ATTRIBUTES OF A VARIABLE

- ★ name
- ★ datatype
- ★ value
- ★ address

} needed when drawing/ visually representing execution stack

- ★ lifetime & scope

"What is a variable?"

A place-holder for a value, this placeholder value is allocated in the computer's memory. i.e a variable is a location in memory.

The programmer doesnt have control over the address in which the variable is stored, but can control everything else.

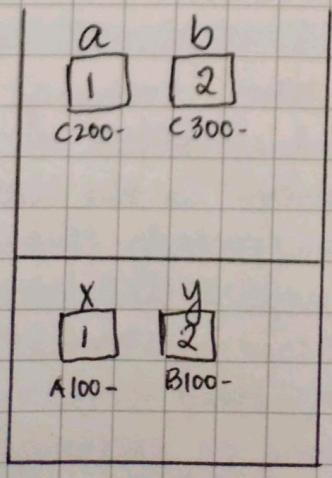
Lifetime & scope → area or region of code where a variable is available to use. i.e areas or sections of the program which the variable can be accessed.

→ how long the variable stays alive in memory i.e lifetime defines the duration for which the computer allocates memory for it.

HOW TO DRAW AN EXECUTION STACK

To draw an execution stack, draw a box for each variable & label with its address. Since its an execution stack, draw it in such a way where first item in is first item out.

Eg. Execution Stack of a function call Myfunction()



Eg.

Q. PROBLEM SPECIFICATION:

The function will exchange the values of 2 given integers.

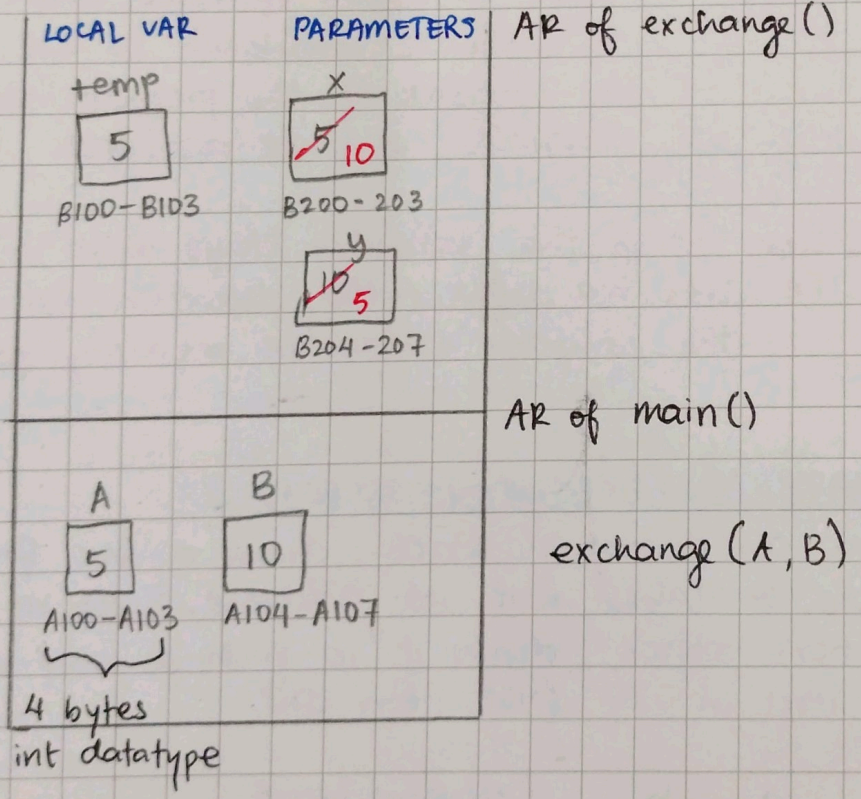
```
void exchange ( int x, int y) {
int temp;
temp = x;
x = y;
y = temp;
}
```

FUNCTION CALL:

```
int A = 5;
int B = 10;
exchange (A, B);
```

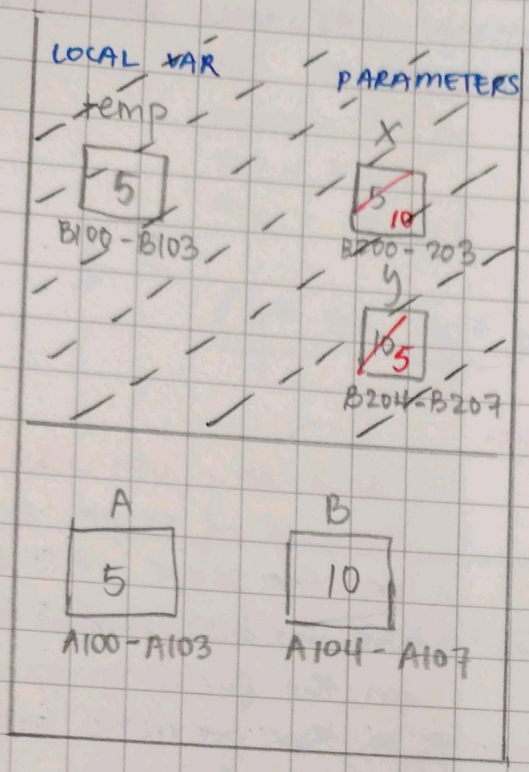
USING THE EXECUTION STACK, SHOW THAT THE FUNCTION WILL NOT WORK BASED ON THE FUNCTION SPECS. EXPLAIN.

ANS.



The function would not work. When exchange() terminates, so does temp, x and y. The values stored in A and B in main were passed by copy only. So the variables A and B in main() were unaffected. This is an example of the effects of scope & lifetime.
Scope of A & B is main(). Scope of temp, x and y is exchange(). As for lifetime of temp, x and y - they remain alive in memory as long as exchange() is not terminated yet.

THIS IS WHAT HAPPENS WHEN exchange() TERMINATES



→ exchange() terminates. temp, x and y no longer exist

→ A and B in main are unchanged.

TO FIX THIS, PASS BY ADDRESS INSTEAD.

★Note: Pass by copy when searching or printing, usually with isMember() or display().
Pass by address when inserting or deleting, when the change created by the function needs to be reflected in main().

TURNING AN EXECUTION STACK INTO CODE AKA HOW TO CODE FUNCTIONS

1. FUNCTION HEADER
2. FUNCTION CALL with the variables used in the call declared and initialized.
3. SHOW THE EXECUTION STACK DEMONSTRATING THE FUNCTION CALL
4. CODE OF THE FUNCTION.

E.g

Q. WRITE THE CODE OF THE FUNCTION THAT WILL EXCHANGE THE VALUES OF 2 GIVEN INTEGERS.

ANS. USING THE STEPS.

① FUNCTION HEADER

```
void exchange(int *x, int *y)
```

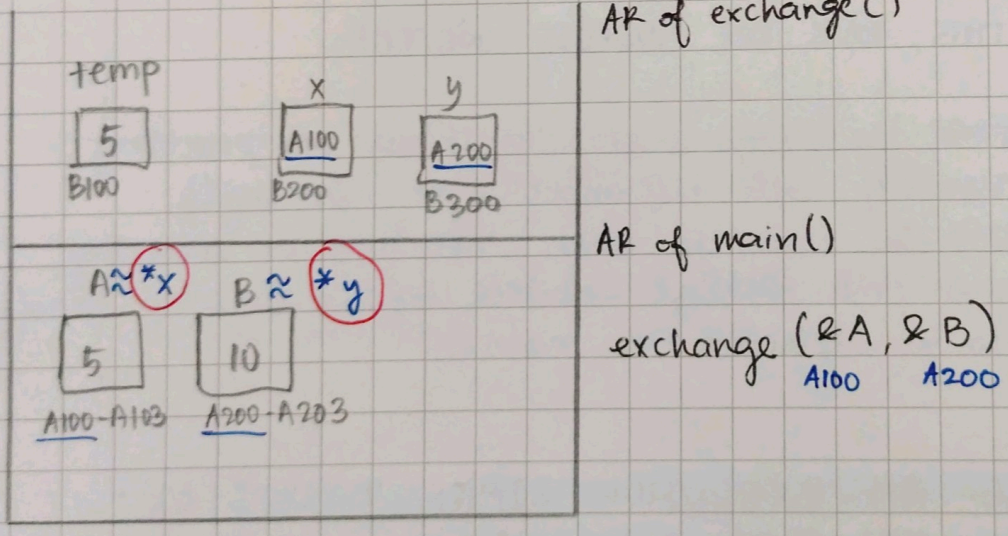
② FUNCTION CALL

```
int A = 5;
int B = 10;
exchange(&A, &B);
```

passing addresses

Why use address?
Because of scope. Based on the scope, A & B are not accessible based on the function exchange() in the previous example.

③ DRAW THE EXECUTION STACK DEMONSTRATING THE FUNCTION CALL

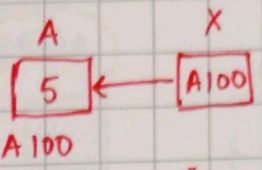


*x is an alias for A. x = &A, x holds the address of A.
*y is an alias for B. y = &B, y holds the address of B.
This way, A and B in main are made accessible by exchange().

④ CODE OF THE FUNCTION

```
void exchange (int *x, int *y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

→ dereferences x
x = &A



therefore *x = 5 (which is the value inside A)

LINKED LISTS & EXECUTION STACK

★ Note: Memory that is dynamically allocated is allocated to the heap portion of the computer's memory. As such, draw the execution stack accordingly.

Eg.
Q. PROBLEM SPECIFICATION: THE FUNCTION WILL INSERT AN ELEMENT TO THE GIVEN LIST AT THE FIRST POSITION OF THE GIVEN LIST.

```
typedef struct node {
    int data;
    struct node * link;
} * LIST;
```

ANS.

① FUNCTION HEADER

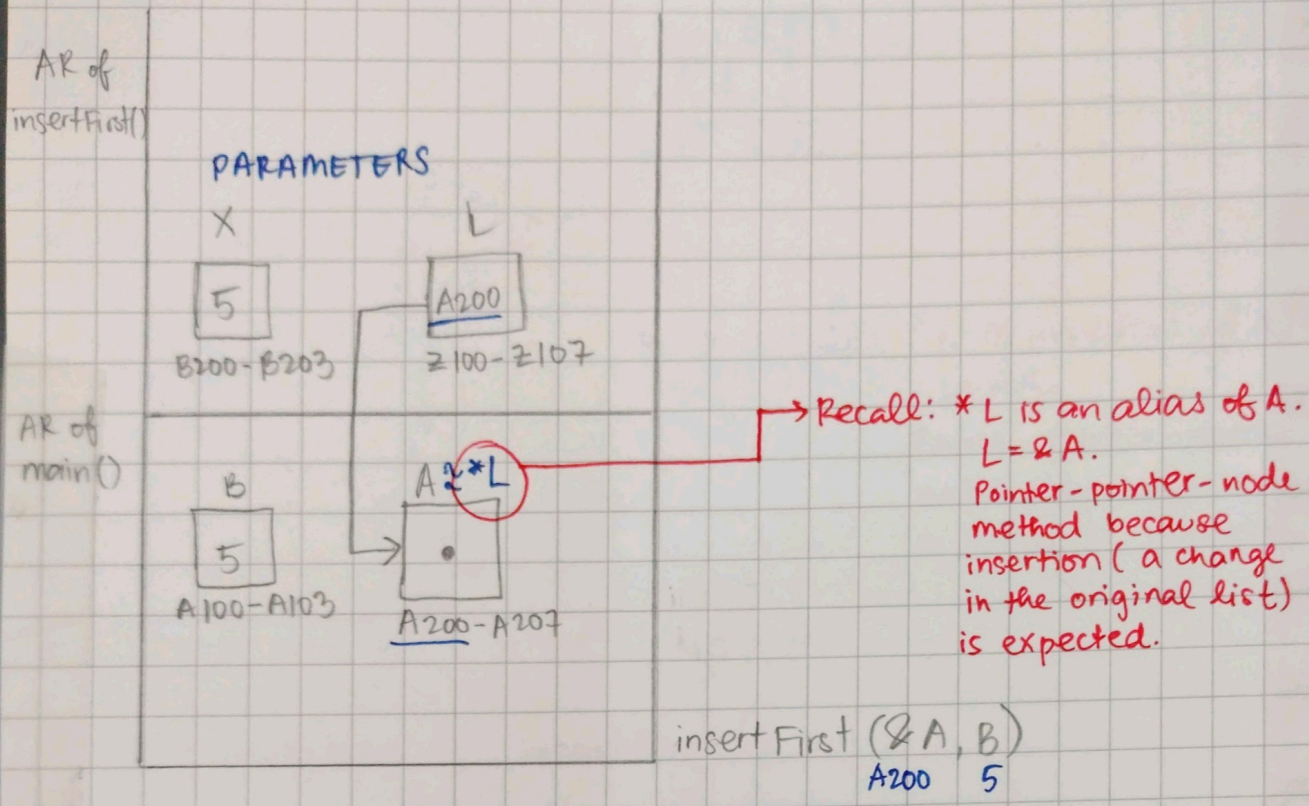
→ passing by address, pointer-to-pointer-to-node method

```
void insertFirst (LIST *L, int x)
```

② FUNCTION CALL

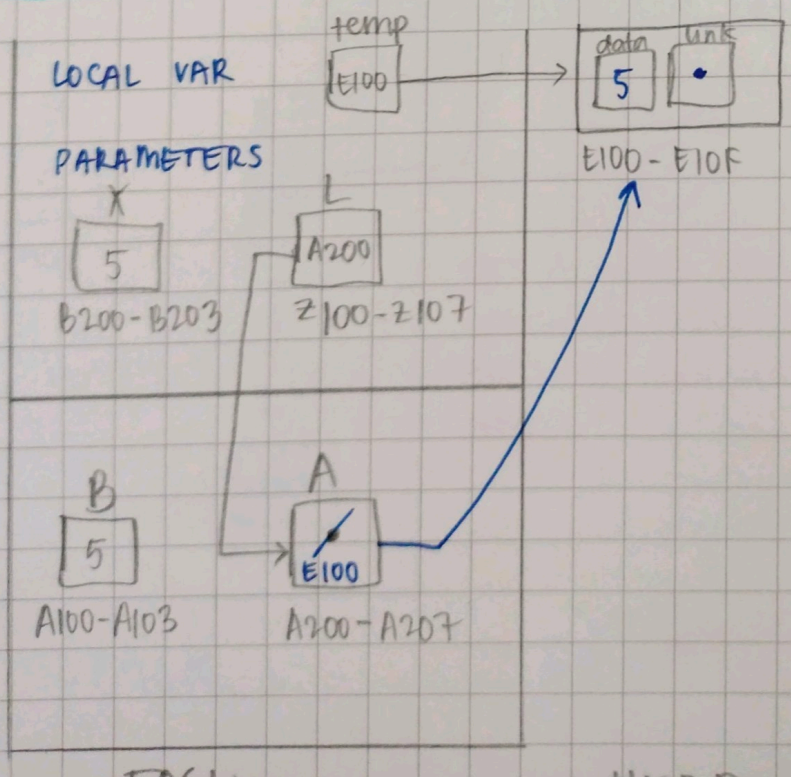
```
LIST A = NULL;   ← initialize NULL because upon declaration, A is garbage
int x;
insertFirst(&A, x);
```

③ Execution Stack



This is the execution stack representing the function call when list is NULL.

Simulation of insertFirst() with NULL list. This serves as test case 1.

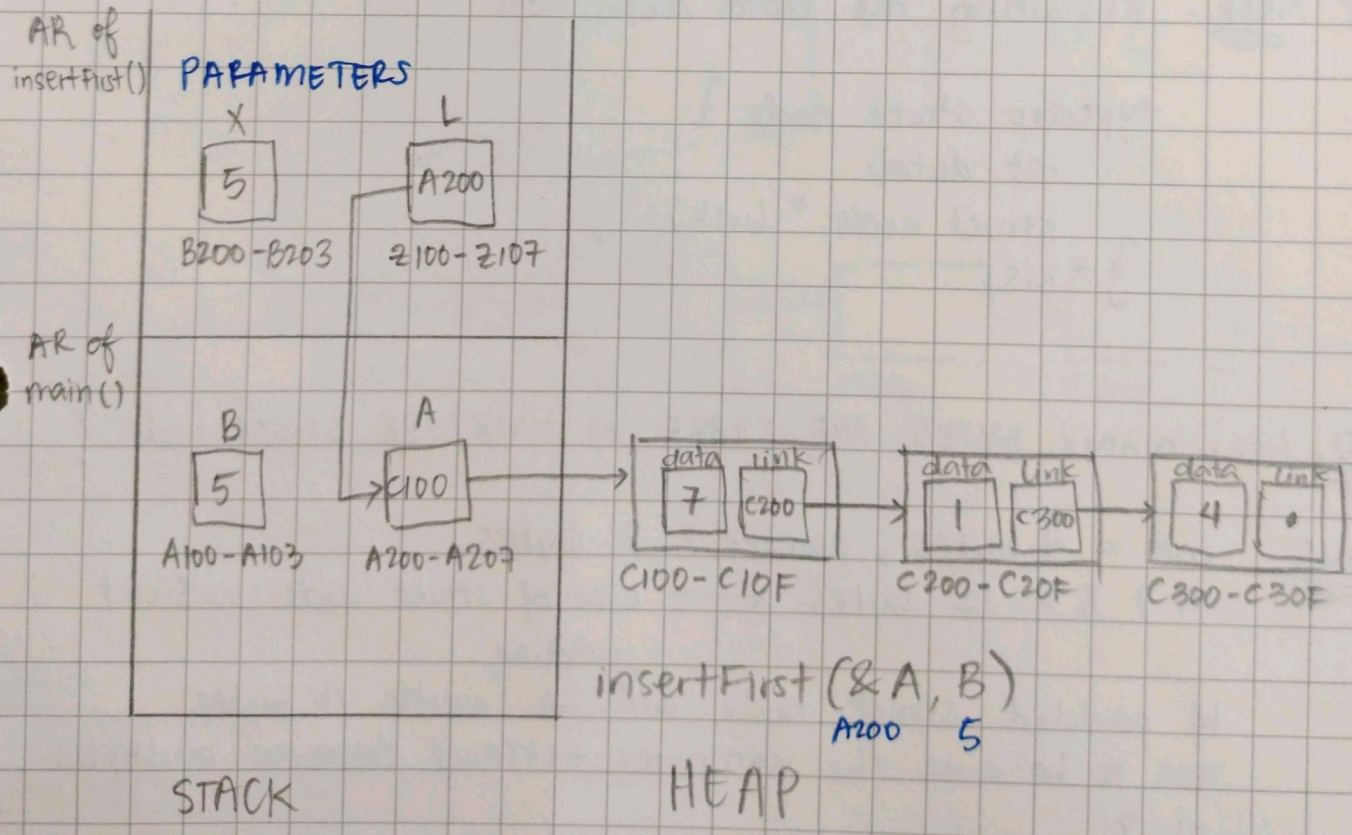


④ CODE OF THE FUNCTION

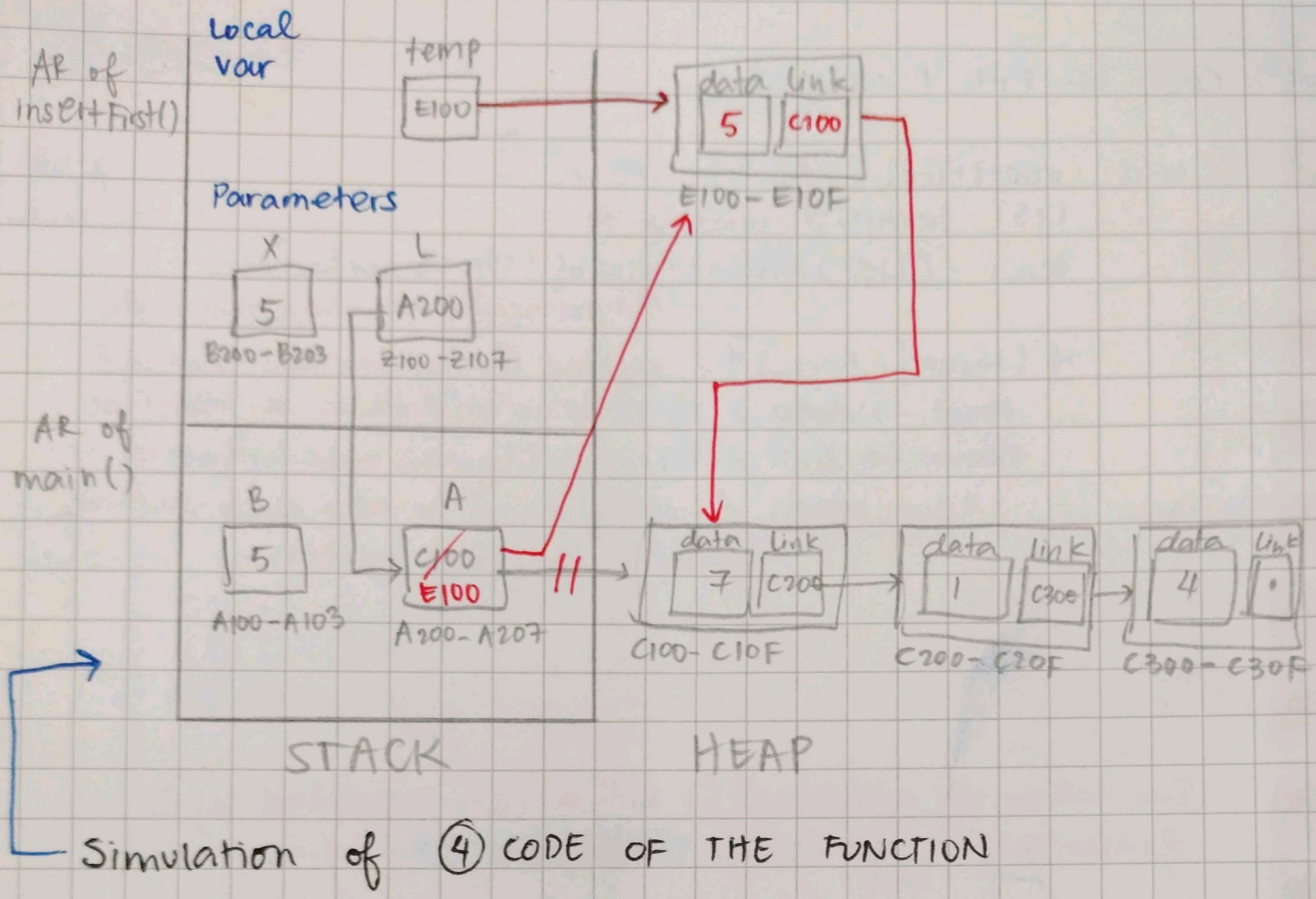
```
void insertFirst (LIST * L, int x) {
    LIST temp; // declaring temp
    temp = (LIST) malloc (sizeof (struct node));
                    // dynamically allocating new node
    if (temp != NULL) { // checking if allocation was successful
        temp → data = x; // assigning data to be inserted
        temp → link = * L; // Linking new node to the next one
        * L = temp; // linking head to new node so that
                    it is in the first position.
    }
}
```

Simulation of insertFirst() will a list populated by 3 elements. This serves as test case 2.

Drawing of the execution stack representing the function call of insertFirst() when list is populated.



★ Note: Variables allocated with malloc() or calloc() are allocated in the heap, not the stack.



Simulation of ④ CODE OF THE FUNCTION

★ Note: Revisiting the data definition

```
typedef struct node {
    int data;
    struct node * link;
} *LIST
```

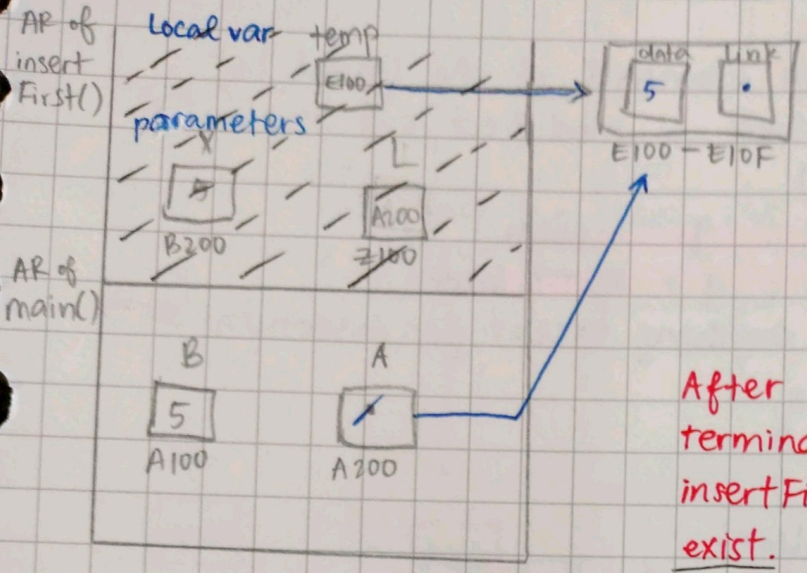
Q. HOW MANY BYTES ARE THERE IN DATATYPE struct node ?

ANS. int = 4 bytes, pointer = 8 bytes.
4 + 8 = 12 bytes ← size of struct node without padding.

If padded, struct node will be worth 16 bytes.
This is because the computer extracts data in multiples of 8.

IN BOTH TEST CASES...

TEST CASE 1 : NULL (EMPTY) LIST

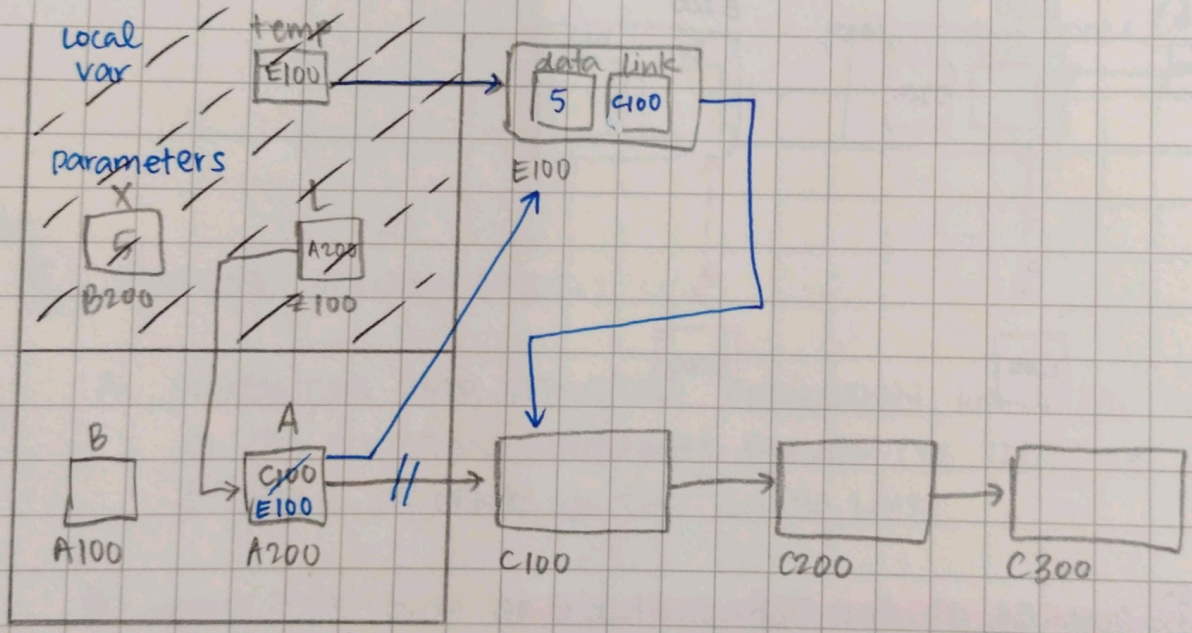


Lines depict function termination

After function insertFirst() terminates, the variables in insertFirst() will no longer exist.

→ *Recall "lifetime"

TEST CASE 2 : POPULATED LIST



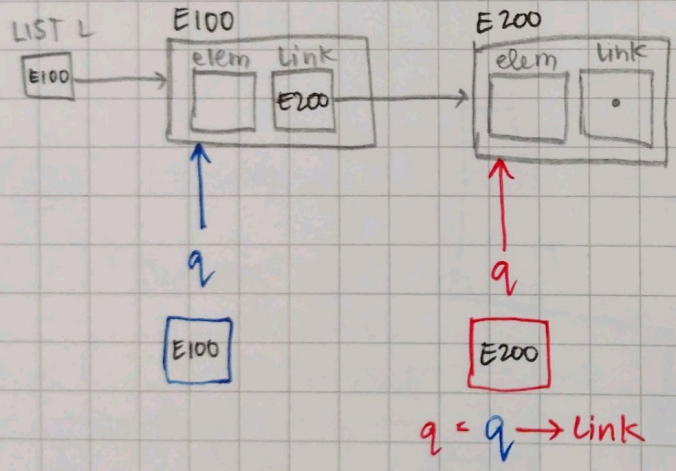
LINKED LIST TRAVERSAL

2 TYPES OF LIST TRAVERSALS :

① POINTER TO NODE

- ★ used when elements of the linked list are retrieved or modified e.g display() or sort()
- ★ Accessing : q → elem
- ★ Traversal : q = q → link
- ★ used with anything except for insertion & deletion

```
typedef struct node {
        int elem;
        struct node * link;
} *LIST;
```

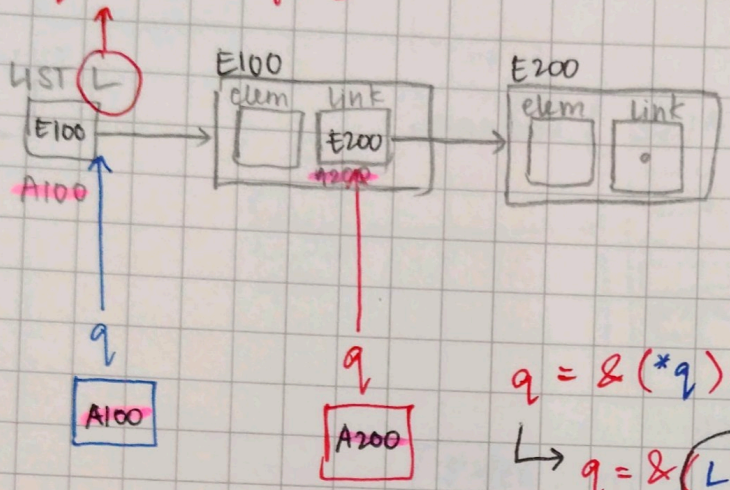


② POINTER TO POINTER TO NODE

- ★ used when inserting or deleting i.e when a change is expected in the original list
- ★ Accessing : (*q) → elem
- ★ Traversal : q = &(*q) → link
- ★ in the case of inserting or deleting the first element, pointer-node can be used.

```
typedef struct node {
    int elem;
    struct node * link;
} * LIST;
```

*q is an alias for L



q = &(*q) → link

↳ q = &(L) → link

Arrow (→) takes precedence over &

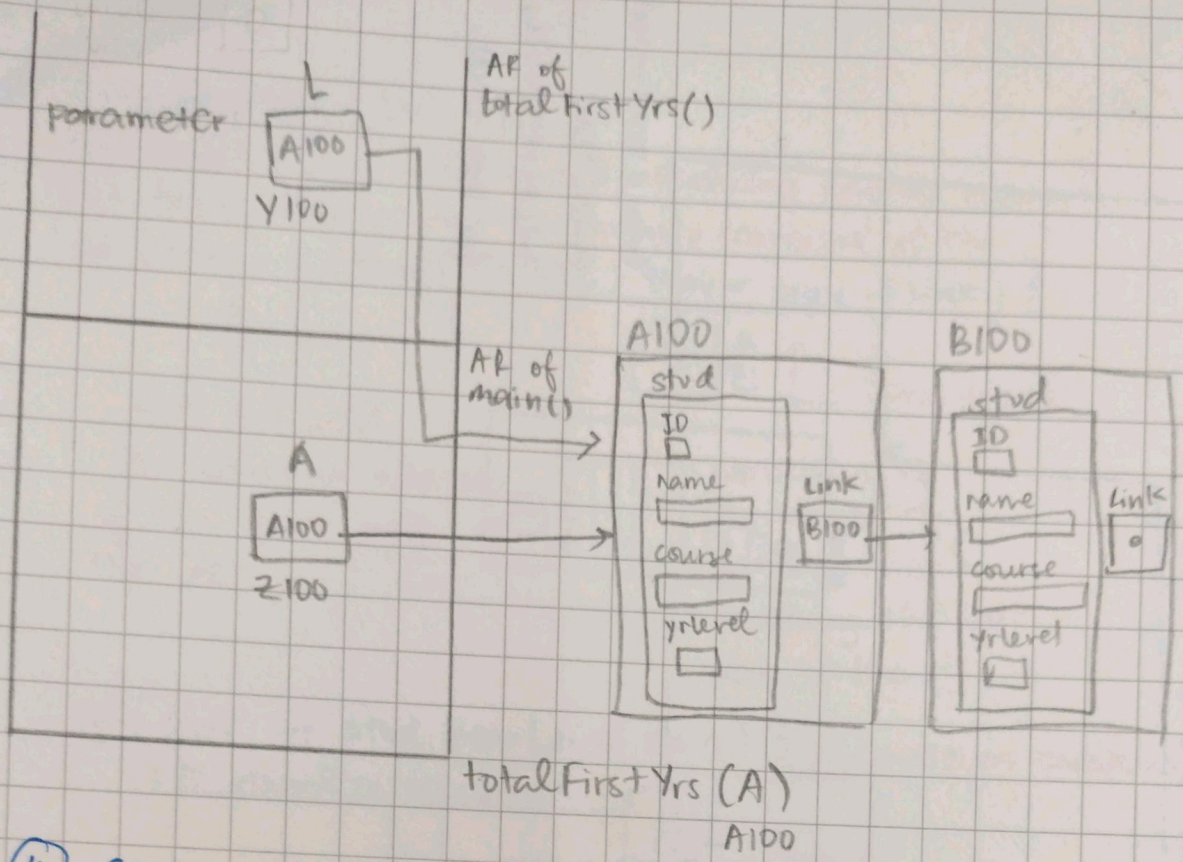
Eg.

Q. PROBLEM SPECIFICATION:

A. GIVEN THE DATA STRUCTURE DEFINITION, WRITE THE CODE OF THE FUNCTION THAT WILL RETURN THE TOTAL NO. OF 1st YEAR STUDENTS IN THE GIVEN LIST.

B. WRITE THE CODE OF THE FUNCTION THAT WILL RETURN TRUE IF THE RECORD BEARING THE GIVEN ID IS IN THE GIVEN LIST, OTHERWISE RETURN FALSE.

NEXT PAGE ⇒

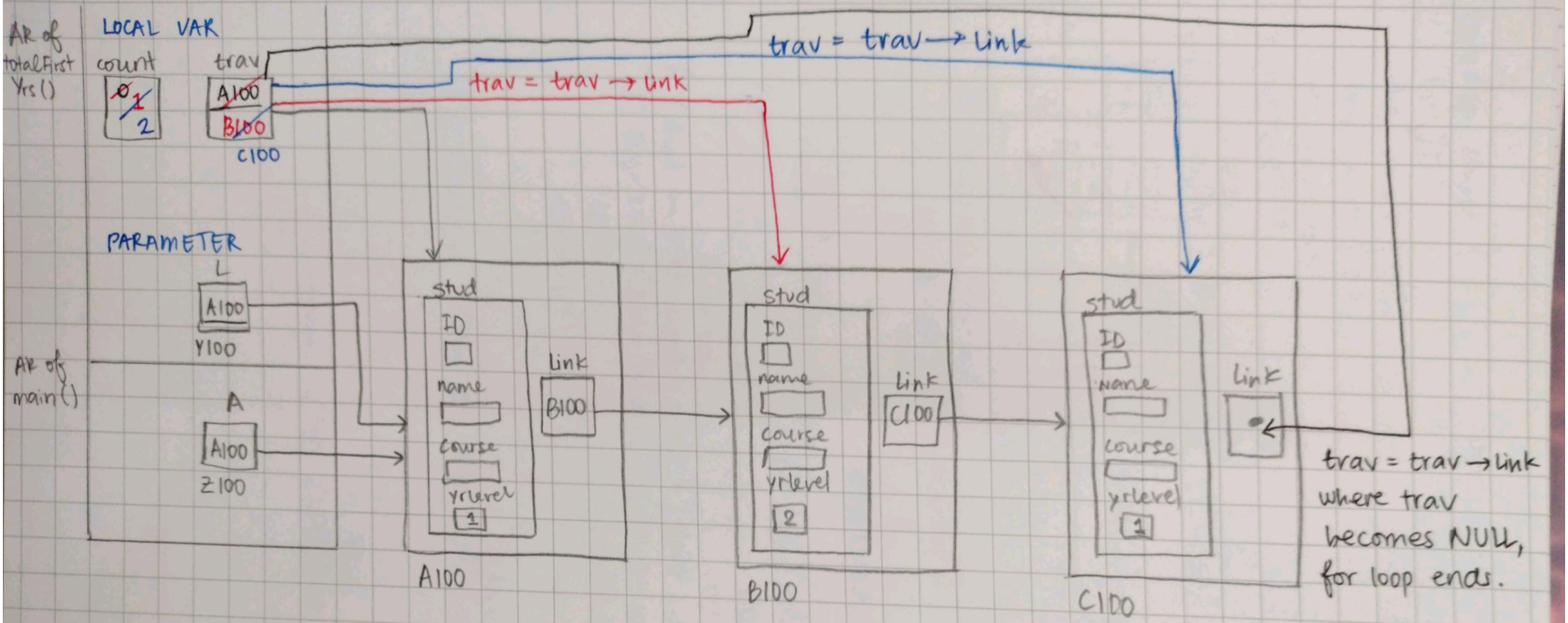


④ CODE OF THE FUNCTION

```
int totalFirstYrs (List L) {
    int count = 0;
    List trav;
    for (trav = L; trav != NULL; trav = trav → link) {
        if (trav → stud.yearlevel == 1) {
            count++;
        }
    }
    return count;
}
```

traversal

Simulation of code
ON NEXT PAGE (ROTATE 90°) ⟶


AR of totalFirstYrs()
LOCAL VAR
count
0 1 2
trav
A100
B100
C100
trav = trav → link
trav = trav → link
PARAMETER
L
A100
Y100
AR of main()
A
A100
Z100
stud
ID
name
course
yrlevel
1
link
B100
A100
stud
ID
name
course
yrlevel
2
link
C100
B100
stud
ID
name
course
yrlevel
1
link
C100
trav = trav → link
where trav
becomes NULL,
for loop ends.

```
int totalFirstYrs (List L) {
    int count = 0; // declaring and initializing counter variable to return
    List trav; // declaring trav to help traversal of the List
    for (trav = L; trav != NULL; trav = trav -> link) {
```

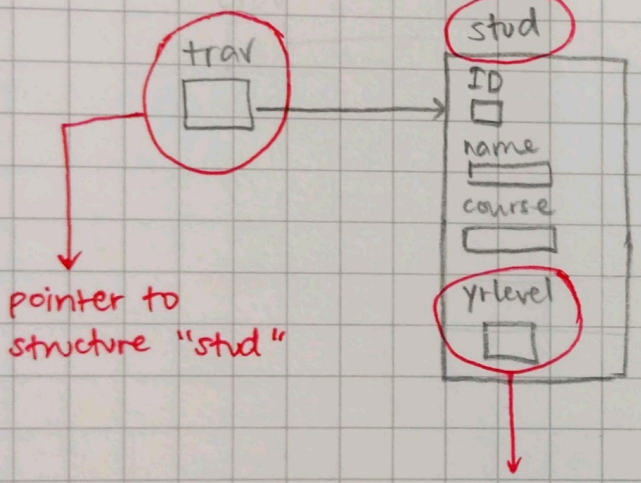
trav will hold the starting address of the first node of the list

condition to terminate the for loop when reaching the last node of the list

traversal

```
    if (trav -> stud.yearlevel == 1) { // condition checking
```

(to visualize accessing...)



to access member of a structure with a pointer, use (→)

If member is inside the structure and no pointer is used, simply use (.)

```
        count++; // increment count if condition is TRUE
    }
}

return count; // return statement, returns int variable that holds the number of 1st yr. students.
}
```

★ Note: Scalar vs. aggregate structures:

SCALAR → int, float, char

AGGREGATE → structures

Traversal → to begin with the first element and to travel through the list.

ARRAYS, ARRAY TRAVERSAL & POINTER ARITHMETIC

ARRAY DECLARATION:

dataType arrayName [arraySize];

e.g float mark [5];

★ Note: size and type of an array cannot be changed once it is declared

ARRAY INITIALIZATION:

int mark [5] = {19, 10, 8};

0	1	2	3	4
19	10	8	0	0

int mark [] = {19, 10, 8};

0	1	2
19	10	8

The name of the array is the address of the first component i.e pointer to the first component.
Use pointer arithmetic as another way to traverse through an array.

E.g int arr [5];

0	5	A100 - A103
1	7	A104 - A107
2	1	A108 - A10B
3	3	A10C - A10F
4	8	A110 - A103

increments by 4 bytes, the size of 1 int.

arr = &arr[0]

arr + 1 = &arr[0] + (1) ⇒ &arr[1]

*(arr) = 5

*(arr + 1) = 7

*(arr + 2) = 1

*(arr + 3) = 3

(*)(arr + 4) = 8

dereferences &arr[0] + N where N is the number of increments.
accesses the value stored inside &arr[0] + N.

Eg.

Q. PROBLEM SPECIFICATION:
GIVEN THE ARRAY OF INTEGERS, THE SIZE OF THE ARRAY AND AN ELEMENT X, THE FUNCTION WILL RETURN TRUE IF X IS IN THE ARRAY, OTHERWISE RETURN FALSE.
typedef enum {TRUE, FALSE} boolean;

ANS.

① FUNCTION HEADER

boolean isMember (int arr[], int size, int elem)
OR int *arr

★ Note: Even though pointer arithmetic is syntactically correct, using [] is the more preferrable coding convention.

3 parameters, which is what the question stated. Be sure to read the problem specs carefully.

② FUNCTION CALL (DECLARE VARIABLES USED IN THE CALL)

```
int A[5] = {1, 2, 3};
int B = 5;
int C = 3;
boolean check = isMember (A, B, C);
```

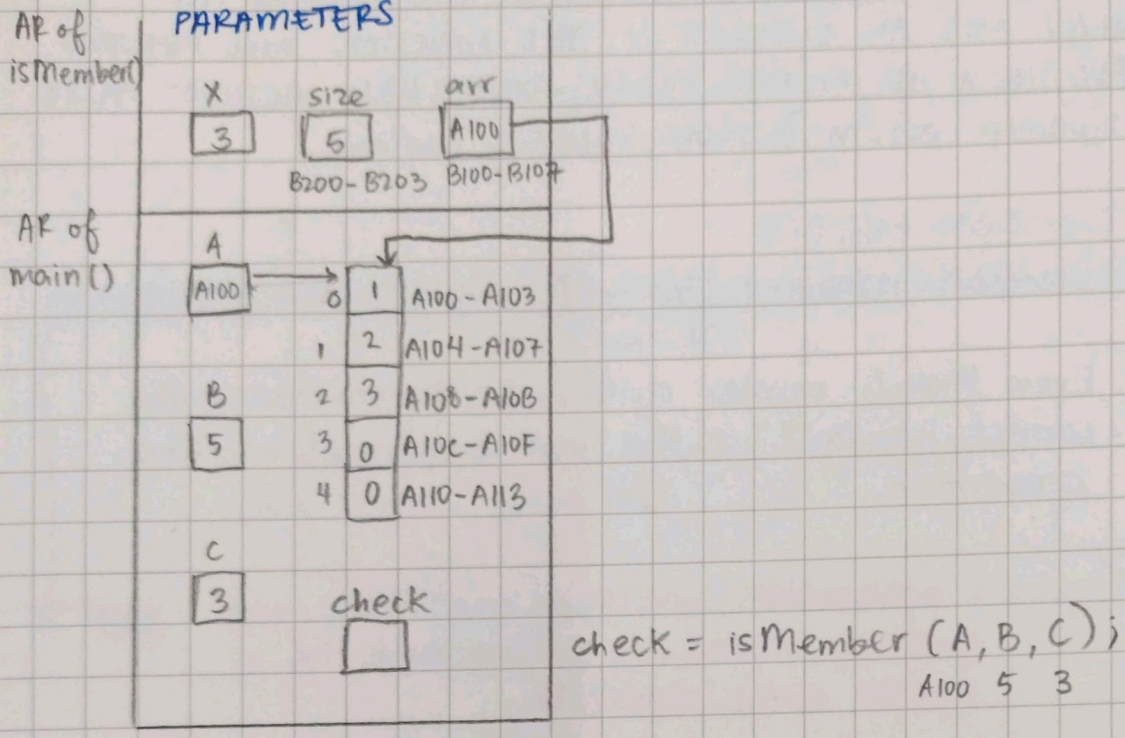
0	1	2	3	4
1	2	3	0	0

to catch the return value of the function. DONT FORGET THIS!

isMember (A, B, C)

- A: array A where the name of the array is the address of the first component. Passes an address
- B: passes size of the array which is int
- C: passes the value element X which is to be searched for

③ DRAW THE EXECUTION STACK ILLUSTRATING THE FUNCTION CALL.



④ CODE OF THE FUNCTION
(my version with step-wise refinement)

```
boolean isMember (int arr[], int size, int x) {
    boolean retval = ~~TRUE~~;
    int ndx;
    for (ndx = 0; ndx < size && arr[ndx] != x ; ndx++) {}
    if (ndx == size) {
        retval = FALSE;
    } else { retval = TRUE; }
    return retval;
}
```

classic if-else statement that can be written with ternary operator

```
(ndx == size)? retval = FALSE : retval = TRUE;
retval = (ndx == size)? FALSE : TRUE;

return (ndx == size)? FALSE : TRUE;
```

(Another version)

```
boolean isMember (int arr[], int size, int x) {
        int i;
        for (i=0; i<size && arr[i] != x; i++){}
        return i<size? TRUE : FALSE;
}
```

wow so concise

Q. PROBLEM SPECIFICATION:

Function inputArray() ALLOWS USER TO INPUT FROM THE TOTAL NUMBER OF VALUES "N" TO BE STORED IN THE NEWLY CREATED ARRAY & PUTS "N" AT INDEX 0 OF THE NEW ARRAY. THE VALUES OF THE ARRAY FROM INDEX 1 TO N WILL ALSO BE INPUTTED FROM THE KEYBOARD. IN ADDITION, THE NEWLY CREATED ARRAY WILL BE RETURNED TO THE CALLING FUNCTION.

ANS.

SUBTASKS (LOGIC BROKEN DOWN FOR A DUMMY LIKE ME):

1. INPUT VALUE OF N
2. ALLOCATE THE ARRAY DYNAMICALLY (MALLOC OR CALLOC)
3. INITIALIZE INDEX 0 OF THE NEW ARRAY
4. INPUT N NUMBER OF INTEGERS
5. RETURN THE ARRAY

code of the function on ⇒ the next page.

(My version)

no parameters because the question did not specify & does not need parameters

```
int * InputArray () {
```

int * as return type.

C programming does not allow to return entire array but can return a pointer to an array. In this case, what is returned is a pointer to an array of integers.

```
int N, ndx, * retval;
```

catches the return value

```
scanf ("%d", &N);
retval = (int *) malloc (sizeof (int) * (N + 1));
```

new array, as stated in the question, will hold the value N + the N number of values i.e N + 1 number of values in the resulting array.

```
if (retval != NULL) { // always check if malloc succeeds
    retval [0] = N; // initializing index [0] to be assigned "N"
    for (ndx = 1; ndx <= N; ndx ++) {
        scanf ("%d", &retval [ndx]);
```

OR *(retval + ndx)

```
    }
}
return retval; // returning the pointer to the dynamically
                  allocated resulting array.
}
```

MALLOC() & CALLOC() - DYNAMIC MEMORY ALLOCATION FURTHER EXPLAINED

SYNTAX OF MALLOC() datatype variable

```
void * malloc (size_t size)
```

* Note: size_t is not a keyword in C but is a datatype defined in a header file.

★ Note: When writing code, base it on syntax, not example. Notice how void * malloc (size_t size), the correct syntax of malloc() is written like a function header, where as something like

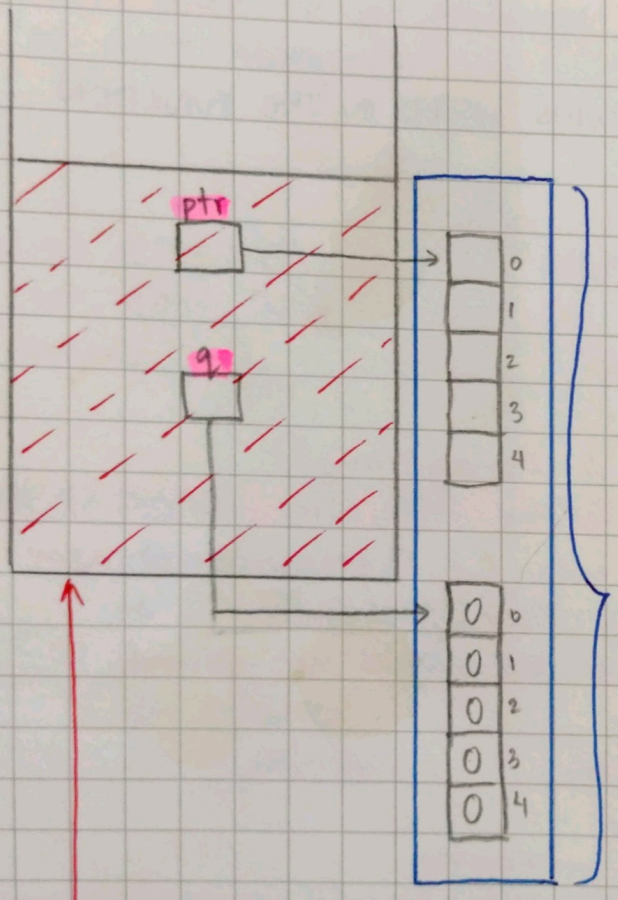
Eg. int * ptr = (int *) malloc (5 * sizeof (int)); is an example

SYNTAX OF CALLOC()

element count ↑ item size ↑

void * calloc (size_t NoOfItems, size_t ItemSize)

Eg. int *q = (int *) calloc (5, sizeof (int));



even if the function terminates and ptr and q are no longer alive in memory, the dynamically allocated spaces will remain. Lifetime of dynamically allocated space starts during program execution & ends when free(); is used.

when the function terminates, ptr and q will no longer be alive in memory.