Lecture IX : Introduction to C

I. AN ELEMENTARY EXAMPLE

The simplest program in C is perhaps the following :

```
hande@p439a:~$ cat simple.c
#include <stdio.h>
main() {
  printf(‘‘Hello!\n’’);
}
```

This is a C program that just prints the word Hello! on the screen when run. An equivalent program would be written in Octave in the following way.

```
hande@p439a:~$ cat simple.m
printf("Hello!\n");
```

Let's analyze and compare the two pieces of code shown above.

- C codes are written in files that have a .c extension rather than a .m extension.
- The C code has to start with a *library header declaration*. In the above program, the header that we use is stdio.h declared through the #include directive.

  ```
  #include <stdio.h>
  ```

  A *library* is a collection of specialized functions in C (also in many other programming languages). A *header* is a file that contains the *declarations* of the functions contained in those libraries. There are several header files in C representing numerous libraries. Each of these libraries is composed of a collection of functions serving a given purpose and providing the user with great flexibility. Some of the headers which are important for our purposes are :

Name of header	Character of functions	Examples
stdio.h	input and output including file manipulation	fopen, fclose, printf, fprintf, scanf, fscanf, fwrite
math.h	mathematical	acos, tan, cosh, sinh, atan2, exp, ceil, fabs
stdlib.h	general utility including memory allocation and random numbers	calloc, malloc, rand, exit, system, free
time.h	reading and converting the current time and date	clock, ctime, time, difftime
string.h	string manipulation	strcmp, memcpy, strcat, strlen

  In Octave on the other hand, although the functions are divided into libraries, there's no need to instruct the program to go and look for it in a specific location with a header. It knows where to look for the functions.

- The keyword main() in C needs to be included with all C programs. The parantheses indicate that main is a function just like everything else in C. Unlike Octave, C does not recognize standalone scripts. In the above example main does not take any arguments but in more general cases, it may take an arbitrary number of arguments. The curly parantheses in the beginning and the end designates the limits of the program.

  In Octave scripts work without a beginning keyword or delimiters. Functions on the other hand also require similar declarations.

- The semicolons at the ends of clauses are mandatory in C while they are optional in Octave.

II. MATHEMATICAL EXPRESSIONS VARIABLE DECLARATIONS, CONSTANTS

Usage of Octave and C for mathematical expressions is similar in some respects and different in others. Let's familiarize ourselves with these similarities and differences with the following example, concentrating on the highlighted areas :

```
#include <stdio.h>
#include <math.h>

main() {

  int  num1, num2, prod;
  double theta, sint;

  /* Find the product of two integers */

  printf("    ====== Product of two integers ======\n");
  printf("Enter two integers : ");
  scanf("%d %d", &num1, &num2);
  prod=num1*num2;
  printf("You entered %d and %d and their product is %d.\n",num1,num2,prod);

  /* Find the sine of an angle entered in degrees. */

  printf("\n   ====== Sine of an angle ======\n");
  printf("Enter an angle in degrees : ");
  scanf("%lf", &theta);
  printf("%lf\n", theta);
  theta*=M_PI/180;
  sint=sin(theta);
  printf("The sine of the number you have entered is %12.9lf.\n\n",sint);
}
```

- **Header :** In order to use mathematical expressions we need to include the `math.h` header.
- **Variable types :** In C, variables are handled in a very different manner than in Octave. While in Octave, types of variables do not have to be preassigned and may be changed arbitrarily during the course of the program, in C, each variable type must be defined at the beginning and is fixed throughout. See the first two columns of the following table for some of the variable types and examples of their declaration(there are a couple more but they are out of the scope of this course).

Variable type	Example declaration	Format
integer	`int var;`	%d
single-precision decimal number (32 bits)	`float var;`	%f
double-precision decimal number (64 bits)	`double var;`	%lf
arrays	`int var[15]; double var[30][20]`	—
characters	`char var`	%c
string	`char var[10];`	%s
file	`FILE *fid`	—

- **Comments :** Comments in C are written inside of `/* .... */`. `/` and `*` have to exist at both sides.
- **Reading input from screen :** In C, things you enter on your screen can be read in and assigned to a variable using `scanf`. In Octave, this is done with the more simplified function, `input`. The usage of `scanf` is rather odd in the sense that input values are read into *adresses* of variables rather than variables themselves. The `&`

operator returns the *address* of a variable in memory and it is a part of a broad and often confusing subject of *pointers*, which we will not get into here.

- **Mathematical functions :** `C` and `Octave` uses many mathematical functions in the same way, such as the product and trigonometric functions. Needless to say, `Octave` has orders of magnitude more mathematical functions specially optimized for specific data types such as arrays. In `C`, such utility functions must be written by the user. For instance, there are no internal functions in `C` that deal with arrays directly. Such operations must be handled through the use of loops.

- **Input and output :** While using `printf` and `scanf`, each variable must be called with a format identifier such as `%lf` or `%d`. These identifiers are given in the table above. These identifiers not only tell the system what kind of a variable will be printed but it also designates how it will be printed, i.e. its format. The format identifier `%12.9lf` in the above code, for instance, says that the variable should use 12 spaces when printed and there should be 9 digits after the decimal point. The last digit is rounded up or down depending on the following digit. This formatting ability allows the code developers to produce more readable, tidier output. `printf` in `Octave` is used mostly in the same way as in `C` except that the real numbers are only allowed the identifier `%f`.

- **Mathematical constants :** Headers do not only provide functions but they also contain certain constants that are *global* to the code, i.e. they can be seen by both the main function and all the subroutines. `M_PI` is the name of the variable that contain the value of π and is provided by the `math.h` file together with several other constants. For a complete list of such constants, see `http://www.gnu.org/software/libc/manual/html_node/Mathematical-Constants.html`

III. LOOPS AND ARRAYS

There are significant differences between the handling of loops and arrays between `C` and `Octave`. Let's see both in the following example :

```
#include <stdio.h>
#include <math.h>

main() {
  int i, N=20;
  int intarr[20];
  double darray[20];

  /* Fill in the arrays */

  for (i=0;i<N;i++) {
    intarr[i]=pow(i,2);
    darray[i]=intarr[i]*M_PI;
  }

  /* Print those elements of the double array which are smaller than 100 */

  i=0;
  while (darray[i]<100) {
    printf("Element no %d : %12.9lf\n",i+1,darray[i]);
    i++;
  }

  /* Report whether less or more than half the array is less than 100. */

  if (i<N/2)
    printf("\nLess than half the array is smaller than 100.\n\n");
  else
    printf("\nMore than half the array is smaller than 100.\n\n");
```

```
}
```

- **Array size :** In `Octave`, the array size does not have to be predeclared. In `C`, there are two ways to declare arrays :
  1. Static : If we know the size as in the above example. There is usually a limit to the number of elements you can use in a static declaration.
  2. Dynamic : If we don't know the size in advance and have to assign it inside the code. If we do it this way, a separate memory allocation using `malloc` or `calloc` need to be used.
- **Loops :** Even though the idea behind loops and the keywords used are the same, the syntax is rather different in `C` and in `Octave`.
  1. The `for` loop : While in `Octave`, the loop index for the `for` loop is an array of integers, doubles, characters or strings, in `C`, it can only be a single variable (integer or double) that needs to be incremented explicitly. The `for` loop in `C` does not need to be terminated with an `endfor`. Instead, if the number of lines within a for loop is more than one, the contents should be enclosed in curly braces. In fact, this applies to all keywords that initiate a block of tasks to be performed.
  2. The `while` loop : The syntax of the `while` loop is more similar to that in `Octave`, the only difference being again the braces and the lack of `endwhile`.
- **The `if` statement** : The condition on the `if` statement is written in the same way (in parantheses) in both `Octave` and in `C`. The `if...elseif...else` sequence in `Octave` is replaced by `if...else if...else` and as can be guessed there is no need for `endif`. Curly braces apply separately to each of the three keywords, as shown below.

```
if (...) {
   .
   .
   .
}
else if (...) {
   .
   .
   .
}
else {
   .
   .
   .
}
```

IV. FUNCTIONS

Just like in `Octave`, or any other programming language for that matter, it is usually advantageous to separate and group some of the tasks into functions. This is especially useful if we find ourselves using a particular sequence of tasks over and over again. Here's an example that demonstrates the usage of the `factorial` function inside `main`.

```
#include <stdio.h>
#include <math.h>

int factorial(int N) {

  int f=1,i;
  for (i=2;i<=N;i++)
    f*=i;
```

```
return f;

}

main() {

  int N;

  printf("Enter an integer : ");
  scanf("%d",&N);

  printf("%d! = %d\n", N, factorial(N));

}
```

- **Function declaration :** Functions can be declared within the same file as `main`. This may also be done in `Octave` although it's not the usual practice. Just like in `Octave`, both the input and the output variables must be declared in the function declaration. The difference in `C` is that the types as well as variable names must be included in the declaration. In fact, for the output variable, the variable name is not necessary, only the type is given.
- **Return value :** Inside the function, the variable that will contain the return value of the function is declared separately. Needless to say, the type of this variable and the return type of the function in the declaration section must be the same. After the function calculates the return value and assigns it to the return variable, this variable is forwarded to the outside world via the keyword `return`.
- **Variables in `main` :** As in `Octave` variables declared inside a function are local to that function and cannot be seen from `main`. We can therefore use the same variable (in this case `N`) both inside and outside the function.
- **Function calls :** Function calls from `main` are executed in the same way as in `Octave`.

Functions can be included in the `main` file as seen in the example above. However, as the size of the code and the number of functions become larger, it become impractical to cram everything in one huge file. For a more modular and readable code, functions are separated in different `.c` files. But then there arises the problem of telling `main` of the existence of these functions. This we can do creating home-grown header files. A user can write his/her own header files and include them in the main program just like predefined header files like `math.h`.

Now we'll write a variation on the factorial example above, separating at the same time the functions used in `main`. Study carefully the following three files :

```
hande@p439a:~$ cat functions.c
int factorial(int N) {

  int f=1,i;

  for (i=2;i<=N;i++)
    f*=i;

  return f;

}

double sum_array(double arr[], int size_array) {

  int n;
  double sum=0.0;

  for (n=0;n<size_array;n++)
```

```
    sum+=arr[n];

  return sum;

}
hande@p439a:~$ cat functions.h
int factorial(int);
double sum_array(double[],int);
hande@p439a:~$ cat subroutineex2.c
#include <stdio.h>
#include <math.h>
#include "functions.h"

main() {

  int i,N;
  double sum,arr[10];

  printf("    ====== Factorial ======\n");

  printf("Enter an integer : ");
  scanf("%d",&N);
  printf("%d! = %d\n", N, factorial(N));

  printf("\n   ====== Sum of array elements ======\n");
  printf("Enter an integer smaller than 10(size of the array) : ");
  scanf("%d",&N);
  printf("Enter the elements of the array : \n");
  for (i=0;i<N;i++) {
    printf("Element no %d : ",i+1);
    scanf("%lf",&arr[i]);
  }
  sum=sum_array(arr,N);
  printf("Sum of the elements of the array is %7.4lf\n\n",sum);

}
```

Starting from `main`, we see that it is aware of both the existence of the function `factorial` and that of `sum_array`. This is because in addition to the standard headers, we have now added our own header file `functions.h`. This is just a header that contains *function prototypes*, meaning function names and data types of input and output variables. The functions themselves are contained in a separate file called `functions.c`. When we run this code, we obtain the following

```
hande@p439a:~$ ./subroutineex2
   ====== Factorial ======
Enter an integer : 4
4! = 24

   ====== Sum of array elements ======
Enter an integer smaller than 10(size of the array) : 3
Enter the elements of the array :
Element no 1 : 1.2
Element no 2 : -0.3
Element no 3 : 2.1
Sum of the elements of the array is  3.0000
```

V. COMPILATION AND LINKING

In all the above codes, the code that we eventually run from the command line (*the executable*) needs to be produced from the *source code*, i.e. the code that is written in `C`. This is particularly important in the above case where the executable needs to know about two other files. Unfortunately, the procedure of going from code to executable is not as transparent in `C` as it is in `Octave`. In fact, in `Octave` commands are executed as they are typed so we don't make much of a distinction between code and executable. In contrast in `C`, the text that makes up the source code in the `.c` and `.h` files should be converted into machine language through what is called a *compiler*. A compiler is basically a program in Linux that does this conversion. There are several versions of `C` compilers in Linux: `cc, gcc, g++` are some of these. The number of options available to compilers is particularly large and to make the best use of them requires expertise. We are not going to go into details of this subject here but instead just see a brief overview. You can see and be amazed at the option repertoire of `gcc` by looking at its `man` pages.

If you have a single `.c` file, say our `simple.c` example above, the most basic way is to run

```
hande@p439a:~$ gcc simple.c
hande@p439a:~$ ./a.out
Hello!
```

As a result of running `gcc`, a machine code is produced, which has the rather eccentric name `a.out`. If you try to see this file on your terminal using `less` or `cat`, you will see that it's a lot of junky binary text, which is interpretable by your computer. This final product that you can no run to see the result of your code is called an *executable*.

To give your executable a more intelligible name, you can employ the `-o` option of your compiler. All compilers mentioned above should have this option available.

```
hande@p439a:~$ gcc -o simple-code simple.c
hande@p439a:~$ ./simple-code
Hello!
```

Unlike `Octave` the filename and the name of the executable may be different. You may choose the name of your executable to be anything you wish as long as it complies with naming conventions. The name you choose for your executable should immediately follow the `-o` option.

Some of the libraries need to be treated specially during compilation. They need to be *linked* using an option that start with `-l`. The full name of the option depends on the specific library that you are using, e.g. `-llapack, -lblas, -lf2c`. If you are using `gcc`, the math library needs to be linked with `-lm` while for `g++` this is not necessary.

```
hande@p439a:~$ gcc -o mathex mathex.c -lm
hande@p439a:~$ g++ -o mathex mathex.c
```

If you have distributed your functions over separate files as e have done in `subroutineex2.c`, the compilation is carried out in multiple steps.

```
hande@p439a:~$ gcc -c functions.c subroutineex2.c
hande@p439a:~$ gcc -o subroutineex2 subroutineex2.o functions.o -lm
```

The first step using the `-c` option creates an intermediate set of files called object files. These files have the `.o` extension. These are again written in machine language. In the second step, these object files are combined with our main file to produce the final executable.

If you are interested in how a compiler works, see a nice account on `http://users.actcom.co.il`).