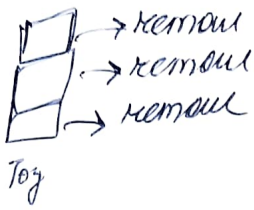
b) Recursive Case - the step where function calls itself with a smaller input. (eg. n-1).

⇒ 

Pointers in C

⊙ What are Pointers?

↳ a pointer is a variable that stores the memory address of another variable

↳ Instead of holding data directly, it "points to", where the data lives in memory.

↳ think of it as a GPS coordinate — it doesn't hold the treasure, it tells you where to find it.

↳ Key concepts:

a) declaration: type *pointer_name (*ptr)

b) Address operator: (&) : get the address of variable (e.g. &x)

c) dereference operator (*) : access the value at the address (*ptr)

Ex -

Memory;

[address: 1000] → [value: 42] // variable x

[address: 1004] → [value: 1000] // pointer ptr pointing to x

ptr = &x; // ptr holds 1000

*ptr = 42; // acc. value at 1000

+91-8233266276
info@grootacademy.com
122/166, 2nd Floor, vijay path, Mansarovar, 302020
www.Grootacademy.com

→
```
#include <stdio.h>
int main() {
    int x = 42;          // variable
    int *ptr = &x;       // pointer to x
    printf("value of x : %d \n", x);          // 42
    printf("address of x : %p \n", (void*)&x);
    printf("value of ptr : %p \n", (void*)ptr);
    printf("value at ptr : %d \n", *ptr);     // 42
    *ptr = 99;           // change x via pointer
    printf("new value of x : %d \n", x);      // 99
    return 0;
}
```

⇒ Pointer Expressions

- what are they?
  - ↳ pointer expressions involve operations on pointers, like arithmatic or comparisons.
  - ↳ pointers are tied to memory addresses, so these operations manipulate where they point.
  - ↳ key operations:
    - a.) add^n / subtr^n:
      - ↳ ptr + 1 moves forward by sizeof(type) bytes
      - ↳ subtraction: diff. b/w 2 pointers (in elements)
      - ↳ ex- `int arr[3] = {10, 20, 30};`

Memory:

[1000] → 10
[1004] → 20
[1008] → 30

```
int *ptr = arr;
ptr + 1 = 1004   → moves next
```

↳ int = 4 Byte.

Pointers & Arrays

Connection

a) array and pointers are closely linked in c
b) An array name (e.g. arr) is a constant pointer to its first element (&arr[0]).
c) arr[i] is equivalent to *(arr+i)

or-

```
int arr[4] = {1, 2, 3, 4};
```

Memory.

```
[2000] → 1
[2004] → 2
[2008] → 3
[2012] → 4
```

arr = 2000 (fixed)
int *ptr = arr;
ptr = 2000 (can change)

or-

```
#include <stdio.h>
int main() {
    int arr[4] = {1, 2, 3, 4};
    int *ptr = arr;
    // access via array notation
    for(int i = 0; i < 4; i++) {
        printf("arr[%d] = %d \n", i, arr[i]);
    }
    // access via pointer
    for(int i = 0; i < 4; i++) {
        printf("*(ptr + %d) = %d \n", i, *(ptr+i));
    }
    // Modify via pointer
    *(ptr+i) = 99;
    printf("arr[i] = %d \n", arr[i]);
    return 0;
}
```

⊙ Pointers & character strings

↳ connection

a) a string is a (char) array ending with '\0'.

b) a (char*) pointer can point to a string.

ex-

```
char str[] = "Hi";
```

Memory:

[3000] -> 'H'
[3001] -> 'i'
[3002] -> '\0'

```
char *ptr = str;
```
(ptr = 3000)

ex-

```
#include <stdio.h>
int main() {
    char str[] = "Hello";
    char *ptr = str;
    printf("String via array : %s\n", str);   // Hello
    printf("String via pointer : %s\n", ptr);  // Hello

    // move pointer & print
    ptr = ptr + 1;
    printf("after +1 : %s\n", ptr);            // ello

    // modify only str,
    *ptr = 'a';
    printf("modified str : %s\n", str);        // Hallo
    return 0;
}
```

* char *ptr = "Hi";
* char str[] is mutable, char * isn't.

⊙ Pointer to Functions

↳ a function pointer stores the address of a function, letting you call it indirectly.
↳ functions live in memory, and their address can be pointed at.
↳ Syntax:

```
return_type (*pointer_name)(parameter_types);
```

eg.
↳ function : int add (int, int)
Memory : [5000] → [add's code]

```
int (*fptr)(int, int) = add;        // fptr = 5000
```

↳ ex-

```
#include <stdio.h>

int add(int a, int b) {
    return a+b;
}
int main() {
    // declar pointer to fn
    int (*fptr)(int, int) = add;
    // call via pointer
    int result = fptr(3,4);
    printf("Result : %d \n", result);   // 7
    return 0;
}
```

⊙ useful in dyanic function calls
⊙ flexibility in design.

Pointers & Structures

↳ Connection

- ⊙ pointers can point to structures. (custom data types)
- ⊙ use (->) to access members via a pointer.
- ⊙ diagram:

```
struct Point { int x, y; }
struct Point p = {3,4};
```

Memory:

```
[6000] → 3      // p.x
[6004] → 4      // p.y
struct Point *ptr = &p      // ptr = 6000.
```

Ex-

```
#include <stdio.h>

struct Point {
    int x, y;
};

int main(){
    struct Point p = {3,4};
    struct Point *ptr = &p;

    printf("x: %d, y: %d \n", p.x, p.y);      // 3,4
    printf("x: %d, y: %d \n", ptr->x, ptr->y);  // 3
```

→ Modify via pointer,

```
ptr->x = 10;
printf("New x: %d \n", p.x);      // 10

return 0;
}
```

↳ Miscellaneous:

⊙ Visual diagram:

Memory:

[1000] → 25 // int x = 25

[1004] → 1000 // int *ptr = &x

⊙ x lives at address 1000, holds value 25

⊙ ptr at 1004 holds (1000) [points to x]

Basic declration ⇒ int x = 25;

int *ptr = &x; (ptr hold's x's address)

deferencing ⇒ Dereferencing: *ptr gets the value at that address.

↳ printf("%d\n", *ptr) // 25

Ex ⇒ declare a float f = 3.14, a pointer to it, and print its value via the pointer.

↳

```
float f = 3.14;
float *ptr = &f;
printf("%f\n", *ptr);
```

⇒ ptr = 5; *ptr = 5;

ptr = 5; (sets the pointer) to address 5 X

```
int x = 25;
int *ptr = &x;
*ptr = 5;
printf("%d\n", x);
```