BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJ.)
CS F111 Computer Programming
LAB SESSION #8

(Compiling Multi-file Projects in C)

In today's lab, we are going to learn about creating a **project in C**. A C project consists of multiple **".c"** and **".h"** files, each of which implements a few modules of our project. We will study about the following topics in today's lab session:

- Header files in C
- Writing Modular Programs
- Creating a "C Project"
- Compiling multiple ".c" and ".h" to create a single executable
- Write shell script to combine multiple unix commands or compilation commands.
- A C project example with structures

Create a directory "**lab8**" inside "**myprogs**" directory for all your programs in this week's lab.

Exercise 1: Header files in C

A header file is a file with extension "**.h**". It contains *C function declarations* and *macro definitions* to be shared between several source files (or **".c"** files). There are two types of header files: *the header files that the programmer writes* and *the header files that come with your compiler*. You request to use a header file in your program by including it with the C pre-processing directive **#include**, like you have seen inclusion of **`stdio.h`** header file, which comes along with your compiler.

Including a header file into a **".c"** file enables us to use the variables declared in the header file. It also enables us to implement or use the functions declared in the header file. A simple practice in C programs is to keep all the ***constants***, ***macros***, ***global variables***, and ***function prototypes (declarations)*** in the header files and include that header file wherever it is required. We will see how and where to include header files in our **".c"** files. We will study about macros later in this course.

Include Syntax

Both the user and the system header files are included using the pre-processing directive **#include**. It has the following two forms –

- **`#include <file.h>`**

  This form is used for system header files that come with the compiler. It searches for a file named '**file.h**' in a standard list of system directories. Example of this type of inclusion is **`#include <stdio.h>.`** This file inclusion makes functions such as *printf()* and *scanf()* accessible to our program. Without this inclusion, we will not be able to use the above functions. It is an exercise to try it out!

- **`#include "myfile.h"`**

  This form is used for header files that we create for our programs. It searches for a file named '**myfile.h**' in the directory containing the current "**.c**" file (the file in which you have included **'myfile.h'**). All the function declarations, structure definitions and global variables that are present in **myfile.h** become accessible in our "**.c**" file.

We will study more about how file inclusion works when we study about C pre-processor, later in this course. Now let us go ahead by creating a simple program and making it modular using multiple **".h"** and **".c"** files. With this example, we will learn about using user made header files.

Writing Modular C Programs

Consider the following program that searches for an element in an array.

```c
// search_version1.c

#include <stdio.h>

int main()
{
    int arr1[10] = {3,24,32,74,28,9,5,1,6,11};

    int x = 32; // element to be searched

    int foundFlag = 0;

    int arrSize = sizeof(arr1) / sizeof(arr1[0]);

    for(int i=0; i<arrSize; i++)
    {
        if(arr1[i]==x)
        {
            printf("The element x is found in arr1 at position %d\n",i+1);
            foundFlag = 1;
            break;
        }
    }

    if(foundFlag ==0) printf("The element x is not found in arr1\n");

    return 0;

}
```

To compile this program, we use the following command:

```
jagat@gpuserver:~/cp$ gcc search_version1.c
```

This creates an executable with the name ./a.out. We can actually change the executable name to our choice by using the following command for compilation:

```
jagat@gpuserver:~/cp$ gcc search_version1.c -o exe1
```

This will create an executable with the name **exe1**. You can simply run the program with **./exe1** and it should print the following:

```
jagat@gpuserver:~/cp$ ./exe1
The element x is found in arr1 at position 3
```

The above program searches for element **x** in the array **arr1**. The entire program logic is written in the main function only. Now, what if in the same program, you wish to search for another element say **y** in another array say **arr2**? You will have to rewrite the entire program logic for the second array and second search element as shown below:

```c
// search_version2.c

#include <stdio.h>

int main()
{
    int arr1[10] = {3,24,32,74,28,9,5,1,6,11};

    int x = 32; // element to be searched

    int foundFlag = 0;

    int arrSize = sizeof(arr1) / sizeof(arr1[0]);

    for(int i=0; i<arrSize; i++)
    {
        if(arr1[i]==x)
        {
            printf("The element x is found in arr1 at position %d\n",i+1);
            foundFlag = 1;
            break;
        }
    }

    if(foundFlag ==0) printf("The element x is not found in arr1\n");

    foundFlag = 0;



    int arr2[10] = {32,2,3,7,2,99,57,17,65,10};

    int y = 17; // element to be searched

    arrSize = sizeof(arr2) / sizeof(arr2[0]);

    for(int i=0; i<arrSize; i++)
    {
        if(arr2[i]==y)
        {
            printf("The element y is found in arr2 at position %d\n",i+1);
            foundFlag = 1;
            break;
        }
    }

    if(foundFlag ==0) printf("The element y is not found in arr2\n");

    return 0;
}
```

To avoid such duplication we can make use of functions. The following is a program that is re-written using functions:

```c
// search_version3.c

#include <stdio.h>

// declare a function search()
// search() function should take an array inside it, its size and an element to be
searched.
// It should return the position of the element in the array.
int search(int arr[], int size, int ele);

// define the above search function
// this function searches for ele in arr and returns its position (index +1) or returns
-1 if not found
int search(int arr[], int size, int ele)
{
    for(int i=0;i < size; i++)
    {
        if(arr[i]==ele)
        {
            return i+1;
        }
    }
    return -1;
}

int main()
{
    int arr1[10] = {3,24,32,74,28,9,5,1,6,11};

    int x = 32; // element to be searched

    int arrSize = sizeof(arr1) / sizeof(arr1[0]);

    int eleIndex = search(arr1, arrSize, x);

    if(eleIndex >= 0) printf("The element x is found in arr1 at position %d\n", eleIndex
);
    else printf("The element x is NOT found in arr1");


    int arr2[10] = {32,2,3,7,2,99,57,17,65,10};

    int y = 17; // element to be searched

    arrSize = sizeof(arr2) / sizeof(arr2[0]);

    eleIndex = search(arr2, arrSize, y);

    if(eleIndex >= 0) printf("The element y is found in arr2 at position %d\n", eleIndex
);
    else printf("The element y is NOT found in arr2");

    return 0;
}
```

In the above program (**search_version3.c**) we have declared and defined a new function called "**search()**" that takes an *array*, its *size* and the *element* to be searched as input. It searches for the element and returns its position if it exists, else returns -1. ***This function has been re-used to find elements x and y in arr1 and arr2 respectively.*** In this way, the function "**search()**" can be re-used for any number of searches either in the same array or in a different array.

Now, suppose if you want to use this function **"search()"** in some other **".c"** file, which has been written for some other purpose. At this stage what we can do is to copy that function code from the current **".c"** file (**search_version3.c**) to the other **".c"** file where you wish to use this function. Doing this way, we might end up in copying 1000s of lines of code if wish to use many functions elsewhere. Can we do something better to avoid copying and duplication?

Here comes the role of user-made header files and multiple **".c"** files to create your C program/project. Now let us create a new C project. Create a new directory **myCProject1**, in which let us create three files: "**search.h**", "**linearsearch.c**", "**search_main.c**".

The file "**search.h**" would look something like this:

```
// "search.h"

// declare a funtion search()
// search() function should take an array inside it, its size and an element to be searc
hed.
// It should return the position of the element in the array.
int search(int arr[], int size, int ele);
```

Essentially **".h"** files are meant to contain *global variables*, *structure definitions* and *function declarations*. In this case, we are declaring the function **"search()"**. Now, we would want to implement / define the function in a **".c"** file. Let us do it in the file: "**linearsearch.c**" file, which would look something like this:

```
// "linearsearch.c"

#include "search.h"

// this function searches for ele in arr and returns its position (index +1) or returns
-1 if not found
int search(int arr[], int size, int ele)
{
    for(int i=0;i < size; i++)
    {
        if(arr[i]==ele)
        {
            return i+1;
        }
    }
    return -1;
}
```

Note that we have included **"search.h"** into this file. By this we can get access to the functions declared in **"search.h"** and we have implemented/defined the function **search()** with its exact signature (return type and input parameters) as that used in the **"search.h"** file. Now, the **“search.h”** and **“linearsearch.c”** files together can be used to support searching in an array into any C project that we want. We don’t have the main function defined in **“linearsearch.c”**. Any C program/project requires one (*and only one*) **main()** function to start executing. Let us create the file **“search_main.c”** that would contain the **main()** function. It can be created as follows:

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

int main()
{
    int arr1[10] = {3,24,32,74,28,9,5,1,6,11};

    int x = 32; // element to be searched

    int arrSize = sizeof(arr1) / sizeof(arr1[0]);

    int eleIndex = search(arr1, arrSize, x);

    if(eleIndex >= 0) printf("The element x is found in arr1 at position %d\n", eleIndex
);
    else printf("The element x is NOT found in arr1");

    int arr2[10] = {32,2,3,7,2,99,57,17,65,10};

    int y = 17; // element to be searched

    arrSize = sizeof(arr2) / sizeof(arr2[0]);

    eleIndex = search(arr2, arrSize, y);

    if(eleIndex >= 0) printf("The element y is found in arr2 at position %d\n", eleIndex
);
    else printf("The element y is NOT found in arr2");

    return 0;
}
```

Essentially, we have moved the declaration and definition of the function **search()** to their respective files and kept the rest as it is. However, we have included **“search.h”** in it, which gives access to the function **search()**.

Now in our **“search_main.c”** file, including **“search.h”** only gives access to the declaration of the **search()** function, but not to its actual implementation that we have implemented in **“linearsearch.c”**. For **“search_main.c”** to have access to the implementation of the **search()** function, we need to compile the files and link them together to generate single executable. Let us compile our **“.c”** files. For compilation of each of the **“.c”** files we must use **–c** option with **gcc**. This generates a corresponding **“.o”** or object file, but not the actual executable. Once we generate “**.o”** files for all the **“.c”** files, we can link them together to generate an executable. Here are the steps:

```
jagat@gpuserver:~/cp/myCProj1$ gcc -c linearsearch.c
jagat@gpuserver:~/cp/myCProj1$ gcc -c search_main.c
```

The above steps would generate **"linearsearch.o"** and **"search_main.o"**. You can use ls to check whether these files are generated or not. Now, we have to link them together to create a single executable as follows:

```
jagat@gpuserver:~/cp/myCProj1$ gcc -o search_exe linearsearch.o search_main.o
```

Now, we can run the program with the following command:

```
jagat@gpuserver:~/cp/myCProj1$ ./search_exe
The element x is found in arr1 at position 3
The element y is found in arr2 at position 8
```

Now, we can create another file with a **main()** function, say **"search_main2.c"** with a different set of input arrays and search elements, include **"search.h"** in it and then link the previously compiled **"linearsearch.o"** with **"search_main2.o"** to create another executable:

```
jagat@gpuserver:~/cp/myCProj1$ gcc -c search_main2.c
jagat@gpuserver:~/cp/myCProj1$ gcc -o exe2 linearsearch.o search_main2.o
```

In this way, we can see that **"linearsearch.c"** and **"search.h"** can be re-used in multiple projects without any need to copy or change the code. In fact there is no need to compile **"linearsearch.c"** again, if we have made no change to it. The previously compiled **"linearsearch.o"** can be directly used for linking while creating the second executable **exe2**.

There is another advantage of creating such modular files. In **"linearsearch.c"**, we implemented the **search()** function of the **"search.h"** file where we *searched the array from left to right to find the element*. Now, say we wish to implement the search function in such a way that it *searches from right to left* instead of left to right. And, whenever a user wants to use left to right version he/she should be able to use it. And, whenever the user wants to use the right to left version he/she should be able to use it. The switching should be done without any change to our actual ".c" files.

Let us create the file **"linearsearchR2L.c"**, in which we implement the search function that searches the array from right to left. This would also include **"search.h"** and would look like the following:

```c
// "linearsearchR2L.c"
#include "search.h"

// this function searches for ele in arr from right to left
int search(int arr[], int size, int ele)
{
    for(int i=size-1; i >= 0; i--)
    {
        if(arr[i]==ele)
        {
            return i+1;
        }
    }
    return -1;
}
```

You can notice the change in the logic of the program in for loop, where search now happens Right to Left. We can now compile this file by:

```
gcc -c linearsearchR2L.c
```

We now have the following **".o"** files in our directory: **search_main.o**, **lineasearch.o**, **linearsearchR2L.o**. Now if we want to use straight search (from left to right), we have to generate the executable with the following command and execute it:

```
gcc -o search_exe linearsearch.o search_main.o
./search_exe
```

And if we want to use search from right to left, we have to generate the executable with the following command and execute it:

```
gcc -o search_exe2 linearsearchR2L.o search_main.o
./search_exe2
```

Now if the user wants to search from left to right, he can use **search_exe** and when user wants to search from right to left, he can use **search_exe2**. We can notice that **search()** function declared in **"search.h"** has been implemented in two **".c"** files – **"linearsearch.c"** and **"linearsearchR2L.c"** Both the files have been used independently while generating executable files, i.e., whenever **"linearsearch.o"** has been linked to generate an executable, **"linearsearchR2L.o"** has not been used in linking. If both are together used in the same linking process to generate an executable, you will encounter an error. This is because both are defining/implementing the same function. You can go ahead and try doing that.

```
gcc -o exe search_main.o linearsearch.o linearsearchR2L.o
/usr/bin/ld: linearsearchR2L.o: in function `search':
linearsearchR2L.c:(.text+0x0):     multiple     definition     of     `search';
linearsearch.o:linearsearch.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Also try linking "**search_main.o"** without linking with **"linearsearch.o"** or **"linearsearchR2L.o"** file. You again notice an error.

```
gcc -o exe search_main.o
/usr/bin/ld: search_main.o: in function `main':
search_main.c:(.text+0x7f): undefined reference to `search'
/usr/bin/ld: search_main.c:(.text+0x119): undefined reference to `search'
collect2: error: ld returned 1 exit status
```

These errors are known as **Linker Errors.** Linker errors occur when the linker is trying to put all the pieces of a program together to create an executable, and it is not able to do because of some reason. The reason(s) can be anything but most cases include:

1. When you have declared and defined your own functions but failed to include all of the necessary object files in the linking process.
2. When there are more than one definition for a function, or a variable.
3. When the linker tries to create an executable and couldn't figure out where the main() function was located. This can happen if you accidently failed to include the file that actually has a main() function in it. A common mistake is also writing **Main()** instead of **main()** and not realising that C is case sensitive. In such cases the linker complains about an "**undefined reference to main**"

4. When you include the correct header files for all of your functions, but the linker is not provided with the correct path to the library that has the actual implementation. Now usually this is taken care of at the time of installation of your compiler and the path to the libraries is appended to the system variables that your linker can use. However, if in case it is not set up correctly it might be one of the issues.

To summarize, we were able to achieve the following by using multiple .c and .h files:

- Write our program in a modular way
- Able to reuse our code with different modules, simply by including the header files
- Able to create multiple definitions of the same function declaration and use whichever we want to by appropriately linking it and generating the executable.
- Same piece of code need not be compiled again and again if there is no change in it. For example, if we made change to **"search_main.c"** and no change to **"linearsearch.c"**, then we need to re-compile only **"search_main.c"** using **gcc –c**, and then re-link using **gcc –o** to generate a fresh executable. You don't need to re-compile **"linearsearch.c"** using **gcc –c**.
- Learnt about linker errors and various causes of it

We can have multiple **".c"** and **".h"** files as well. We will see such an example in the exercise coming next.

One important thing to be noted here is that only one **".c"** file in the entire project or (files that are linked to create an executable) must have a **main()** function defined. And program starts its execution from there.

<u>Shell scripts for compilation and linking:</u>

We can write multiple stages of compilation and linking into a single shell script file. This will enable us to do recompilation for every change we make in any of the files in the project, to a single line.

- Create a file **"myScript.sh"** in the same directory where our **".c"** and **".h"** files are present.
- Add the following lines into **"myScript.sh"**
  ```
  gcc -c linearsearch.c
  gcc -c search_main.c
  gcc -o search_exe linearsearch.o search_main.o
  ./search_exe
  ```
- Now simply run the following command:
  ```
  sh myScript.sh
  ```
  This will execute all of the above commands. Shell scripts can be used to bundle multiple commands together and execute them in one go. Any commands like **cp**, **mv**, **mkdir**, **echo**, etc. can be put inside a **".sh"** file. You can also run a shell script by: `bash myScript.sh`.

  You can also include the following lines to myScript.sh before the gcc statements:
  ```
  rm *.o
  rm *exe
  ```
  These two lines will remove the previously compiled **".o"** files and the previously created executables (files that end with the string **exe**). Then we can execute the remaining lines to freshly compile and create **".o"** files and the **exe**.

Additional Practice Exercises

Q1. Create a header file **“answer.h”** with the following function prototype.

```
int answer(int questionNo);
```

Now create 3 **“.c”** files as **“student1.c”, “student2.c” and “student3.c**". Each of these C files implement the answer function which returns an integer answer based on the question Number provided as given in the table below.

Question Number	Student 1 Answer	Student 2 Answer	Student 3 Answer	Correct Answer
1	15	5	45	5
2	10	10	10	10
3	2	0	0	0
4	4	4	4	4

Next, create a file **“checkAnswers.c”** that has a **main()** function implemented in it and includes the **“answer.h”** file. This file then scores the answers by calling the answer function and comparing the answers with the correct one. The script finally scores the student by telling how many answers were correct.

Now compile all the **“.c”** files and create their respective “.o”. Create executables separately for each student by linking their respective “.o” file with the “**checkAnswers.o**”. Then run each of the three executables one after the other to see the output. Also try to create a script file that automatically does all the compilation and scoring for all the students in on go.