Modular Programming

Splitting into Multiple files.

Using Makefiles.

Pre-Class Work

Read, understand, implement and test functions in the code:

https://onlinegdb.com/fOgZ6HXyKw

You may need to download this code locally (copy paste to local c file).

Make sure your implementation pass all the test cases (option 15)

Installing GDB and Make

See directions here

🧩 Why Modularize Code?

Problem:

- Large C programs become hard to manage when everything is in one `.c` file.
- Difficult to debug, maintain, and reuse.

Solution:

- Split code into **multiple files**, each handling a clear responsibility.
- Use **headers** (`.h`) for declarations and **source files** (`.c`) for definitions.

Benefits:

- Easier to understand and modify.
- Enables **team collaboration**.
- Promotes **reusability** and **faster compilation**.

🎯 Goal

We'll start with a small C program written in a **single file**, and gradually convert it into a **multi-file modular program** built using a **Makefile**.

At each step, we'll discuss the **benefits** and **improvements** introduced.

Step 0: Single File (Unmodularized)

Let's start with a simple C program that performs arithmetic operations.

`calculator.c`

```c
#include <stdio.h>

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

int subtract(int a, int b) {
    return a - b;
}

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

int divide(int a, int b) {
    if (b == 0) {
        printf("Error: Division by zero!\n");
        return 0;
    }
    return a / b;
}

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

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

Step 1: Split Logic into Separate Files

We move all arithmetic logic into a new source file.

`math_utils.c`

```c
#include <stdio.h>

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

int subtract(int a, int b) {
    return a - b;
}

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

int divide(int a, int b) {
    if (b == 0) {
        printf("Error: Division by zero!\n");
```

Step 2: Introduce a Header File

We define all function **prototypes** in a `.h` file.

`math_utils.h`

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
int divide(int a, int b);

#endif
```

Update `main.c`

```
#include <stdio.h>
#include "math_utils.h"
```

Step 3: Organize into Folders

Let's move to a standard directory structure:

```
calculator/
├── include/
│   └── math_utils.h
├── src/
│   ├── main.c
│   └── math_utils.c
```

Now compile with include path:

```
gcc -I./include src/main.c src/math_utils.c -o calculator
```

✅ **Advantage:**

- Follows real-world conventions.
- Keeps headers, sources, and binaries neatly separated.

🧰 Compiling Multi-File Projects Manually

```
gcc -c src/math_utils.c -o src/math_utils.o
gcc -c src/main.c -o src/main.o
gcc src/main.o src/math_utils.o -o program
```

- `-c` → compile to object file (no linking yet).
- The final command **links** object files into an executable.

🏗️ Why Use a Makefile?

Without a Makefile:

- You must retype all commands each time.
- Every file recompiles even if unchanged.

With a Makefile:

- Automates the build process.
- Rebuilds **only changed files**.
- Ensures consistent compiler flags.

Step 4: Add a Makefile

We automate the build.

`Makefile`

```
CC = gcc
CFLAGS = -Wall -I./include -g
SRCS = src/main.c src/math_utils.c
OBJS = $(SRCS:.c=.o)
TARGET = calculator

all: $(TARGET)

$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $@ $(OBJS)

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

clean:
```

🎯 Step 5: Scaling Up — Adding More Modules

Now that our calculator is modular and buildable with a Makefile, let's **extend** it to perform:

- **Input/Output** (handled by `io_utils`)
- **Statistics** (mean, variance) — handled by `stats`

We'll see how to integrate these **new modules** seamlessly into the existing structure.

🧱 Updated Directory Structure

```
calculator/
├── include/
│   ├── math_utils.h
│   ├── io_utils.h
│   └── stats.h
├── src/
│   ├── main.c
│   ├── math_utils.c
│   ├── io_utils.c
│   └── stats.c
└── Makefile
```

💻 io_utils Module — Handling Input and Output

include/io_utils.h

```
#ifndef IO_UTILS_H
#define IO_UTILS_H

void get_two_numbers(int *a, int *b);
void print_result(const char *operation, int result);

#endif
```

src/io_utils.c

```
#include <stdio.h>
#include "io_utils.h"

void get_two_numbers(int *a, int *b) {
    printf("Enter two integers: ");
    scanf("%d %d", a, b);
```

📊 stats Module — Basic Statistical Calculations

`include/stats.h`

```c
#ifndef STATS_H
#define STATS_H

double mean(int arr[], int n);
double variance(int arr[], int n);

#endif
```

`src/stats.c`

```c
#include <stdio.h>
#include "stats.h"

double mean(int arr[], int n) {
    double sum = 0;
    for (int i = 0; i < n; i++)
```

🧠 Updated `main.c`

`src/main.c`

```c
#include <stdio.h>
#include "math_utils.h"
#include "io_utils.h"
#include "stats.h"

int main() {
    int a, b;
    get_two_numbers(&a, &b);

    print_result("Addition", add(a, b));
    print_result("Subtraction", subtract(a, b));
    print_result("Multiplication", multiply(a, b));
    print_result("Division", divide(a, b));

    int data[5] = {a, b, a + b, a - b, a * b};
    printf("\nStats on sample data: ");
    printf("\nMean = %.2f", mean(data, 5));
    printf("\nVariance = %.2f\n", variance(data, 5));

    return 0;
```

⚙️ Updated Makefile

`Makefile`

```
CC = gcc
CFLAGS = -Wall -I./include -g
SRCS = src/main.c src/math_utils.c src/io_utils.c src/stats.c
OBJS = $(SRCS:.c=.o)
TARGET = calculator

all: $(TARGET)

$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $@ $(OBJS)

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f $(OBJS) $(TARGET)
```

🚀 Build and Run

```
make
./calculator
```

Sample Output:

```
Enter two integers: 10 5
Addition result = 15
Subtraction result = 5
Multiplication result = 50
Division result = 2

Stats on sample data:
Mean = 17.40
Variance = 254.24
```

🧩 Advantages of Scaling Up

Challenge	Modular Solution
More functionality	Add new `.c` and `.h` files easily
Maintenance	Modify one module without breaking others
Reuse	`io_utils` and `stats` can be used in other projects
Build complexity	Makefile handles it automatically
Teamwork	Different modules can be owned by different developers

🧭 Recap: Incremental Journey

Step	Change	Key Takeaway
0	Single File	Simple but messy
1	Split into `.c` files	Logical separation
2	Added `.h` files	Centralized declarations
3	Folder structure	Organized and scalable
4	Added Makefile	Automated builds
5	Multiple modules	Extend functionality easily
6	Added I/O + Stats	Real modular project ready for teamwork

✅ Final Thoughts

Modular programming is not just about organization — it’s about:

- **Reusability**
- **Maintainability**
- **Team collaboration**
- **Faster builds and debugging**

With a Makefile and clear module boundaries, your C projects can grow without becoming chaotic.

🧭 Example in Practice

Project: Mini Social Network

```
mini_social_network/
├── include/
│   └── network.h
│   └── friendships.h
├── src/
│   ├── main.c
│   ├── network.c
│   └── friendships.c
└── Makefile
```

Each file handles one major concern:

- **network.c** → Core data structures and file I/O