Birla Institute of Technology and Science, Pilani
CS F211, Data Structures and Algorithms
Lab #1

Objectives:

1. **Splitting your program in multiple files**

Splitting your program into multiple files

The programs you have seen so far have all been stored in a single source file. As your programs become larger, and as you start to deal with other people's code (e.g. other C libraries) you will have to deal with code that resides in multiple files. Indeed, you may build up your own library of C functions and data structures, that you can re-use in your own scientific programming and data analysis. Here we will see how to place C functions and data structures in their own file(s) and how to incorporate them into a new program.

You have already seen that one way of including custom-written functions in your C code, is to simply place them in your main source file, above the declaration of the main() function. A better way to re-use functions that you commonly incorporate into your C programs is to place them in their own file, and to include a statement above main() to include that file. When compiled, it's just like copying and pasting the code above main(), but for the purpose of editing and writing your code, this allows you to keep things in separate files. It also means that if you ever decide to change one of those re-usable functions (for example if you find and fix an error) that you only have to change it in one place, and you don't have to go searching through all of your programs and change each one.

Header files

A common convention in C programs is to write a header file (with .h suffix) for each source file (.c suffix) that you link to your main source code. The logic is that the .c source file contains all of the code and the header file contains the function prototypes, that is, just a declaration of which functions can be found in the source file.

This is done for libraries that are provided by others, sometimes only as compiled binary "blobs" (i.e. you can't look at the source code). Pairing them with plain-text header files allows you see what functions are defined, and what arguments they take (and return).

Getting started

- Log into the Linux machine and write any C program (you can write a simple C program to add two numbers.
- You can compile the file with the following command: **`gcc sourcefilename.c`**
- Observe that executable formed is called a.out. Execute the file using the command: **`./a.out`** (Here ./ implies that the path of a.out is the current directory)

Figure-1 explains about the compilation process, which in general is split into roughly 5 stages: Preprocessing, Parsing, Translation, Assembling, and Linking.

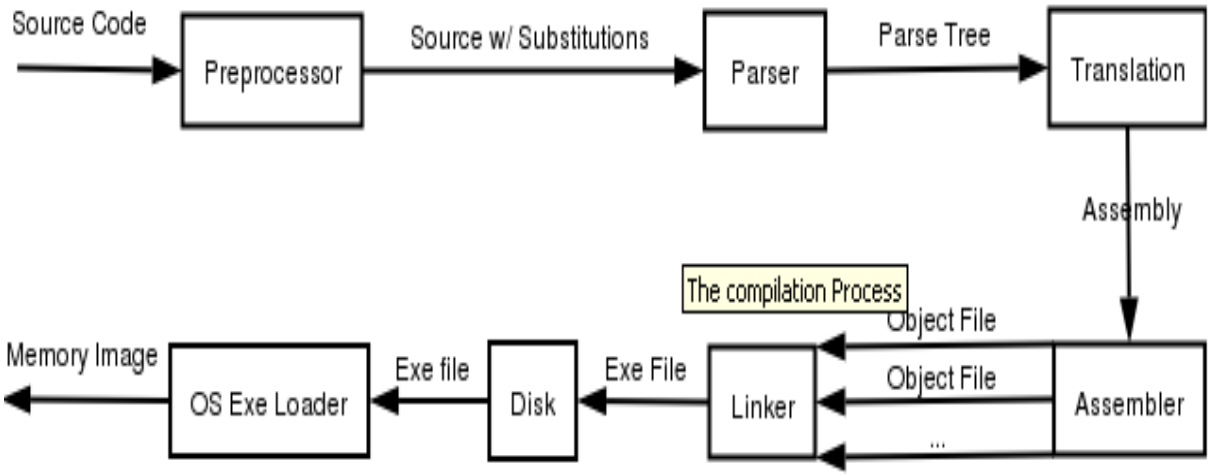

Figure-1: Compilation Process

What a compiler do?

gcc normally invokes the preprocessor, compiles the process code to assembly language code, assembles it, and then links it with the link editor. The term "compile" is a bit of an overloaded term because it is used at a high-level to mean "*convert source code to a program*", but more technically means to "*convert source code to object code*". A compiler like gcc actually performs two related, but arguably distinct functions to turn your source code into a program: compiling (as in the latter definition of turning source to object code) and linking (the process of combining the necessary object code files together into one complete executable). So, can you stop the compilation process at any of its intermediate step?? YES.

Preprocessor: handles logic behind all # directives in C and acts as a substitution engine.

- **`gcc –E`** runs only the preprocessor stage, but do not compile. This places all **`#include`** files in .c file and translates all macros.
- **`-o`** option can be used to redirect to a file.
- For example, suppose you wrote a program **`test.c`** (to add two numbers). Now, do the following:

  ```
  gcc –E test.c – o test_pre.c
  ```

  With above command you stopped the compilation process at preprocessor level and redirected the output to **`test_pre.c`** Now, using vi editor see the file **`pre_test.c`** (it is a huge file and your code to add two numbers will be the last few lines of this file).
- Similarly, you can use the following command to produce output file with a different name other than default **`./a.out`**

  ```
  gcc test.c –o output
  ```

  And, now you can execute your program with: **`./output`** (where output is the name of executable instead of a.out)

Assembler: Similarly, you can stop compilation process upto assembler and just creat object files (with extension .o); and later link multiple files together to create an executable file.

Program Structuring: modularity and reuse

So, as your program grow in size, how to manage complexity? One strategy is to divide it into modules and reuse these modules. The only difference with what you have been doing is that here these modules will be in separate files. Each module is a file or a function in C. A file module contains declarations or definitions (of data or functions). In other words, you will make separate files for "**what**" and "**how**": where

- **"what"** is known as interface; or in C, it is specified by a function declaration.
- **"how"** is known as implementation; or in C, it is specified by a function definition.

Fine. Enough Reading! Let's turn the things in action. Let's understand first use of dividing your program in multiple files. Suppose you want to write two functions which finds maximum and minimum element respectively in an array of size n. Proceed as follows:

1.	Create a file **search_max.c** and put following code in it:	int search (int arr[], int n)
		{
		int i;
		int max = arr[0];
		for (i=1;i<n;i++)
		{
		if (arr[i] > max)
		max = arr[i];
		}
		printf ("Inside function, max = %d",max);
		return max;
		}

2.	Create a file **search_min.c** and put following code in it:
	
	Observe that name of the function is same in both the files (**search_max.c** and **search_min.c)**
		int search (int arr[], int n)
		{
		int i;
		int min = arr[0];
		for (i=1;i<n;i++)
		{
		if (arr[i] < min)
		min = arr[i];
		}
		printf ("In function minimum = %d",min);
		return min;
		}

3.	Create a file **search.h** and put following code in it:	extern int search (int arr[], int n);

4.	Create a file **main.c** and put following code in it.
	
	So, now if you will compile **main.c**, which search function it will execute ??
		#include <stdio.h>
		#include "search.h"
		int main()
		{
		int a[] = {2, 4, 3, 1, 5, 9, 8, 10};
		int ele = search (a, 8);
		printf ("In function main, element = %d", ele);
		}

5.	Create an object file **search_max.o**	gcc -c search_max.c
6.	Create an object file **search_min.o**	gcc -c search_min.c
7.	Link **main.c** with the search function defined in **search_max.c** and create an executable **search1**	gcc -o search1 main.c search_max.o
8.	Execute **search1**	./search1
9.	Link **main.c** with the search function defined in **search_min.c** and create an executable **search2**	gcc -o search2 main.c search_min.o
10.	Execute **search2**	./search2

Now, suppose your friend has implemented a search function in file **search_new.c**, and he don't want to give his code to you ☹. How to deal with this situation?? *Just take the corresponding object file and use it*! This is the second use of the above process.

OK. Enough doing simple stuff! Let's do something complicated. Read along the following question; *it's a lengthy one*. Try to understand the question and implement the functions asked. For your reference, you can download the question folder from Nalanda. It has all the partially completed files. Also, if you are not able to solve, don't worry! You will learn it. For your reference, you can download the solution folder also from Nalanda. ***Important aspect is that you should understand the question, its solution, and create appropriate executable files.***

Consider a multi-programming environment, which allows upto MAX Jobs to arrive at the CPU for execution. The environment selects one job at a time and executes it. Following are two possible ways for choosing the order in which the jobs will get executed:

a) Highest Priority and Lowest Execution Time.
b) Highest Priority and Lowest Arrival Time.

This can be achieved by sorting the list of jobs according to either of the two ways and choosing the first job in the list. Each Job contains the following attributes:

1. An integer "id" representing the unique identification number of Job.
2. Priority of Job "pri" which can be either PRI_0, PRI_1 or PRI_2. Here, PRI_2 is the highest priority and PRI_0 is the lowest priority.
3. An integer "at" containing the arrival time of the job.
4. An integer "et" containing the execution time of the job.

In our implementation, an array JobList contains the list of all Jobs waiting for execution. The definitions of the structure Job and the list JobList is available in storage.h. For sorting the JobList, the designer chose to use a slightly modified version of bucket sorting. The Jobs will first be put into 3 buckets corresponding to each of the three priorities. The jobs will be inserted into the buckets based on increasing order of either arrival time or execution time of the job depending on the case. Then they will be copied back to the original JobList from bucket for PRI_2, bucket for PRI_1 and then bucket for PRI_0 in that order. To improve efficiency of this step, the buckets are created using sequential list. Thus, the jobs will be inserted such that all the Jobs with Priority 0 are stored in first SeqList, those with Priority 1 in second SeqList, and those with Priority 2 in third SeqList.

The type Job captures all the attributes of a job, as listed above. The sequential lists corresponding to each of the priorities is implemented in the storage st (Store st). The Store st is defined as the array of 'Location', where each 'Location' is a structure containing

1. A variable "ele" representing a Job.
2. An integer "next" representing the array index of next Location of SeqList.

The variable nextFreeLoc (defined globally in SeqListOps.c) holds the index of next available free location in the storage st.

A SeqList contains the following attributes:

1. An integer "head" which stores the index of the first element of the SeqList in Store st, if present. Else, it stores the value -1.
2. An integer "size" which stores the total size of the SeqList.

The SeqLists are maintained as an array, such that i'th element of this array correspond to the SeqList for PRI_i. ie, assuming an array of three SeqLists, SeqList sl[0] will correspond to jobs with priority 0,SeqList sl[1] will correspond to jobs with priority 1, and similarly for job with priority 2.

The operations defined in SeqListOps.c are described below.

1. int initialize_elements (JobList list)
This function initializes List, which is an array of jobs. This function reads input n jobs from the user and stores it in List. Also, it returns n, number of jobs read from the user.

2. void void insertelements (joblist list,seqarr s)
This function takes JobList list and seqarr (i.e. array of structure seqlist) as input parameters. It creates three SeqLists seq[0], seq[1] and seq[2] for keeping track of Jobs with PRI_0, PRI_1 and PRI_2 respectively. It uses the function createList() described below to create these lists. The jobs in JobList list is inserted into seq[0], seq[1] or seq[2], based on their priority by calling insert (job ele , SeqList sl) that is described below. The contents of SeqLists seq[0], seq[1] and seq[2] are then copied back into JobList list using the function copy_sorted_ele(Seqarr s , jobList list).

3.SeqList createlist ()
This function creates and returns a new SeqList. It initializes the head of the new SeqList to nextFreeLoc of st and st[nextFreeLoc].next field is assigned to -1,indicating the list is empty. The size of the SeqList is set to zero.

4. SeqList insert(Job j , SeqList sl)
This function inserts the Job j in the SeqList sl. Insertion is done according to any one of the following two criteria:

a) Insertion on execution time of Job, in non-decreasing order.
b) Insertion on arrival time of Job, in non-decreasing order.

Insertion is done based on one of this two criterion based on the use of the compare() function (whose prototype is available in compare.h). ie. This compare function can have two implementations, each of which corresponds to one of the criterions. The two different implementations of compare() function are provided in files compare_at.c (comparison on arrival time) and compare_et.c (comparison on execution time). The function returns LESSER, GREATER or EQUAL (which are the values of enum ORDER in file storage.h).

5. void printSeqList (SeqList seq)
This function prints the attributes of all Jobs in seq. To accomplish this task, it repeatedly calls the print function, printjob (Job j), for each job in seq.

6. void printjoblist (joblist j)
This function prints all the Jobs in joblist. This function calls the printjob() function to print each job in j.

7. void copy_sorted_ele(SeqList s[3] , JobList ele)
This function copies all the Jobs in three SeqLists (of seqarr) back into ele. Copying is to be done priority wise, ie copy the Jobs with Priority 2 first into ele, then jobs with Priority 1 and then jobs with Priority 0. Now, joblist consists of Jobs sorted in order of highest Priority and lowest arrival time OR highest Priority and lowest execution time.

Implement the following functions. Assume that these functions are called only if the corresponding preconditions are met.

a) Write the code for functions insertelements() and insert () in file SeqListOps.c.
b) Write the two different implementations for compare function in files compare_at.c and compare_et.c
c) Write the code for function printSeqList() in file SeqListOps.c . The code for printJob() function is given in the file.
d) Write the code for function printJobList() in file SeqListOps.c . The code for printJob() function is given the in the file.
e) Write the code for function copy_sorted_ele() in file SeqListOps.c .
f) Compile the above files such that the resulting executable called exe_at will read the Jobs from user and store it in JobList, sort based on Highest priority and lowest arrival time using SeqLists and copy back the jobs to JobList.

g) Create an executable called exe_at, that will perform the above said operations based on Highest priority and lowest arrival time.
h) Create an executable called exe_et, that will perform the above said operations based on Highest priority and lowest execution time.

Assignment Task to be uploaded

Observe that the STORE (as defined above) stores the jobs of Priority_i (where i >=0 and i<= 3) at different indexes, of course in sorted order. Modify the above program to traverse the joblist 3 times. Each time, take the jobs with priority i and insert it in STORE in sorted order. Now, finally, the function copy_Sorted_ele () can copy the jobs back into joblist.
Upload your complete code (in zipped folder) on Nalanda. Name the folder with your BITS ID.