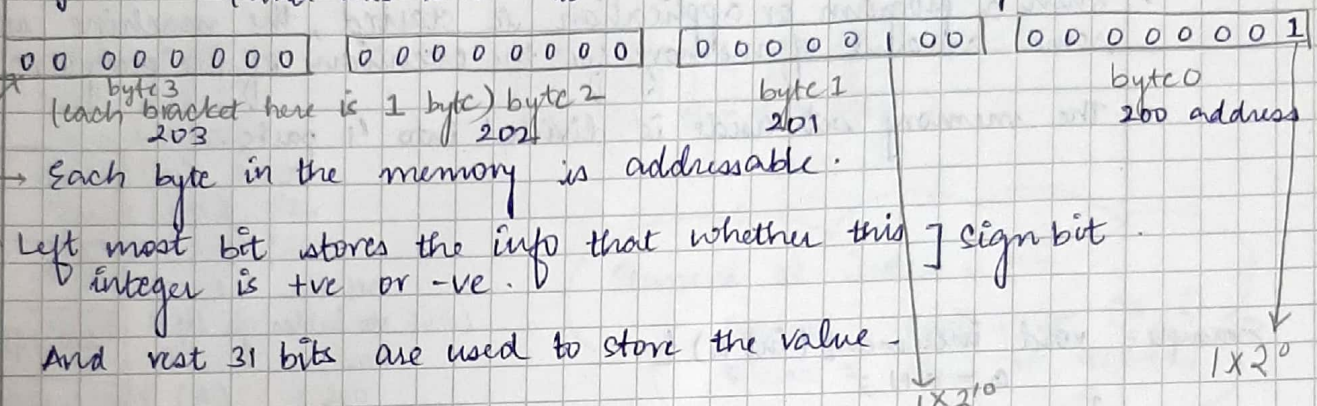
POINTERS

- We need to create pointer variable of a particular type to store the address of a particular type variable.
  eg., int * will be used to store address of int type variable.

  Why strong types? Why not some generic type? We don't use pointer variables just to store addresses but also to dereference these addresses and access/modify these addresses.

eg., int a = 1025
(and this is how it is laid out in memory)



(each bracket here is 1 byte)

→ Each byte in the memory is addressable.

Left most bit stores the info that whether this integer is +ve or -ve.] sign bit.

And rest 31 bits are used to store the value.

```
int *p
p = &a
Print p   // 200
Print *p  // Look at 4 bytes starting 200 // 1025
```

If p was a character type pointer, it would have looked into only 1 byte.

Example 1:

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

printf("size of integer %d bytes\n", sizeof(int));
printf("Address = %d, value = %d \n", p, *p);

char *p0 ;

p0 = (char*)p ;   // typecasting

printf("size of char %d bytes \n", sizeof(char));
printf("Address = %d, value = %d \n", p0, *p0)

print(p+1, *(p+1))   // 2948668, -85899

print(p0+1, *(p0+1))  // 2948665, 4
```

Size of integer 4 bytes
Address = 2948664, value = 1025
Size of char 1 bytes
Address = 2948664 value = 1

Why 1?

∵ When we write 1025 in binary using 32 bits.

Since p0 is of type character, it will only see 1 byte which is 0 0 0 0 0 0 0 1

In binary this is 1

VOID POINTER — GENERIC POINTER.

```
int a = 1025;
int *p;
p = &a;
print(p)
```

Address = 3341104

```
void *p0;
p0 = p;
print(p0, *p0);
```

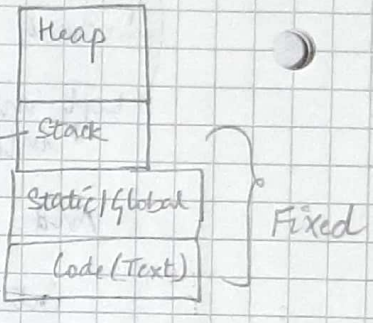
→ compile time error.

Address = 3341104

print(p0+1) → compile time error.

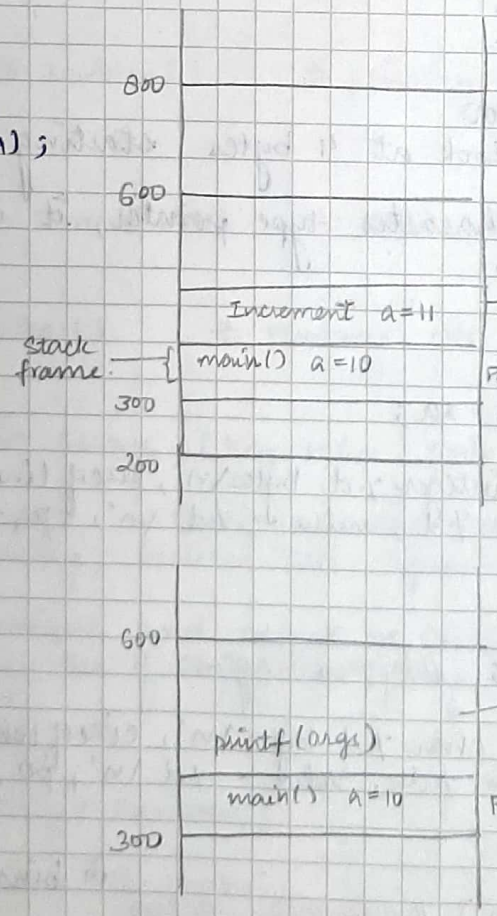
POINTERS AS FUNCTION ARGUMENTS

- When a program or application is started, the machine sets aside some amount of memory for the execution of the program.
- The memory set aside is divided into 4 parts.



Example:

```
void increment(int a){
    a = a+1;
}
int main(){
    int a = 10;
    increment(a);
    print(a);
}
```



Let's say machine has assigned memory for our program from 200 - 800.

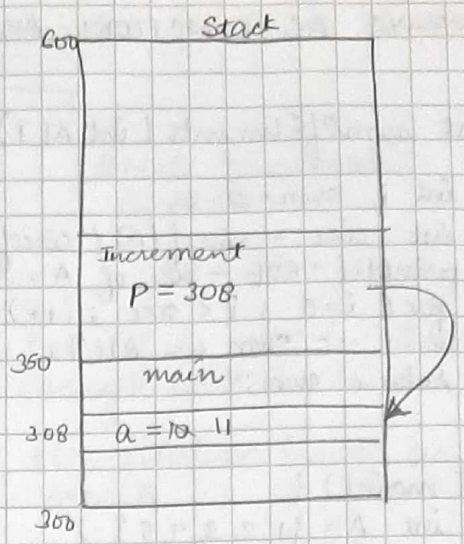
200 - 600 is for stack

∴ printf is a library function

NOTE: There's a definite size of stack frame. If you have a scenario in which one function keeps calling another function indefinitely → then memory of stack will overflow and program will crash.

Example :

```
void increment (int *p) {
    *p = (*p) + 1;
}
int main() {
    int a = 10;
    increment (&a);
    psintf("a = %d", a);
}
```

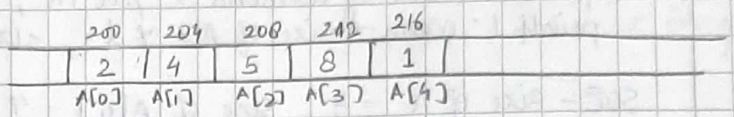


POINTERS AND ARRAYS

```
int A[5] ;
int *p ;
p = &A[0]
psint (p)  // 200
psint (*p)  // 2
print (p+1) // 204

int *p1 ;
p1 = A ;
psint (A) ;  // 200
psint (*A)  // 2
```

Base Address



Element at index i :-
Address : &A[i] or (A+i)
Value : A[i] or *(A+i)

NOTE :

```
int A[ ] = {2, 4, 5, 8, 1}
int i ;
int *p = A;
A++ ;  // Invalid
p++ ;  // Valid
```

Last address
Base address

ARRAYS AS FUNCTION ARGUMENTS

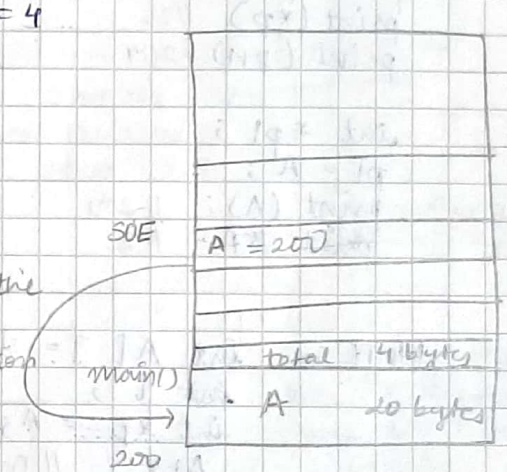
Example:
```
int sumOfElements(int A[]){        // compiler implicitly converts it to int *A
    int i, sum = 0;
    int size = sizeof(A) / sizeof(A[0]);
    printf("SOE - Size of A = %d, size of A[0] = %d\n", sizeof(A), sizeof(A[0]));
    for(i=0; i<size; i++)
        sum += A[i];
    return sum;
}

int main(){
    int A = {1,2,3,4,5};
    int total = SumofElements(A);
    printf("Sum of elements = %d\n", total);
    printf("Main - Size of A = %d, size of A[0] = %d\n", sizeof(A), sizeof(A[0]));
```

SOE - Size of A = 4, size of A[0] = 4
Sum of elements = 1
Main - Size of A = 20, size of A[0] = 4

To solve this issue

A copies the address of first element in the array of calling function




```
int main(){
    int A = {1,2,3,4,6};
    int total = 5;
    int size = sizeof(A) / sizeof(A[0]);
    sumofElements(A, size);
}
```

→ Call by reference.
The changes that you do in SOE will be reflected in main()

CHARACTER ARRAYS AND POINTERS

Example:
```
int main(){
    char c[4];
    c[0] = 'J';
    c[1] = 'O';
    c[2] = 'H';
    c[3] = 'N';
    printf("%s", c);
```

Output :- JOHN ╞╞╞D√2

char c[5] or 6 or 7 or anything ≥5

→ If I add c[4] = '\0'
O/P : JOHN

NOTE: strlen() counts the characters until it finds a null character.

String Literal:

```
char c[20] = "JOHN"   // Null termination here is implicit.
```

NOTE:

```
char c[20];
c = "JOHN"   // invalid

char c[] = "JOHN"   // size of c will be 5 bytes.
                    // Length will be 4.

char c[4] = "JOHN"   // Compilation error as it at least expects 5.

char c[5] = {'J', 'O', 'H', 'N', '\0'};
```

∴ null termination is not implicit here

Example:

```
char c1[6] = "Hello";
char *c2;
c2 = c1;          // c1 = c2 is invalid
print(c2[1]);     // e
c2[0] = 'A';      // Aello
```

NOTE: c2[i] is *(c2+i) ✓✓ c1[i] is *(c1+i) ✓✓

c1++; ✗ c2++; ✓✓

- Arrays are always passed to function by reference.

Example:

```
void print(char *c) {
    int i = 0;
    while(c[i] != '\0') {
        printf("%c", c[i]);
        i++;
    }
    printf("\n");
}

int main() {
    char c[20] = "Hello";
    print(c);
}
```

c[i] → *(c+i) OR *c

c[i] → *(c+i) OR *c

] we'll not require 'i' variable for second one.

NOTE: The 'c' in main() and 'c' in print() are different. They have different scopes.

↳ base address was been passed.

Example:

```
char *c = "Hello";   // string gets stored as compile time constant
c[0] = 'A';
```

Example:

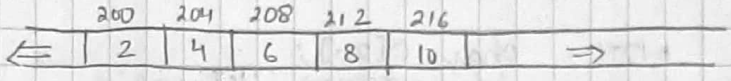
```
void print(const char *c) {   // we want the function only to read the string
    c[0] = 'A';               // → Compilation error
    while(*c != '\0') {
```

⊛ POINTERS & MULTI-DIMENSIONAL ARRAYS

⊛ One-Dimension :-

```
int A[5];

int *p = A;
psint p   //200
psint *p  //2
psint *(p+2) //6
```



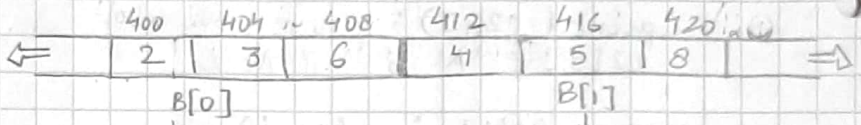

```
psint A      //200
psint *A     //2
psint *(A+2) //6
```

*(A+i) is same as A[i]
(A+i) is same as &A[i]

P = A ✓
A = P X

⊛ Two-Dimension :-

```
int B[2][3];
```




```
int *p = B;  X
```
→ will return a pointer to 1D array of 3 integers

```
int (*p)[3] = B;
```
That is why this

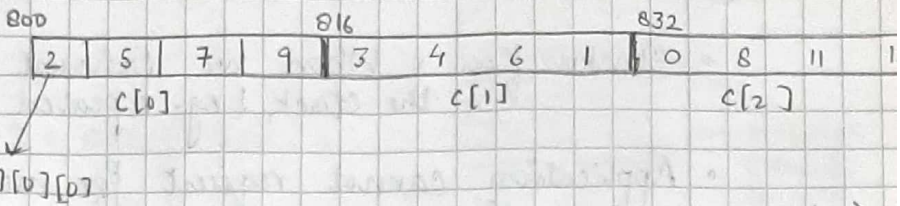
- psint B or &B[0] //400
- psint *B or B[0] or &B[0][0] //400
- psint B+1 //400 + size of 1D array of 3 integers (in bytes)
  //400 + 12 = 412
- psint *(B+1) or B[1] or &B[1][0] //412
- psint *(B+1) + 2 // 412 + (2*4) = 412 + 8 = 420
  (*(B+1) : int *)
- psint *(*B+1)
  (*B : B[0])
  //400 → 400 + (1*4) = 404 &B[0][1]
  *(404) = 3

For 2D array :-

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

Three Dimension:

int C[3][2][2]



int (*p)[2][2] = C; ⟶ integer pointer to a 2D array of (2×2 size)

print C 11800

print *C or C[0] or &C[0][0] 11800

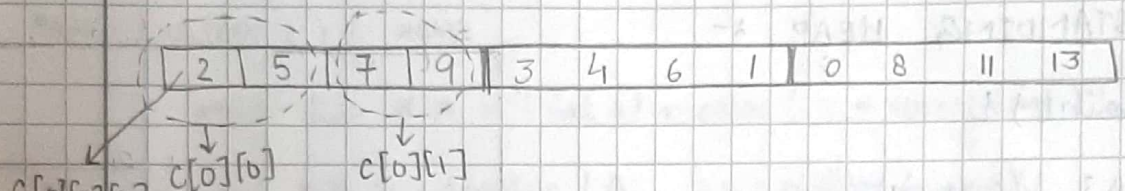
C[i][j][k] = *(C[i][j] + k) = *(*(C[i] + j) + k)

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

first integer of [7|9]

Print *(C[0][1] + 1) or C[0][1][1] 119

print *(C[1] + 1) or C[1][1] or &C[1][1][0] 11824



print C. } 11800

print *C

print C[0]

print &C[0][0]

print (C[0][0] + 1) 115

Passing multi-dimensional array as a function argument:

~~int~~

void func(int *A){

}

int main(){

int A = {1, 2}; // A returns int*

int B[2][3] = {{2,4,6}, {5,7,8}}; // B returns int(*)[3]

func(A);

}

void funct(int (*A)[3])

OR

int A[][3]

⊛ POINTERS AND DYNAMIC MEMORY

- Stackoverflow : When we exhaust the whole space reserved for the stack (eg. repeated calls during recursion)
- Application cannot request for more memory from stack during run time.
- Heap size can vary during the run time of application.
  - ↳ free store
  - Not related to data structure at all.

C	C++
malloc	new
calloc	delete
realloc	
free	

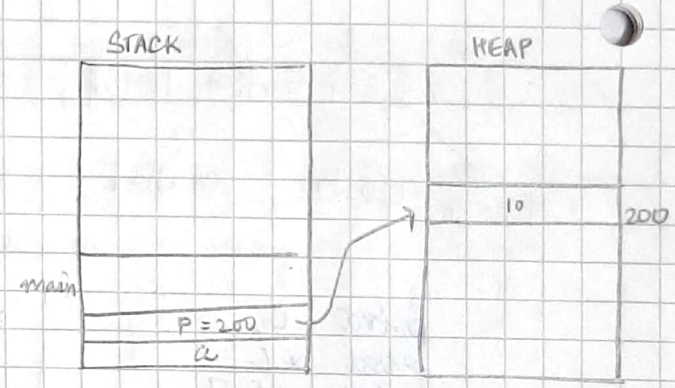
← for dynamic memory allocation

UNDERSTANDING HEAP :-

```
int main() {
    int a;  // goes on stack
    int *p;
    p = (int *) malloc(sizeof(int));   // (returns an address)
    *p = 10  // will store 10 in heap.
```

↓

The only way to use memory on heap is through reference.

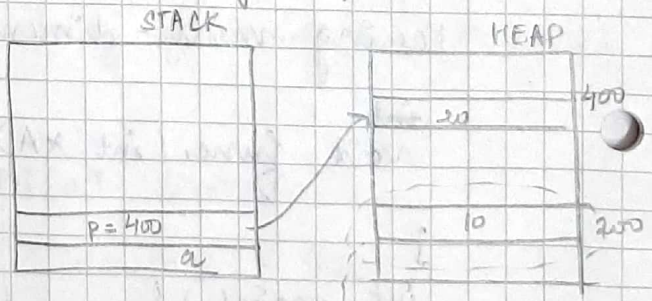



```
p = (int *) malloc (sizeof(int));
*p = 20
```

OR

```
p = new int[20];
delete [] p; // for array

p = new int;
delete p;  // for normal stuff
```



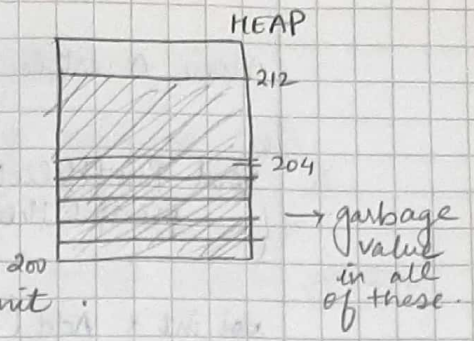
This needs to be cleared.
free(p)

⊛ MALLOC, CALLOC, REALLOC, FREE

- malloc():

```
void * p = malloc( 3 * sizeof(int) );
```

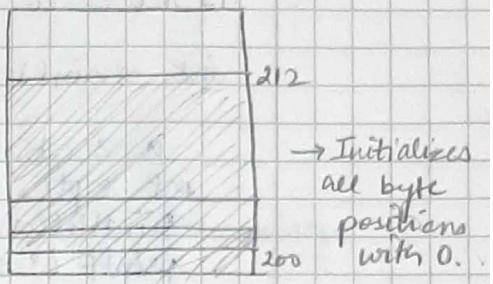
(3 → no. of elements; sizeof(int) (% 4) → size of one unit.)



- calloc():

```
void *p = calloc (3, sizeof(int))
```

(3 → no. of elements; sizeof(int) → size of one unit)



- realloc():

```
~~void~~ int * A = (int *) malloc ( 3 * sizeof (int) );
int * B = realloc (A , 2 * 3 * sizeof (int));
```

If the neighbouring area near to A is empty, block will be extended else a new block is allocated and content from the previous block is copied and previous block is deallocated.

- Write free(A) using ~~malloc~~ realloc()

```
int *A = (int*) realloc (A, 0);
```

- Write malloc using realloc()

```
int *B = (int *) realloc (NULL, 3* sizeof (int));
```

⊛ POINTERS AS FUNCTION RETURN:

BEFORE

```
int Add ( int *a , int * b){
    int c = (*a) + (*b);
    return c;
}
```

OR

AFTER

```
int * Add ( int *a, int * b) {
    int * c = (*a) + (*b);
    return &c;
}
```

return address of 'c'

```
int main() {
    int a = 2, b = 4;
    int c = Add (&a, &b);
    printf(c);
}
```

```
int main() {
    a = 2, b = 4
    int * ptr = Add (&a, &b);
    print (*ptr);
}
```

now, a little twist,

```
void PrintHelloWorld() {
    printf("Hello, World");
}

~~voi~~ int * Add(int *a, int *b){

    int c = (*a) + (*b);
    return &c;
}

int main() {

    a=2, b=4
    int *ptr = Add(&a, &b);
    PrintHelloWorld();
    printf("sum = %d\n", *ptr);
```

→ Suppose ptr holds the address of a block in which (*a) + (*b) occured in heap.

→ Now, when this function is called, the block for Add() got erased from memory and PrintHelloWorld came into its place.

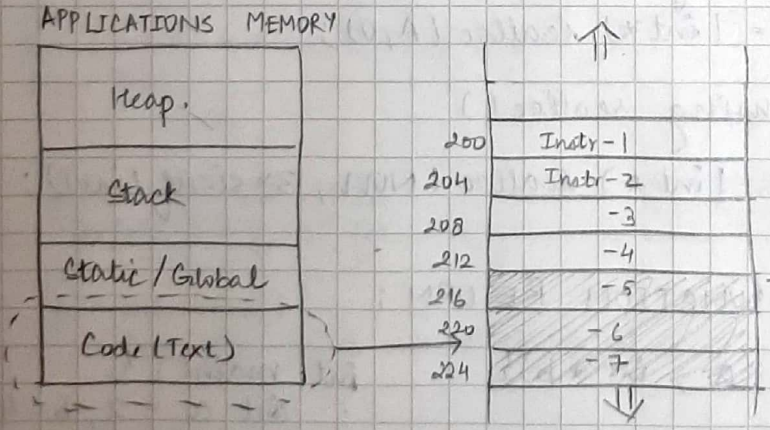
Hence, we got a garbage value.

o/p: Hello, World
02576321 — ???

Lesson Learnt: It is not ok to return the address of local variable from top to bottom in a call stack.

⊛ FUNCTION POINTERS

↳ store ~~or~~ address of functions.
(address of first instruction)



Example:
```
int Add(int a, int b){
    return a+b
}

int main(){
    int c;
    int (*p)(int, int);
    p = &Add;
    c = (*p)(2,3)  // de-referencing and executing the function.
}
```

- Did you notice the syntax for function pointer.

  `int (*p)(int, int);` ✓

  `int *p(int, int)` ✗

  ⇓ declaring a function "p" which would return pointer to integer.

- NOTE: "`p = &Add`" is same as "`p = Add`"

  "`c = (*p)(2,3)`" is same as "`c = p(2,3)`"

⊛ FUNCTION POINTERS & CALLBACKS (USE CASES OF F.P.)

Suppose we wish to sort an array of integers in both increasing and decreasing order.

```
int compare (int a, int b) {
    if (a > b) return -1;
    return 1;
}
```

for decreasing

For increasing order, return 1 when (a>b)

```
void BubbleSort (int A[], int n, int (*compare)(int, int)) {
    int i, j, temp;
    for (i=0; i<n; i++)
        for (j=0; j<i; j++) {
            if (compare(A[j], A[j+1]) > 0) {
                temp = A[i];
                A[i] = A[j];
                A[j] = temp;
            }
        }
}

int main() {
    int i, A[] = {2, 4, 3, 7, 5, 6};
    BubbleSort(A, 6, compare);
    for (i=0; i<6; i++)
        printf("%d ", A[i]);
}
```

Now, we wish to include negative nos. as well in our array of integers,

```
int compare (const void *a, const void *b) {
    int A = *((int *)a); // typecasting to int*
    int B = *((int *)b);
    return A-B; // for increasing order  For decreasing order = B-A
}
```

```
int main(){
    int i, A[] = {-3, 22, -1, 50, -6, 4};
    qsort(A, 6, sizeof(int), compare);
    for (i=0; i<6; i++) printf("%d", A[i]);
}
```

⑧ MEMORY LEAK in C/C++

↓

- When we get some memory in heap but do not free it when we are done using it.
- Our application is holding onto some unused memory on heap.
- Happens only through heap.
- Java and C# has garbage collector ⇒ Implies avoid memory leak.