Void Pointer in C

- A **void pointer** is a special type of pointer that can hold the address of any data type.
- Declared using `void *`.
- Since its type is not known, it **cannot be dereferenced directly** without typecasting.

Exmaple:

```
#include <stdio.h>
int main() {
    int a = 10;
    float b = 5.5;
    char c = 'X';

    void *ptr;  // void pointer

    // Pointing to an integer
    ptr = &a;
    printf("Value of a = %d\n", *(int *)ptr);

    // Pointing to a float
    ptr = &b;
    printf("Value of b = %.2f\n", *(float *)ptr);

    // Pointing to a char
    ptr = &c;
    printf("Value of c = %c\n", *(char *)ptr);

    return 0;
}
```

Key Points:

1. Can point to any data type (`int`, `float`, `char`, etc.).
2. Must be **typecasted** before dereferencing.
3. `printf("%d", *(int *)ptr);`
4. Used in **generic functions** (like `malloc`, `calloc` in stdlib.h return `void *`).
5. Increases flexibility and reusability of code.

Null Pointer in C

- A **null pointer** is a pointer that does not point to any valid memory location.
- It is used to **initialize pointers safely** before assigning them proper addresses.

Key Points:

1. Value of a null pointer is `0` (but better to use `NULL` macro from `<stddef.h>`).
2. Prevents accidental access to garbage memory.
3. Often used to check whether a pointer is assigned or not:
4. Helps in avoiding **dangling pointers** (pointers that point to freed memory).

Example:

```
#include <stdio.h>
int main() {
```

```
    int *p = NULL;   // null pointer

    if (p == NULL) {
        printf("Pointer is NULL, no memory assigned yet.\n");
    }

    int x = 100;
    p = &x;   // now memory assigned
    printf("Value of x = %d\n", *p);

    return 0;
}
```

Dangling Pointer in C

Definition

A **dangling pointer** is a pointer that **points to a memory location which has already been freed or is no longer valid**.
Using such a pointer may cause **unexpected behavior, crashes, or security issues**.

Causes of Dangling Pointers

1. Deallocation of memory

When memory is freed using `free()` but pointer still points to that location.

```
int *ptr = (int*)malloc(sizeof(int));
free(ptr);   // memory freed
*ptr = 10;   // Dangling pointer access
```

2. Returning local variable address

Returning address of a local variable from a function.

```
int* fun() {
    int x = 5;
    return &x;  // Dangling pointer (x is destroyed after function ends)
}
```

3. Out of scope variables

Pointer points to a variable that goes out of scope.

```
int* ptr;
{
    int a = 10;
    ptr = &a;
}  // 'a' goes out of scope, ptr becomes dangling
```

Problems/Issues Caused by Dangling Pointers

1. **Segmentation Fault (Crash)** – accessing freed memory may cause program crash.
2. **Data Corruption** – program may overwrite random memory locations.
3. **Security Issues** – attackers can exploit dangling pointers to execute malicious code.

Prevention

- After freeing memory, set pointer to `NULL`.
- `free(ptr);`
- `ptr = NULL;  // Safe`
- Avoid returning local variable addresses.
- Initialize pointers properly before using.

Wild Pointer in C

Definition

A **wild pointer** is a pointer that **is declared but not initialized**, so it points to an unknown or garbage memory location.
Using it causes **undefined behavior** (program crash, wrong values, etc.).

Example

```
int *ptr;   // Declared but not initialized → Wild pointer
*ptr = 10;  // Undefined behavior (may crash)
```

Causes of Wild Pointers

1. Pointer declared but **not initialized**.
2. Pointer not assigned any valid address before use.

Problems/Issues

1. **Program Crash** (Segmentation Fault).
2. **Unpredictable Behavior** (garbage values, corruption).
3. **Security Risks** (attackers may exploit).

Prevention

- Always initialize pointers at declaration:
- `int *ptr = NULL;      // Safe initialization`
- `int a = 10;`
- `int *p = &a;          // Valid pointer`
- Do not use uninitialized pointers.

Difference between Dangling vs Wild Pointer (for clarity):

- **Wild Pointer** → Never initialized.
- **Dangling Pointer** → Was valid once, but became invalid after memory free/out of scope.

C Preprocessing

Definition

- **Preprocessing** in C is the **step performed before compilation**.

- A **preprocessor** is a program that processes the source code before it is compiled.
- Preprocessing commands begin with a `#` symbol.

Types of Preprocessor Directives

1. **Macros (`#define`)**

Used to define constants or short code replacements.

```
#define PI 3.14
#define SQUARE(x) (x*x)
```

2. **File Inclusion (`#include`)**

Used to include header files.

```
#include <stdio.h>   // Standard header file
#include "myfile.h"  // User-defined header file
```

3. **Conditional Compilation (`#if`, `#else`, `#elif`, `#endif`, `#ifdef`, `#ifndef`)**

Used to compile parts of the program conditionally.

```
#define DEBUG
#ifdef DEBUG
   printf("Debugging mode\n");
#endif
```

4. **Other Directives**

`#undef` → Undefines a macro.
`#pragma` → Gives special instructions to the compiler.

Advantages of Preprocessing

- Increases **code readability**.
- Reduces **repetition** (macros, constants).
- Makes **modular programming** easier (header files).
- Allows **conditional compilation** (useful for debugging and portability).

Predefined Macros in C

Macro	Meaning
`__LINE__`	Current line number in the source code.
`__FILE__`	Current file name (as a string).
`__DATE__`	Date of compilation (in "Mmm dd yyyy" format).
`__TIME__`	Time of compilation (in "hh:mm:ss" format).

Example Program

```
#include <stdio.h>

int main() {
    printf("File Name   : %s\n", __FILE__);
    printf("Line Number : %d\n", __LINE__);
    printf("Date        : %s\n", __DATE__);
    printf("Time        : %s\n", __TIME__);
    return 0;
}
```

File Handling in C

Definition

- **File handling** in C allows us to **store data permanently** on disk and **read/write** it later.
- We use a special pointer called **File Pointer** (`FILE *`) for file operations.

Basic Steps in File Handling

1. **Declare a file pointer**
2. `FILE *fp;`
3. **Open a file** using `fopen()` with a specific mode.
4. **Perform operations** (read/write/append).
5. **Close the file** using `fclose()`.

File Opening Modes

Mode	Meaning
`"r"`	Open file for **reading** (file must exist).
`"w"`	Open file for **writing** (creates new or overwrites existing file).
`"a"`	Open file for **appending** (writes at end, creates new if not exist).
`"r+"`	Open for **reading and writing** (file must exist).
`"w+"`	Open for **reading and writing**, creates new file (overwrite).
`"a+"`	Open for **reading and appending**.

Common Functions

- `fopen("filename", "mode")` → Opens file.
- `fclose(fp)` → Closes file.
- `fprintf(fp, "format", data)` → Write to file.
- `fscanf(fp, "format", &data)` → Read from file.
- `fgetc(fp)` / `fputc(ch, fp)` → Read/Write single character.
- `fgets(str, size, fp)` / `fputs(str, fp)` → Read/Write string.

Example 1: Write to a File

```
#include <stdio.h>

int main() {
    FILE *fp;
    fp = fopen("example.txt", "w");   // open file in write mode

    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    fprintf(fp, "Hello, File Handling in C!\n");
    fclose(fp);   // close file

    printf("Data written successfully.\n");
    return 0;
}
```

Example 2: Read from a File

```
#include <stdio.h>

int main() {
    FILE *fp;
    char str[100];

    fp = fopen("example.txt", "r");   // open file in read mode
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    while (fgets(str, 100, fp) != NULL) {
        printf("%s", str);   // print file content
    }

    fclose(fp);
    return 0;
}
```