in program.

Heap → dynamic memory.
↳ No relation with heap data structure.

To use dynamic memory in C++, we need to know about.

C++ { malloc, calloc, realloc, free } functions

new, delete } operators