🔍Preprocessor Stage in C

📌 1. What is the Preprocessor?

The **C Preprocessor** is a **program that processes your source code before it is compiled** by the compiler.

It handles everything **before the actual compilation** begins.

🛠️ **Key Tasks**:

- File Inclusion (`#include`)
- Macro Expansion (`#define`)
- Conditional Compilation (`#ifdef`, `#ifndef`, etc.)
- Removing Comments

✅ The output of this stage is a **"pure C code file"** with all macros expanded and header files included.

🔁 2. How Preprocessor Works (Step-by-Step)

1️⃣ You write your source code: `program.c`
2️⃣ The Preprocessor scans for lines beginning with `#`
3️⃣ It **expands macros**, **adds included files**, and **removes comments**
4️⃣ Generates an **intermediate file**: `program.i`
5️⃣ This `.i` file is passed to the compiler

📌 `gcc -E program.c -o output.i`
👉 Use this command to see the output of the preprocessor stage.

🔠 3. Types of Preprocessor Directives

Directive	Purpose
`#include`	To include header files
`#define`	To define macros/constants
`#undef`	To undefine a macro **(is used to remove a previously defined macro)**
`#ifdef`	Conditional compilation (if defined)
`#ifndef`	Conditional compilation (if not defined)
`#if`, `#else`	Conditional code inclusion
`#error`	Generates custom compile-time error

📁 4. #include Directive

- Used to **add external files** to your program.

Syntax:

```
#include <file.h>   // For standard library files
#include "file.h"   // For user-defined files
```

Working:

Type	Search Path
`<file.h>`	Compiler looks in **system paths (default path)**
`"file.h"`	Looks in the **current directory** first

📂 Example:

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

📌 **In Preprocessor Stage**:
→ The actual contents of the `.h` file are **pasted** into your `.c` file.

🔣 5. #define Directive & Macro Expansion

The `#define` directive is used to create **macros** — essentially, **textual replacements**.

1️⃣ Object-like Macros

✅ Used to define constants

🔤 Syntax:

```
#define PI 3.1415
```

🔁 All instances of `PI` will be **replaced** with `3.1415` before compilation.

🔍 **Memory Insight**:

- **At Preprocessing**: Replaced in code.
- **At Compile Time**: Treated as literal constants.
- **Memory Segment**: If used in computation, value may reside in the **stack** or **register**.

2️⃣ Function-like Macros

✅ Used to simulate small functions.

🔤 Syntax:

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

🔁 All instances like `SQUARE(5)` become `((5) * (5))` before compilation.

📌 **No actual memory is allocated for `SQUARE` function** because it's just text replacement.

🔍 **Memory Insight**:

- **Preprocessor**: Performs substitution.
- **Compiler**: Compiles the result like normal expression.
- **Runtime**: Evaluated like any expression.
- **Stored In**: **Stack/register**, depending on usage.

⚠️ **Pitfall**: Use parentheses carefully to avoid incorrect expansion.

3️⃣ Multiline Macros

✅ Allow you to define macros that span multiple lines.

🔤 Syntax:

```
#define DISPLAY(a, b) \
printf("A = %d\n", a); \
printf("B = %d\n", b);
```

💡 Use `\` to indicate line continuation.

🔍 **In Memory**:

- Acts as if those lines were typed normally.
- No memory is used for the macro itself.

🧠 6. How Macros are Stored in Memory (In-Depth)

Stage	What Happens	Memory Used
Preprocessor	Replaces macros with actual values/code	None
Compiler	Compiles replaced values as normal code	Code Segment
Execution	Literal constants/functions executed	Stack/Register

🔍 Memory Segments:

- 📌 **Code Segment**: Stores compiled program instructions
- 📌 **Data Segment**: Stores global/static variables
- 📌 **Stack Segment**: Stores local variables and function calls
- 📌 **Heap**: Dynamic memory allocation

✅ Macros themselves **do not occupy memory**. Only the **values they expand into** may use memory during runtime.

❗ 7. Common Errors with Preprocessor

Using macros without parentheses:

```
#define SQUARE(x) x*x   // ❌ Error-prone
#define SQUARE(x) ((x)*(x)) // ✅ Safe
```

- Forgetting `\` in multiline macros

Recursive macros (not allowed):

```
#define A B
#define B A  // ❌ Infinite loop
```

✅ 8. Why Use Preprocessor Directives?

🔹 Improve **readability**
🔹 Avoid **magic numbers**
🔹 Enable **portable code**
🔹 Helpful for **debugging**
🔹 Useful in **modular programming**

🎯 Real-Life Analogy

💡 Imagine writing a cooking recipe:

- You say: “Use 1 tsp salt” ➡️ Like `#define SALT 1`
- And: “Include base ingredients” ➡️ Like `#include "base.h"`

🔁 Before someone starts cooking (compilation), they **replace** SALT with **actual quantity** and **include** the base ingredients into the recipe.
