Conditional Compilation in C

Introduction

Conditional Compilation is a feature provided by the C preprocessor. It allows the compiler to include or exclude specific blocks of code during the compilation process. This helps in writing flexible code that can behave differently depending on certain conditions.

Why is Conditional Compilation Used?

- To **compile or skip** specific parts of code based on certain conditions.
- Helpful in **debugging** – you can include print statements temporarily.
- Useful in **platform-specific code** – such as compiling different code for Windows vs Linux.
- Helps in **managing large codebases** by selectively compiling parts.

Example 1: Macro Check

Suppose we declare a macro and we want to check whether it is already defined or not:

```
#ifdef MY_MACRO
   // Do nothing
#else
   #define MY_MACRO
   printf("Macro is now defined.\n");
#endif
```

- If `MY_MACRO` is already defined, the code inside `#ifdef` block runs.
- Otherwise, the macro is defined and a message is printed.

Example 2: Debugging Purpose

Suppose we want to check the input stored in a variable only while debugging:

```
#define DEBUG  // Comment this line to disable debug output

int a;
scanf("%d", &a);

#ifdef DEBUG
    printf("Input = %d\n", a);
#endif

printf("Result = %d\n", a * 10);
```

- When `DEBUG` is defined, the input is printed.
- In production, just comment out the `#define DEBUG` line to disable debug prints without removing code.

Conditional Compilation Directives and Their Uses

Directive	Purpose
`#if`	Checks whether a condition is true.
`#ifdef`	Checks if a macro is defined.
`#ifndef`	Checks if a macro is not defined.
`#elif`	Provides another condition if the previous fails.
`#else`	Executes when none of the above conditions are met.
`#endif`	Marks the end of the conditional block.

Extra: `#pragma` Directive

- `#pragma once` is used to **avoid multiple inclusions** of the same header file.

- It is a **compiler-specific extension** (works in GCC and modern compilers).
- It acts as an alternative to:

```
#ifndef HEADER_FILE
#define HEADER_FILE
// code
#endif
```

What Happens in Memory with Conditional Compilation?

🔧 1. At What Stage Does Conditional Compilation Happen?

Conditional Compilation happens during the **preprocessing phase**, which occurs *before* actual compilation.
That means memory allocation, variable creation, or stack/heap activity has not started yet.

🧠 2. What Happens in Memory After Conditional Compilation?

Let's understand using an example:

📄 **Code:**

```
#include <stdio.h>

#define DEBUG

int main() {
    int x = 10;

#ifdef DEBUG
    printf("Debug Info: x = %d\n", x);
#endif

    printf("Result: %d\n", x * 2);
    return 0;
}
```

🔍 What Does the Preprocessor Do?

Since `DEBUG` is defined, the line `printf("Debug Info...")` is **retained** in the compiled output.
If `DEBUG` were not defined, only `printf("Result...")` would remain.

🧬 Memory Perspective (When DEBUG is defined):

1. `int x = 10;`
    - Stored in **stack memory** (4 bytes typically for int).
2. `printf("Debug Info...")`
    - The string literal `"Debug Info: x = %d\n"` is stored in the **read-only data segment**.
    - `printf()` function is invoked from the **stack** during runtime.
3. `printf("Result...")`
    - Another string literal stored in the **read-only data segment**.

📤 If `DEBUG` is not defined:

```
#include <stdio.h>

int main()
{
   int x = 10;
   printf("Result: %d\n", x * 2);
   return 0;
}
```

- The line `printf("Debug Info...")` is **removed at preprocessing stage**.
- No memory is allocated for the debug string.
- The stack frame is smaller.

- The compiled binary is smaller.

📌 Summary Table: Impact of DEBUG on Memory

Component	DEBUG defined	DEBUG undefined
String Literal (`Debug Info`)	Stored in read-only section	Not present
`printf()` call	Compiled and executed	Not present
Stack usage	Slightly higher	Lower
Binary size	Slightly larger	Smaller
Output	Shows debug info	Does not show debug info

Conclusion: Memory Effects of Conditional Compilation

- **No memory is reserved** for excluded code.
- **Reduces memory footprint** by skipping unnecessary string literals or function calls.
- **Improves runtime efficiency**.
- **Helps maintain clean and optimized binaries**.
