1. Memory Leak

Definition: Memory leak happens when we allocate memory dynamically (malloc/calloc) but never release it (free()), and we lose access to it.

Impact: The memory remains reserved in the heap, but program cannot use it → wastes memory (critical in embedded/long-running systems).

Ex:

```
int main() {
    int *p = (int*) malloc(3 * sizeof(int)); // allocate memory for 3
integers

    p[0] = 10;
    p[1] = 20;
    p[2] = 30;

    printf("Values: %d %d %d\n", p[0], p[1], p[2]);

    // forgot free(p);  Memory Leak occurs

    return 0;
}
```

Tracing (Memory Layout)

- **Before malloc**
    - **Stack: p (uninitialized)**
    - **Heap: empty**

1.

After malloc
Stack: p → 0x1000   (say address 0x1000 allocated in heap)

Heap: [0x1000] 10 | 20 | 30

2.

At program end (without free)

Stack: destroyed

3. **`Heap: memory at 0x1000 not freed → inaccessible → LEAK`**

Using:

```
int main() {
    int *p = (int*) malloc(3 * sizeof(int));

    p[0] = 10;
    p[1] = 20;
    p[2] = 30;

    printf("Values: %d %d %d\n", p[0], p[1], p[2]);

    free(p);    //  memory released
    p = NULL;   //  avoid dangling pointer
    return 0;
}
```

Example:

```
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(100 * sizeof(int));
    // use p...
    p = NULL;   // ❌ lost reference, memory not freed
    return 0;
}
```

Here, 100 integers worth of memory are allocated but never freed → **memory leak**.

Analogy:
Like renting 100 chairs for an event but never returning them. They sit unused, and no one else can use them.

---

2. Dangling Pointer

A **dangling pointer** is a pointer that points to memory that has been **freed** or **gone out of scope**.

- Pointer still holds the **old address**, but the memory is **invalid**.
- `Using it` → **undefined behavior** (crash, garbage values, security issues).

```
cdac@cdac-virtual-machine:~/Mahesh/pointers/dynamic_memory$ cat dang_Ex.c
#include <stdio.h>

int* getPointer() {
    int x = 10;   // local variable (stack memory)
    return &x;    // address of local variable returned
}

int main() {
    int *p = getPointer();
    printf("%d\n", *p);  //  dangling pointer
    return 0;
}
```

Step-by-Step Tracing

Step 1: Inside getPointer()

Stack Frame (getPointer):

x = 10   [address 0x7ffeabcd]

`return &x → p = 0x7ffeabcd`

Step 2: After function returns

Stack Frame (getPointer) destroyed

p still = 0x7ffeabcd (address of invalid memory)

Step 3: Dereferencing *p

`p → invalid memory`

`* p = unpredictable → crash or garbage`

Example 1 (free case):

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(sizeof(int));
    *p = 10;
    free(p);      // memory freed
    printf("%d", *p);  // ❌ dangling pointer: accessing freed memory
    return 0;
}
```

Example 2 (scope case):

```
int* getPointer() {
    int x = 10;
    return &x;   // returning address of local variable
}
```

After function ends, x disappears, so the pointer is **dangling**.

Analogy:
Imagine writing a friend's house address on a slip, but the house was demolished. The address is still there, but going there is unsafe.

---

3. Wild Pointer

A **wild pointer** is a pointer that is **declared but not initialized**.

- It contains a **garbage address** (whatever random value is present in that memory location/register).
- `Dereferencing a wild pointer` → **undefined behavior** (crash, wrong data, overwrite).

Example:

```
int main() {
    int *p;        // declared but not initialized → WILD pointer
    *p = 10;     // ❌ writing into random memory location
    printf("%d\n", *p);
    return 0;
```

}

Analogy:
You pick up a random house key from the street (wild pointer).

- You don't know which house it belongs to.
- If you try to open some door with it → ❌ trouble, may even break into wrong house.

Safe:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = NULL;          // good practice → not wild
    p = (int*) malloc(sizeof(int)); // allocate valid memory
    *p = 10;
    printf("%d\n", *p);    // ✅ prints 10
    free(p);               // free after use
    return 0;
}
```

---

4. Double Free

A **double free** happens when we call free() **twice** on the same memory pointer.

First free() releases the memory → valid.

Second free() tries to release again → ❌ undefined behavior (can crash, corrupt heap, or even be exploited in attacks).

Example:

```
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(10 * sizeof(int));
    free(p);
    free(p);   // ❌ double free
    return 0;
}
```

Analogy:
Like returning the same rented chair **two times** → vendor gets confused, may cause a fight!

5)NULL Pointer:

- A pointer that is initialized with NULL (value 0x0 address).
- It does not point to any valid memory.
- `Dereferencing a NULL pointer → always a segmentation fault.`
- Safe to check: if (p == NULL) before use.

```
cdac@cdac-virtual-machine:~/Mahesh/pointers/dynamic_memory$ cat null.c
#include <stdio.h>

int main() {
    int *p = NULL;    // initialized
    if (p == NULL) {
        printf("Pointer is NULL, no memory assigned yet.\n");
    }
     *p = 5;   // ❌ would crash
    return 0;
}
```

Summary Table

Problem	Cause	Example	Analogy
Memory Leak	Memory allocated but never freed	p = NULL; without free(p)	Not returning chairs
Dangling Ptr	Using freed/out-of-scope pointer	Access after free(p)	Visiting demolished house

Wild Ptr	Using uninitialized pointer	int *p; *p=5;	Calling random number
Double Free	Freeing memory twice	free(p); free(p);	Returning chair twice

Best Practices to Avoid Memory Problems

1. Always initialize pointers

```
int *p = NULL;   // safe
```

- Avoids **wild pointers**.
- You know a NULL pointer is invalid, so you won't accidentally dereference garbage.

2. After free(), set pointer to NULL

```
free(p);
p = NULL;   // avoids dangling pointer
```

- Ensures you don't accidentally use or free it again.
- Checking if (p != NULL) before using is safer.

3. Check return value of malloc/calloc/realloc

```
p = malloc(100 * sizeof(int));
if (p == NULL) {
    printf("Memory allocation failed!\n");
    exit(1);
}
```

- Prevents accessing invalid memory when allocation fails.
- Otherwise, program may crash.

---

4. Pair allocations with deallocations

- malloc → free
- calloc → free
- realloc → free (when done)
  Think like borrowing & returning chairs.

5. Avoid double free

- Keep a **clear ownership rule**: the one who allocates is responsible to free.

- Always set to NULL after freeing.