#include <stdio.h>

int main() {

printf("Hello, World!\n");

return 0;

}

1. Preprocessing

The preprocessor interprets the `#include <stdio.h>` line and includes the content of the `stdio.h` header file.

You can see the result of preprocessing by running `gcc -E main.c -o main.i` on a Unix-like system. This will generate `main.i` which contains the preprocessed code.

2. Compilation

Next, the compiler translates the preprocessed C code into assembly instructions.

You can see the result of compilation by running `gcc -S main.i -o main.s`. This will compile `main.i` into assembly code, which will be saved as `main.s`.

3. Assembly

The assembler now translates the assembly code into machine instructions (object code).

You can perform the assembly step with `gcc -c main.s -o main.o`. This assembles `main.s` into machine code and saves it in the object file `main.o`.

4. Linking

Next, the linker takes `main.o` and links it with the necessary libraries to create an executable file.

Perform the linking step with `gcc main.o -o main`. This will create the executable `main`.

5. Loading

Finally, when you run `./main` on a Unix-like system, the loader loads the `main` program into memory and starts its execution.

The above process of manually running each stage is typically used for educational or debugging purposes. In a production environment, you usually run the `gcc main.c -o main` command, which automatically performs all these stages.

For this example, let's create a simple program that uses both a dynamic library (also known as a shared library or DLL on Windows) and a static library. Our program will call a function defined in each library. This example will use Unix-like syntax, since that's most common for explaining low-level details.

First, let's define our libraries:

Static Library

Create a file `add.c`:

c

int add(int a, int b) {

return a + b;

}

Compile `add.c` to an object file, then archive it into a static library:

gcc -c add.c -o add.o

ar rcs libadd.a add.o

This creates a static library named `libadd.a` that exports a function `add`.

Dynamic Library

Create a file `sub.c`:

int sub(int a, int b) {

return a - b;

}

Compile `sub.c` into a shared library:

gcc -shared sub.c -o libsub.so

This creates a shared library named `libsub.so` that exports a function `sub`.

Main Program

#include <stdio.h>

int add(int a, int b);

int sub(int a, int b);

int main() {

int a = 5, b = 3;

printf("%d + %d = %d\n", a, b, add(a, b));

printf("%d - %d = %d\n", a, b, sub(a, b));

return 0;

}

Compile and link `main.c` with both libraries:

gcc main.c -L. -ladd -lsub -o main

The `-L.` option tells gcc to look in the current directory for libraries, and the `-ladd` and `-lsub` options tell it to link with `libadd.a` and `libsub.so`.

You can then run the program with `./main`.

Dynamic Linking and Loading

When you run the program, the dynamic linker/loader will be involved to handle `libsub.so`. Here's roughly what it does:

1. The loader loads the `main` executable into memory.

2. It sees that the executable needs `libsub.so`, so it finds this library and loads it into memory as well.

3. It adjusts the program's memory image so that calls to `sub` go to the correct location in `libsub.so`.

4. It starts the program running.