GCC Compilation Stages and Essential Flags

Phase 1: The Four Stages of Compilation

1. Preprocessing (`-E`)

This stage expands macros, includes header files, and removes comments.

- **Flag** : `-E`
- **Output**: Preprocessed Source Code (text)

Command:

```
# Stops after preprocessing. Output goes to standard out (terminal) by default.
gcc -E main.c > main.i
# OR using -o
gcc -E main.c -o main.i


### 2. Compilation (`-S`)
This stage translates the preprocessed code into Assembly language specific to your processor
(x86_64, ARM, etc.).

- **Flag**: `-S` (Note: Capital 'S')
- **Output**: Assembly Code (.s or .asm file)

**Command**:
```bash
# Generates 'main.s'
gcc -S main.c -o main.s
```

3. Assembly (`-c`)

This stage translates assembly code into machine code (binary), but it is not yet executable because it hasn't been linked to system libraries.

- **Flag**: `-c`
- **Output**: Relocatable Object File (.o)

Command:

```
# Generates 'main.o'
gcc -c main.c -o main.o
```

4. Linking (Default)

The final stage combines your object files (.o) with system libraries (like printf from glibc) to create the final executable.

- **Flag**: (None, or `-o` to name the file)
- **Output**: Executable binary

Command:
```

```
# Generates 'my_program' (on Linux) or 'my_program.exe' (on Windows)
gcc main.o -o my_program
# Or directly from source:
gcc main.c -o my_program
```

Phase 2: Essential Daily Drivers (Intermediate)

These are the flags you should be using for almost every project to catch bugs and configure your build.

Warnings (Crucial)

Never compile without these. They catch undefined behavior before it crashes your program.

- `-Wall` : Enables "all" common warnings (unused variables, bad return types).
- `-Wextra` : Enables extra warnings that `-Wall` misses.
- `-Werror` : Treats all warnings as errors (stops compilation). Great for forcing discipline.

```
gcc -Wall -Wextra main.c -o app
# For even stricter code:
gcc -Wall -Wextra -Werror main.c -o app
```

Debugging

- `-g` : Adds "debug symbols" to your binary. This allows `gdb` (GNU Debugger) to show you the original C source code line-by-line instead of just memory addresses.

```
gcc -g main.c -o app_debug
```

C Standards

- `-std=c99` / `-std=c11` / `-std=c17` : Force the compiler to adhere to a specific C standard.
- `-ansi` : Equivalent to C89 (strict old-school C).

```
gcc -std=c11 main.c -o app
```

Phase 3: Optimization & Performance (Advanced)

When you are ready to release your code, use these flags.

- `-O0` (Default): No optimization. Fastest compilation time, best for debugging.
- `-O1` : Basic optimization.
- `-O2` : Recommended for deployment. Optimizes for speed without increasing binary size too much.
- `-O3` : Aggressive optimization. Can make compilation slow and debugging impossible. Sometimes increases binary size (loop unrolling).
- `-Os` : Optimize for size. Great for embedded systems where memory is tight.
- `-Ofast` : Disregards strict standards compliance for raw speed (can break math precision).

```
# Typical release build command
gcc -O2 -Wall -Wextra main.c -o production_app
```

Phase 4: Systems & Libraries (Expert)

Since you are interested in kernel and socket programming, you will often need to link against external libraries (like pthreads for threading or openssl for crypto).

Linking Libraries

- `-l<name>` : Link a library. Note: drop the `lib` prefix and `.so` / `.a` extension.
  - Example: To link `libm.so` (math library), use `-lm` .
  - Example: To link `libpthread.so` , use `-lpthread` .
- `-L<path>` : Tells gcc where to look for library files if they aren't in standard folders (`/usr/lib` , etc.).

Header Files

- `-I<path>` : Tells gcc where to look for header files (.h) if they aren't in `/usr/include` .

Preprocessor Definitions

- `-DNAME` : Defines a macro from the command line (useful for conditional compilation without changing code).

Example in code:

```
#ifdef DEBUG
    printf("Debug mode enabled\n");
#endif
```

Command:

```
gcc -DDEBUG main.c -o app
```

Recommended daily development command:

```
gcc -Wall -Wextra -Werror -g -std=c11 main.c -o app_debug
```

Recommended release command:

```
gcc -Wall -Wextra -O2 -std=c11 main.c -o app
```

```
# GCC Advanced Compilation Flags (Continued)

## Phase 5: Library Creation (Crucial for your Project)
Since you plan to build a Linear Algebra library, you need to understand how to build Static (`.a`)
and Shared (`.so`) libraries.

### 1. Position Independent Code (`-fPIC`)
- **Concept**: Shared libraries (DLLs in Windows, `.so` in Linux) are loaded into memory at different
addresses for different programs. The machine code cannot rely on fixed addresses.
- **Flag**: `-fPIC`
- **Usage**: Mandatory when compiling object files for a shared library.

```bash
gcc -c -fPIC matrix.c -o matrix.o
```

2. Creating a Shared Library (`-shared`)
```

- **Flag**: `-shared`
- **Usage**: Takes `.o` files and turns them into a `.so` file.

```
gcc -shared matrix.o vector.o -o liblinalg.so
```

3. Static Linking (`-static`)

- **Flag**: `-static`
- **Usage**: Forces gcc to bundle all system libraries (including glibc) into your executable.
- **Result**: The binary becomes huge (from ~20KB to 800KB+), but it is completely portable. It will run on a Linux machine even if that machine has no libraries installed.

```
gcc -static main.c -o portable_app
```

Phase 6: Architecture & Hardware Control

In kernel development, you often cross-compile (compile on x64 but for an ARM embedded board) or need to strip the code down to the bare metal.

1. Architecture Specifics

- `-m32` : Compile a 32-bit binary on a 64-bit machine (great for learning stack overflows and legacy memory mapping).
- `-m64` : Force 64-bit (usually default).
- `-march=native` : Tells gcc to detect your specific CPU (e.g., Intel i7 Alder Lake) and use special instructions (AVX2, AVX-512) that are fast but won't run on other computers. Great for your local Linear Algebra benchmarks.

2. The "Freestanding" Environment (`-ffreestanding`)

- **Context**: Used in kernel and OS development.
- **Effect**: Tells the compiler: "There is no Standard C Library (libc) here." It disables built-in optimizations that assume functions like `memcpy` or `printf` exist. You must provide them yourself.

```
gcc -ffreestanding -c kernel.c -o kernel.o
```

3. Disable Builtins (`-fno-builtin`)

- **Effect**: GCC often replaces standard functions like `printf("Hello")` with `puts("Hello")` automatically for speed. If you are writing your own OS and implementing `printf` yourself, this creates a conflict. This flag stops GCC from interfering.

Phase 7: Security Hardening (Cryptography Context)

Since you are interested in cryptography, you need to know how GCC protects (or exposes) memory.

1. Stack Canaries (`-fstack-protector`)

- **Concept**: GCC inserts a "canary" (a random integer) on the stack before the return address. If a buffer overflow overwrites the canary, the program crashes immediately instead of allowing code execution (preventing hacks).
- **Flags**:
  - `-fstack-protector-all` : Protects all functions (slower, but safer).
  - `-fno-stack-protector` : Turns it off (dangerous! Used when learning how to exploit buffer overflows).

2. ASLR Support (`-pie` / `-fPIE`)

- **Flags**: `-fPIE` (for compilation) and `-pie` (for linking)
- **Effect**: Compiles the executable so it can be loaded at random memory locations by the OS, making it harder for attackers to predict memory addresses.

Phase 8: Dependency Management (The "Pro" Move)

When you write a Makefile, you have a problem: If you change `header.h`, GCC doesn't know it needs to recompile `main.c` (which includes it).

- **Flag**: `-MMD` (or `-MD`)
- **Effect**: While compiling, GCC looks at all the `#include` statements and generates a `.d` dependency file. This file lists exactly which header files your C file depends on.
- **Usage**: Include the generated `.d` files in your Makefile so builds update automatically when headers change.

Phase 9: Verbose & Dry Runs

Sometimes GCC fails, and you don't know why (e.g., it's looking in the wrong folder for libraries).

- `-v` (Verbose): Prints everything GCC is doing: exact paths, version numbers, and the long internal commands it calls.
- `-###` (Dry Run): Prints the commands GCC would run, but doesn't actually execute them. Great for debugging configuration issues.

Summary for your 4th Sem Project (Linear Algebra Lib)

For your library project, your compilation pipeline will likely look like this:

1. Compile Object Files (with PIC for shared lib)

```
gcc -Wall -Wextra -O3 -march=native -fPIC -c matrix_ops.c -o matrix_ops.o
```

2. Create Shared Library

```
gcc -shared matrix_ops.o vector_ops.o -o liblinalg.so
```

3. Link against it (for testing)

```
gcc main.c -L. -llinalg -o main
```

(Note: `-L.` tells gcc to look in the current folder for the library. You may also need `-Wl,-rpath=.` for runtime linking if running from the build directory.)