Reading Material for Abstract Data Types

As Part of

Operating Systems [CS F372] Course

Semester I, 2022 - 2023

Edited by

BIJU K RAVEENDRAN, Dept. of CS & IS

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI –K K BIRLA GOA CAMPUS

INDEX

Sec. No.	Section Title	Page No.
1.	Introduction to modular code	3
2.	Interface File, Implementation File, Driver File and Makefile	3
3.	Homework problem	7

Introduction to modular code

As the size of a program grows, it becomes more and more difficult to manage, especially if multiple functionalities are being implemented in the program. To deal with the increasing complexity, the program is split into modules which are logical abstractions that allow the developers to compartmentalize their code.

Apart from providing a logical separation of functionalities, modules also enable code reuse. For executing a particular functionality, the module can be invoked again and again.

In C, a function can be considered as a single module. As we shall see, a single module may have its own individual file or it can be included along with other modules in a common file.

For example, let us consider a bank account management software. The software lets users check their account balance, make transactions, see their transaction history and request the bank for a new cheque book. Imagine how tricky things would get if we try to implement all these functionalities within main() function. Even if we could do it, no one else would be able to understand our code or add additional features or debug errors as and when required. Instead, if the software provided by us is modular, then the solution will be easy to understand, maintain, enhance and reuse.

Interface File, Implementation File, Driver File and Makefile

It is critical that our modular programs have a well-defined, easy to understand and easy to execute structure.

Imagine that you and your team have developed a bank account management software that has 50,000 lines of C code and tens of modules. How would you ensure that you will be able to make sense of the code when your customers report bugs 5 years later? How would you let your customers know about all the functionalities you are offering without showing them your source code and without confusing them with all the myriad features?

To do this, you would need to split up your program into four types of files: interface file, implementation file, driver file and Makefile.

Interface file - This file is also called header file in C. It simply lists down all the method signatures that your program contains. It can also contain values of certain constants that you need to use in different methods. An interface file can also define the Abstract Data Type (implemented using structs in C) that you will use in the code.

For example you have three methods and are defining one ADT as follows:

```
#ifndef INTERFACE_FILE_H
#define INTERFACE_FILE_H

//Defines the ADT of a bank account
typedef struct{
        char accountNumber[20];
        Int accountBalance;
        int atmPin;
        int chequeBookRequested:
}bankAccount;

//Returns the account balance of the given bank account number.
extern int getAccountBalance(bankAccount* bA);

/* Requests for a new cheque book for the given bank account number.
   Returns 1 if request is successful; 0 otherwise.  */
extern int requestChequeBook(bankAccount* bA);

/* Changes the ATM pin for the given account number from oldATMPin to
newATMPin. Returns newATMPin if successful and oldATMPin otherwise. */
extern int changeATMPin(bankAccount* bA, int oldATMPin, int newATMPin);

#endif /*INTERFACE_FILE_H*/
```

Note: 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/

Note: Remember that struct type variables have to be passed as pointers to functions in C. If you don't, then the struct will be passed by value and any modifications to it inside the function will not reflect in the new copy of the struct that was created for the function. This new copy is destroyed after the function exits (Revising structs, pointers, pass by value and pass by reference from your CP notes will help to clear up doubts.)

Implementation file - Implementation files implement one or more functions that you have defined in the interface file. Make sure that you import the interface file using #include in your implementation file. Otherwise the ADTs that you have defined in the interface file will not be understood by the compiler while compiling the implementation file.

Driver file - This file contains the driver module. It should be kept as short as possible and it shouldn't contain any other function apart from int main(int argc, char *argv[]). Wherever required, it should call other modules to execute operations.

Once all the above files have been implemented, we need to create linkable modules from the files. A linkable module contains machine level code of a standalone file (which may implement one or more functions). We can combine multiple linkable modules with the driver to produce a single executable file. You will be studying about linkable modules and the various phases of compilation in much more detail in your Principles of Programming Languages Course.

Continuing with our above example, suppose in the initial stages of development, we have only implemented getAccountBalance() and requestChequeBook() functions in separate modules. Later if we decide to develop a new module which implements changeATMPin and other ATM related functions as required.

Create linkable modules:

```
gcc -c accountBalanceModule.c
gcc -c chequeBookModule.c
gcc -c driverModule.c
```

Create executable:

```
gcc -o bankAccountManagement.out driverModule.o accountBalanceModule.o chequeBookModule.o
```

Run the executable:

```
./bankAccountManagement.out
```

This mechanism provides separation of:

(A) Interface and implementation
(B) Implementations of different modules
(C) Executable file and implementations of modules

This multi-stage separation makes the code much easier to understand, debug and extend. For example, the bank is now introducing Demat accounts for trading in the stock market.
We can introduce a new ADT for Demat accounts which tracks total invested amount, number of shares of different companies, total profit, etc. We can have separate modules that implement functionalities for Demat accounts.
This way, we also protect our source code of the implementation by giving only the interface file and the object files of the implementation files to the bank. By doing so, the bank will not be able to see the implementation of our code, but will be able to write a driver according to its own needs as it will be aware of all the datatypes and method signatures that are available from the interface file. The bank can then link its own driver with the object files provided by us and generate an executable to use.

Makefile - The above operations of creating linkable modules and executables from them using the gcc command on the Linux terminal can get quite repetitive. Moreover, if you have tens of modules and only make changes to a few, you only need to produce object files for these modules again.

We can automate this by creating Makefile. A Makefile is a recipe for creating the executable file. It contains bash commands that can be executed on the Linux terminal. We only need to execute the Makefile to carry out all the above described steps right from creating the linkable modules to executing the program. The Makefile is executed by simply executing ‘make’ on the terminal.

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

For our above described example the Makefile will be as follows:

```
# Variable declaration.
executableName = bankAccountManagement.out
driver = driverModule

# Makes
all: $(driver).o bankAccountModule.o chequeBookModule.o
```

```
	gcc $(driver).o bankAccountModule.o chequeBookModule.o -o $(executableName)

$(driver).o: $(driver).c
	gcc -c  $(driver).c

bankAccountModule.o: bankAccountModule.c
	gcc -c bankAccountModule.c
chequeBookModule.o: chequeBookModule.c
	gcc -c chequeBookModule.c
# To clean the compilation.
clean:
	rm -f *.o $(executableName)
```

We have provided you with TWO practical examples of interface file, implementation file, driver and Makefile for program that can perform [(1) linear search and binary search. (2) BST]. Go through the given examples, run it and understand all the nitty gritties of this approach to programming. Henceforth for all OS assignments, you will be required to adhere to this method of programming. Single code file submissions will not be considered for evaluation.

Homework problem

Implement a program that:

Takes an array as input from a file [Take name of the file as command line argument].

Takes a command line argument which is the algorithm that the user wants to use for sorting. It could be either bubble sort, insertion sort, merge sort or quicksort.

Implement each of the sorting algorithms as a separate module along with a driver module.

Sort the array and store it in a file [Get the name of the file from user].

You are supposed to use File operations [not redirections] to solve this problem. Please refer your CP notes / online resources for revising File operations in C.

File Operations to Study for the Lab

1. Command line arguments
2. fopen, fclose
3. fscanf, fprintf, fread, fwrite
4. fseek, ftell, rewind, fgetpos, fsetpos