Give me the step by step c commands considering there is 2 library files which needs during linking and main files and whatever required. Now give c code from the starting pre processing to linking for static libraries and dynamic libraries

Got it — you want the **full pipeline**, from **preprocessing → compilation → assembling → linking**, with **two library files**, shown for **both static and dynamic (shared) libraries**, plus **actual C code** and the **exact commands**. Let's do this clean, step-by-step, like you'd see in a systems programming or OS class.

I'll assume:

- OS: Linux
- Compiler: `gcc`
- Two libraries: `libmath1` and `libmath2`
- One main program: `main.c`

---

1 Project structure

```css
project/
│
├── main.c
│
├── math1.c
├── math1.h
│
├── math2.c
├── math2.h
```

---

2 Source code

math1.h

```c
#ifndef MATH1_H
#define MATH1_H

int add(int a, int b);

#endif
```

math1.c

```c
#include "math1.h"

int add(int a, int b) {
    return a + b;
}
```

---

math2.h

```c
#ifndef MATH2_H
#define MATH2_H

int multiply(int a, int b);

#endif
```

math2.c

```c
#include "math2.h"

int multiply(int a, int b) {
    return a * b;
}
```

main.c

c Copy code

```c
#include <stdio.h>
#include "math1.h"
#include "math2.h"

int main() {
    int x = 10, y = 5;

    printf("Add: %d\n", add(x, y));
    printf("Multiply: %d\n", multiply(x, y));

    return 0;
}
```

---

3 Compilation stages (conceptual view)

Every `.c` file goes through:

1. **Preprocessing** → `.i`
2. **Compilation** → `.s`
3. **Assembly** → `.o`
4. **Linking** → executable

You normally don't do all manually, but I'll show you **explicit commands**.

---

4 STATIC LIBRARY (.a)

Step 1: Preprocessing

bash Copy code

```bash
gcc -E math1.c > math1.i
gcc -E math2.c > math2.i
gcc -E main.c  > main.i
```

Step 2: Compilation (to assembly)

bash Copy code

```
gcc -S math1.i
gcc -S math2.i
gcc -S main.i
```

Produces:

css Copy code

```
math1.s  math2.s  main.s
```

Step 3: Assembly (to object files)

bash Copy code

```
gcc -c math1.s
gcc -c math2.s
gcc -c main.s
```

Produces:

css Copy code

```
math1.o  math2.o  main.o
```

Step 4: Create STATIC libraries

bash Copy code

```
ar rcs libmath1.a math1.o
ar rcs libmath2.a math2.o
```

Step 5: Linking (static)

bash Copy code

```
gcc main.o -L. -lmath1 -lmath2 -o static_app
```

🔹 `-L.` → look in current directory
🔹 `-lmath1` → finds `libmath1.a`
🔹 `-lmath2` → finds `libmath2.a`

Step 6: Run

bash Copy code

```
./static_app
```

✅ All library code is **copied into the executable**

5️⃣ DYNAMIC (SHARED) LIBRARY (.so)

Now we do the same thing, but properly for shared libraries.

Step 1: Compile with Position Independent Code (PIC)

bash Copy code

```
gcc -fPIC -c math1.c
gcc -fPIC -c math2.c
```

Produces:

Copy code

```
math1.o  math2.o
```

Step 2: Create SHARED libraries

bash Copy code

```
gcc -shared -o libmath1.so math1.o
```

```
gcc -shared -o libmath2.so math2.o
```

Step 3: Compile main program

bash Copy code

```
gcc -c main.c
```

Step 4: Linking (dynamic)

bash Copy code

```
gcc main.o -L. -lmath1 -lmath2 -o dynamic_app
```

At this point the executable **references** the `.so` files — it does not copy them.

Step 5: Runtime linker path

Option A: Temporary (recommended for testing)

bash Copy code

```
export LD_LIBRARY_PATH=.
./dynamic_app
```

Option B: Embed rpath (cleaner)

bash Copy code

```
gcc main.o -L. -lmath1 -lmath2 -Wl,-rpath=. -o dynamic_app
```

Step 6: Verify dynamic linking

bash Copy code

```
ldd dynamic_app
```

You'll see:

```ini
libmath1.so => ./libmath1.so
libmath2.so => ./libmath2.so
```

---

6 Static vs Dynamic summary

Feature	Static (.a)	Dynamic (.so)
Linked at	Compile time	Run time
Executable size	Larger	Smaller
Dependency	None at runtime	`.so` required
Speed	Slightly faster startup	Slight overhead
Memory sharing	❌	✅

If you want, next I can:

- Show `nm`, `objdump`, `readelf` output
- Explain **symbol resolution order**
- Show **what breaks if library order is wrong**
- Do the same thing with **Makefile**

Just say the word 👌