Modular Programming in C

1 MODULES

- Aim is that independent functionality should be present in independent modules i.e. think how can the same data structure be used in other algorithms without changing any existing code
- Each modular program typically has following components
  - Interface Files
  - Implementation Files
  - Use Files
  - Driver File
- Execution should start from Driver file to test the usage given in Use File, which in turn should call the code in Implementation File

- Interface File(s):
  - Header file of Implementation and Use File(s)
  - In included by Use files or Driver file
  - Should contain all function declarations and global variable declarations

- Implementation File(s):
  - Source code (definition) file containing all logic and implementation
  - No main function
  - Should contain all function definitions and global variable definitions
  - Should also handle all required memory allocation here
  - Typically contains the data structure implementation

- Use File(s):
  - Source code to use the implementation to do something logical
  - Typically contains the code to use data structures to implement an algorithm
  - No main function

- Driver File:
  - A single *.C file containing (preferably only) main function
  - Should not contain any logic implementation, only testing code
  - Preferably no memory allocation calls
  - Can have support functions to encapsulate testing logic (minimal inputs required to test an algorithm implemented in Use file)

2 MULTI-FILE COMPILATION

2.1 PROCESS

- Compile and debug each independent module separately using -c flag of gcc
- Link object files of all modules together to generate a single executable program

2.2 COMMANDS

- Compiling

```
gcc  -c  module1.c  -o  module1.o
gcc  -c  module2.c  -o  module2.o
gcc  -c  module3.c  -o  module3.o
```

// -c flag tells the compiler to compile and produce only an object file
// -o flag tells the compiler the name of output file of this command i.e. name of object file

- Linking

```
gcc module1.o module2.o module3.o  -o myprogram
```

- Execution

```
./myprogram
```

2.3 USE OF EXTERN KEYWORD

- If a global variable which is defined in another implementation (let's say, source.c) file needs to be used in the current (let's say, useit.c file), then use "extern" keyword to *declare* it in useit.c file, as follows:

```
extern int globalCounter;          //don't give an initial value here
```

2.4 USE OF #IFDEF TO RESOLVE MULTIPLE INCLUSION

- To resolve multiple inclusion error, for each header file, think of a global name. Typically, it is an underscore prefixed to the name of file in all-caps, like _MYHEADER
- Move all contents of header file between following definitions

```
#ifndef  _MYHEADER
#define  _MYHEADER

        <content of old header file comes here>
#endif
```

- It will stop compiler from including content of this header file is name _MYHEADER has already been defined (included) by any other file

3 GENERAL GUIDELINES

- Main's return type should be int and it should return 0 to indicate normal termination, non-zero otherwise.
- Global variables should be avoided (instead, variables should be passed by reference to functions)
- Memory allocation
  - No static memory allocation for collections (array, list, trees, etc.)
  - Only required memory should be allocated and unused should be freed, unless there is a better allocation algorithm in place
- Pointer to the data structure to be modified should be passed as argument to the function modifying/using it
- Code should always be indented, more so in case of hurry

3.1 CODING GUIDELINES

- One of the ways to write a big code is to
  - Divide it into logical modules
  - Define interface for each module i.e. top-level function headers
  - Keep I/O module separate from rest of the algorithm. Implement and test it first.
  - Execute the program with user I/O from dummy main function to verify the flow through dummy (empty) function definitions
  - Pick each module one by one
  - Define function wise, module wise, and program wise test cases before implementing each one of them
  - Implement each function to do only one job
  - Top level function can call multiple small functions to implement the whole logic i.e. logic flow of the module should be clear by looking at the code of one top level function
  - Do not start implementing a second module while a dependent module has bugs. At least, isolate the bug.

3.2 DEBUGGING GUIDELINES

- One of the ways to debug a code is to
  - Accept that a mistake has been made
  - Check if the bug is reproducible i.e. on what exact inputs it gives what exact error?
  - Use a debugger (GDB is usually installed everywhere). If not, then add a dummy scanf statement after every printf statement to correctly get the flow, especially in case of segmentation fault.
  - Eliminate correct code one by one in order of its execution. If the program is a tree and each statement is a branch and execution happen from left to right, then prune the tree from left to right by checking the output (status of variables) after each branch has been executed. When a branch (function call) is detected to be faulty, then go deeper and repeat the procedure.