I. C-TUTORIAL 1

The goal of the tutorial is to pick up C along the way. Therefore, this tutorial will not be exhaustive, but will teach you stuff you need to pick up for the sessions. There are excellent references for C, both online and books. O'Reilly C book is definitely a great reference. A useful online referece is at www.cprogramming.com - this site offers C and C++ tutorial and will be a quick reference guide.

We begin by looking at area.c

```
1  /*FILE: area.c
2
3  Programer; Sunethra Ramanan suna@physics.iitm.ac.in
4
5  Date: 22 Dec 2011
6
7  Version: Original
8
9  Revision-History:
10
11 Comments:
12 22 Dec 2011: This code calculates the area of a circle. The input radius is obtained
13 as a user input.
14
15 NOTES:
16 Compile using gcc -o area area.c This generates an executable called area. Run this
17 executable using ./area at the prompt.
18
19 Todo:
20 1. Hard code radius instead of an interactive input
21 2. Change the code such that r is now a grid starting from some r_min to r_max for
22 example 0. to 5. with a step size of 0.1. Generate the r grid. Using a for loop, loop
23 through all values of r and calculate the area. Print the output to screen.
24 3. Now declare a file pointer, open a file called area.dat and write the output to
25 file.
26 */
27
28 #include <stdio.h>
29 #include <math.h>
30 #include <stdlib.h>
31
32 /*****************************************************************************/
33 int
34 main(void)
35 {
36   const double pi = 4. * atan(1.);              /*defining pi*/
37   double r;                                     /*radius of the circle*/
38   double area;                                  /*stores the value of area*/
39
40   /*getting input*/
41   printf("Enter radius of the circle:");
42   scanf("%lf", &r);
43
44   /*calculate area*/
45   area = pi * r * r;
46
47   /*print output*/
48   printf("area of circle is:%f \n", area);
49
50   return(0);
51 }
```

1. Headers: Notice that there are several include statements before the code even begins, stdlib.h, math.h etc. These are called header files. They contain function prototypes. The files included here with the $\langle \cdots \rangle$ are

standard C headers and are kept in /usr/include directory. You can of course define your own header files. If this file is in your current working directory, for example my_header.h, then this is included within quotes:

```
#include "my_header.h"
```

Stuff declared with a # mark are called preprocessor directives. The processor sees this first.

2. The main code: This part of the code is the first that the compiler calls. It has a specific function prototype:

```
int main(void)
```

or

```
int main(int argc, char * argv[])
```

In the first case, nothing is passed to the main code. In the second case, one can pass command line inputs, which are also called standard inputs to the code. For instance, if the executable for the area.c code is called area and it requires an input file such as a list of radii for which the are should be calculated, then these could be stored in an input file and passed to the code as follows:

```
area area_input.inp
```

where the suffix ".inp" is a suggestion to the user that the file contains input data. argc counts the number of command line strings. In this example we have two strings: area and area_input.inp - the first input is always the name of the executable. If no input file is passed, then argc = 1. In this example, argc = 2.

3. Inputs: Almost all calculation will require some sort of input. This could be interactive, where the user types a choice or a value of a parameter at the terminal (command line input) or it could taken in through an input file, which is a non-interactive input. In this example, we use an interactive input. If you try compiling and running the code with the following commands:

```
gcc -o area area.c          /*compilation*/
./area                      /*running executable*/
```

then we get the following message on screen (try it!):

```
Enter radius of the circle:
```

In response we enter a number at the colon and hit enter. The code will take the input and store it in the variable called "r". The scanf statement does this. Note that the value of r is stored in its address (r is preceeded by &). This is just one way of doing an interactive input. In fact this way has a flaw. For example, instead of entering a number, if we enter an alphabet, which is stored as a character, or a blank space, the code will collapse. We need another way that takes care of such errors. Look that the following piece of code that should replace the scanf statement:

```
char line[100];/*Declaration of a character array of size 100*/
double radius; /*declaration of a variable called radius*/

printf("Enter radius of circle: \n");
fgets(line, sizeof(line), stdin);
sscanf(line, "%lf", &radius);
```

The declaration of a character array called line is done first. Of course we have already declared the radius in our code. Now we ask the user to enter the value of "r". The fgets statement reads the entire string and stores it in the character array called line. sizeof(line) tells the computer how big the array is, so that the appropriate amount of memory is allocated. We could of course replace sizeof(line) by the actual number which is 100, but this is very dangerous because if we change the size of array in the declaration in order to read a more descriptive line, we need to make sure we change the dimension in the fgets statement. Using the sizeof function takes care of this automatically. "stdin" means standard input, which is from the terminal. Therefore we tell the computer to read the line along with the value of r from the terminal and store it in "line". Next the sscanf statement reads this line and looks for a real (double) data type. If there is no such data, which would happen if the user enters nothing or a character, the return from sscanf would be '0'. If the user hits enter by mistake,

File mode	What they do
r	Read-only mode - file should exist
w	Write only - if file does not exist, it is created
	if file exists, it is re-written
a	Append - a file that exists is opened and data is appended
	if file does not exist, it is created
b	data in binary mode
r+	open for reading and writing (start at the beginning)
w+	open for reading and writing (file is overwritten)
a+	open for reading and writing

TABLE I: File opening modes

# index	Radius
1	0.5
2	1.0
3	1.5

TABLE II: Sample data file

the return from sscanf would be EOF (End of file character). We can catch this easily by the following lines of code:

```
if(sscanf == 0 | sscanf == EOF)
{
  printf("You must enter a real number for the radius! \n");
  exit(1);
}
```

As you can see, these lines make sure that the code runs only if the correct value of the radius which should be a real number is given by the user. Otherwise the code exits.

Another way to get inputs is to read from a file. Usually this means we need to declare file pointer. We do that as follows:

```
/*This declaration will be the first line in the main code*/
FILE *input_ptr; /*Declaring a file pointer*/
```

We then need to open the file and read from it. This can appear anywhere we need in the code once all the variable declarations are over. Here is how we open a file, let us call it radius.dat:

```
input_ptr = fopen("radius.dat", "r");
```

The general format for opening a file is

```
fopen(name of file, mode);
```

Here "r" stands for the mode in which the file should be opened, which is in the read-only mode. In order to open a file to write we need to replace "r" by "w". Here is a list of file opening options. Once the file is opened, we can read the data in it using the fgets and sscanf statements. Here is a sample data file which lists the radii for which the area should be calculated: The file in table II is read using the following lines of code which should come after the file is opened.

```
fgets(line, sizeof(line), input_ptr);

while(fgets(line, sizeof(line), input_ptr) != NULL)
{
  sscanf(line, "%d %lf", &index, &radius);
```

Symbols	Arithmetic Operations
+	Addition
-	Subraction
/	Division
*	Multiplication
%	Remainder of a division

TABLE III: Mathematical operators in C

```
  /*perform calculation here*/
  /*display results on screen or write into file*/
}
```

Here the first fgets statement reads the first line, which is a comment in the data file. The while statement continues to read the file line by line until the NULL character which is the end of file is reached. The symbol "! =" means not equal to. Therefore the code reads a value of the radius which is stored in the location of the variable radius, which could be used to calculate the area, display the result or write the result into another file and proceed to read the next line. This continues until the end of file is reached.

4. Calculations: In our example we calculate the area of a circle. The standard mathematical functions in C are defined as follows:

5. Output: The final step in the code is the output. Here the output which is the area of the circle is printed on the screen. *Can you modify the code such that the output is written into a file?*

6. Return value: C code treats every .c file as a function that does something and returns a value to some other function that calls it. The main function since it is the one calling other functions, should return 0. If the radius part was moved into a subroutine, then the return value would not be 0 but the value of the area. For example the following code tells you how such a subroutine could look:

```
 double area_calc(double r)
{
  const double pi = 4. * atan(1.);

  double a; /*variable to store the radius of a circle*/

  /*calculating area*/
  a = pi * r * r;

  return(a);
}
```

The function called area, which returns a double data type and takes in a double data type called r calculates the area which is stores in the variable called a and returns it at the end of the code to the calling code.

Here is a summary of the general structure of a C code:

```
 /*preamble within the comments: This should have the file name, the programer
details, date of writing the code, revision history and what the code does*/

header files included through the preprocessor directives.

Function Declarations: example:
double area_calc(double r); /*note the semi-colon at the end of the
declaration*/

main code
{
```

```
  FILE declarations
  integer declarations
  real declarations
  character declarations

  initializations of various variables

  opening files and reading data

  execution of something

  writing output

  return(0);
}

subroutines
```

Compiling and executing the code: Once a code is written, we need to compile the code, where the compiler translates the lines of code into an object file in machine language. This is done by the following piece of code typed at the terminal:

```
gcc area.c
```

This command compiles the code called area.c and generates an executale called a.out. In order to run the executable the following command can be used.

```
./a.out
```

While this will work, it is often useful to have separate names for the executable instead of the standard name a.out. We can do this by the following command:

```
gcc -o area area.c
```

This generates an executable called “area” and the executable can be run using:

```
./area
```

If there are libraries to be used they are added after name of the code. For example:

```
gcc -o area area.c -lm
```

where a math library which is not a part of the standard C menu is added. We will see examples, where you will add libraries to the code while compiling.

Sometimes we could have more than one code to compile and one code could call another. In such cases, it is very very useful to use makefiles which contain all the commands in a single file. A makefile in general captures more errors than the simple compilation seen here. We will see more examples and discuss this in detail soon.