Introduction

As the size of a program grows, it becomes increasingly difficult to manage, especially when multiple functionalities are being implemented. To address this complexity, the program is split into modules, which are logical abstractions that allow developers to compartmentalize their code.

In C, a function can be considered a single module. A single module may have its own individual file, or it can be included along with other modules in a common file. This approach not only provides a logical separation of functionalities but also enables code reuse. For executing a particular functionality, the module can be invoked repeatedly.

Let's illustrate this with a simple calculator program that performs basic arithmetic operations like addition, subtraction, multiplication, and division. Imagine trying to implement all these functionalities within the main() function. The resulting code would be difficult to understand, maintain, and extend. By making the program modular, the solution becomes easy to understand, maintain, enhance, and reuse.

Interface File, Implementation File, Driver File, and Makefile

A well-defined, easy-to-understand, and easy-to-execute structure is crucial for modular programs. To achieve this, the program is split into four types of files: interface file, implementation file, driver file, and Makefile.

Interface File (calculator.h)

The interface file, also known as the header file in C, lists all the method signatures that the program contains. It can also define values of certain constants and the Abstract Data Types (structs) that will be used in the code. For example:

```
#ifndef CALCULATOR_H
#define CALCULATOR_H

// Function declarations
extern int add(int a, int b);
extern int subtract(int a, int b);
extern int multiply(int a, int b);
extern double divide(int a, int b);

#endif /* CALCULATOR_H */
```

Always use header guards in all the header files that you write. You can read more about them here: https://www.learncpp.com/cpp-tutorial/header-guards/

Implementation File (calculator.c)

The implementation file includes the interface file and implements the functions defined in the interface file. Make sure that you import the interface file using #include in your implementation file. For example:

```
#include "calculator.h"

// Function definitions
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;
}

double divide(int a, int b) {
    if (b != 0) {
        return (double) a / b;
    } else {
        return 0; // Handle division by zero appropriately
    }
}
```

Driver File (main.c)

The driver file contains the driver module and should be kept as short as possible. It should only contain the main function and call other modules to execute operations. For example:

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

int main() {
    int a = 10, b = 5;

    printf("Addition: %d + %d = %d\n", a, b, add(a, b));
    printf("Subtraction: %d - %d = %d\n", a, b, subtract(a, b));
    printf("Multiplication: %d * %d = %d\n", a, b, multiply(a, b));
    printf("Division: %d / %d = %.2f\n", a, b, divide(a, b));
```

```
    return 0;
}
```

Creating Linkable Modules

Once all the above files have been implemented, we need to create linkable modules from these files. A linkable module contains machine-level code of a standalone file (which may implement one or more functions). These modules are compiled object files (.o files) that can be linked together to form an executable program.

For example, let's compile our calculator program's files into linkable modules:

```
gcc -c calculator.c   # Compiles calculator.c into calculator.o
gcc -c main.c         # Compiles main.c into main.o
```

In this case, calculator.o and main.o are the linkable modules.

Creating the Executable

After creating the linkable modules (object files), we need to combine them to produce a single executable file. The executable file is the final output of the compilation process and can be run directly on the system.

To create the executable, we link the object files together:

```
gcc -o calculator main.o calculator.o
```

This command creates an executable file named calculator by linking main.o and calculator.o.

Makefile

The Makefile automates the compilation process, specifying how to build the object files and link them into the executable. For example:

```
# Variable declaration
executableName = calculator

# Targets and recipes
all: $(executableName)

$(executableName): main.o calculator.o
        gcc -o $(executableName) main.o calculator.o
```

```
main.o: main.c calculator.h
	gcc -c main.c

calculator.o: calculator.c calculator.h
	gcc -c calculator.c

clean:
	rm -f *.o $(executableName)
```

The Makefile begins with the declaration of a variable `executableName`, which is set to `calculator`. This variable is used to refer to the name of the executable throughout the Makefile.

The `all` target is defined as the default build target, which depends on the executable specified by `$(executableName)`. The next target specifies that the executable (`calculator`) depends on two object files: `main.o` and `calculator.o`. The recipe for this target is to use the `gcc` command to link these object files and produce the executable. Specifically, `gcc -o $(executableName) main.o calculator.o` translates to `gcc -o calculator main.o calculator.o`, creating the final executable file.

Further, the Makefile defines individual targets for `main.o` and `calculator.o`, specifying their dependencies and how to compile them. The `main.o` target depends on `main.c` and `calculator.h`, and the recipe is to compile `main.c` using `gcc -c main.c`. Similarly, the `calculator.o` target depends on `calculator.c` and `calculator.h`, and the recipe is to compile `calculator.c` using `gcc -c calculator.c`.

Lastly, the `clean` target is defined to remove all object files and the executable, with the recipe `rm -f *.o $(executableName)`, ensuring that a fresh build can be performed by cleaning up previous build artifacts.

This structure provides a clear and automated way to build the project, compile the necessary files, link them to create the executable, and clean up the build environment when needed.

For a tutorial on Makefile, you can refer to:
https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Building and Running the Program

Open a terminal and navigate to the directory containing the files.
Run make to compile the program and create the executable.
Run ./calculator to execute the program.
Run make clean to remove the compiled object files and the executable.

This structure ensures that the program is modular, easy to understand, maintain, and extend. By using modular design and a Makefile, we ensure that the program is maintainable and scalable, making it easier to manage as it grows in size and complexity.

Abstract Data Types and Modular Programming

In C, ADTs are typically implemented using structs to define the data and functions to define the operations. By combining ADTs with modular programming, we can create well-structured, maintainable, and reusable code. Each ADT can be implemented as a separate module, with its own header and implementation files. The driver file can then include these headers and use the ADTs as needed. This approach ensures a clear separation of concerns, making the program easier to understand and extend.

For instance, in the bank account management system, we could have separate modules for handling bank accounts, transactions, and user interactions, each with its own ADTs and associated operations. This modular approach not only simplifies development and maintenance but also enhances the robustness and scalability of the software.

Read about structs here:
https://www.w3schools.com/c/c_structs.php

Let's take a look at a modular calculator program with a struct:

Interface File (calculator.h)

For the calculator program, we define a struct to hold the operands and the results of operations. The file lists all the method signatures and any constants or macros used across the modules.

```
#ifndef CALCULATOR_H
#define CALCULATOR_H

// Define the ADT for a calculator operation
typedef struct {
    int operand1;
    int operand2;
    int result;
    char operation; // '+', '-', '*', '/'
} Calculator;

// Function declarations
void initializeCalculator(Calculator* calc, int a, int b, char op);
void performOperation(Calculator* calc);
```

```
void printResult(const Calculator* calc);

#endif /* CALCULATOR_H */
```

Implementation File (calculator.c)

The implementation file includes the interface file and provides the actual definitions for the functions declared in the header file. It contains the logic to initialize the calculator, perform the specified operation, and print the result.

```
#include "calculator.h"

#include <stdio.h>

void initializeCalculator(Calculator* calc, int a, int b, char op) {

    calc->operand1 = a;

    calc->operand2 = b;

    calc->operation = op;

}

void performOperation(Calculator* calc) {

    switch (calc->operation) {

        case '+':

            calc->result = calc->operand1 + calc->operand2;

            break;

        case '-':

            calc->result = calc->operand1 - calc->operand2;

            break;

        case '*':

            calc->result = calc->operand1 * calc->operand2;

            break;

        case '/':
```

```
            if (calc->operand2 != 0) {
                calc->result = calc->operand1 / calc->operand2;
            } else {
                printf("Error: Division by zero\n");
                calc->result = 0;
            }
            break;
        default:
            printf("Error: Unknown operation\n");
            calc->result = 0;
    }
}
void printResult(const Calculator* calc) {
    printf("Result: %d %c %d = %d\n", calc->operand1, calc->operation, calc->operand2,
calc->result);
}
```

Driver File (main.c)

The driver file contains the `main` function and is responsible for creating instances of the ADT, initializing them, performing operations, and displaying results. It keeps the program execution logic straightforward.

```
#include <stdio.h>
#include "calculator.h"
int main() {
    Calculator calc;
    // Example usage
    initializeCalculator(&calc, 10, 5, '+');
```

```
    performOperation(&calc);
    printResult(&calc);
    initializeCalculator(&calc, 10, 5, '/');
    performOperation(&calc);
    printResult(&calc);
    return 0;
}
```

Note: Notice pass by reference instead of pass by value.

Makefile

The Makefile automates the build process, specifying how to compile the source files and link them into the executable.

```
# Variable declaration
executableName = calculator
# Targets and recipes
all: $(executableName)
$(executableName): main.o calculator.o
	gcc -o $(executableName) main.o calculator.o
main.o: main.c calculator.h
	gcc -c main.c
calculator.o: calculator.c calculator.h
	gcc -c calculator.c
clean:
	rm -f *.o $(executableName)
```