

# 1 Introduction to C – part I

In this class, we will be employing the modern “C” language which is more and more commonly becoming the standard for scientific programming. We will not use any of the Object Oriented properties of C++, as these are seldom required in scientific programming, although they can be useful in some applications. Some of you will already have a working knowledge of C. Here follows a basic introduction to get you started or as a quick refresher.

## 1.1 Computer Memory and Variables

Basically speaking, a computer program is a series of commands that manipulate data stored in the computer’s memory and yields a result. One can think of the memory of a computer as a series of boxes, each of which can contain one piece of data (a number, a character, etc.). Each of these boxes, or locations in memory, has its own **address**. A **variable** is a label which points to the address of a particular location in memory. It is very seldom necessary in scientific programming to know the *actual* location or address of a particular piece of data. However, in C programming, the concept of address is very important, although we will always reference a piece of data using a variable name.

There are a number of different types of variable in C. For instance, we may have to deal with integers (whole numbers, such as 1, 5, -32, etc.) in our program. These need to be distinguished from numbers that have decimal points (0.56, 2.934, 4.345e-20) called floating-point numbers. We may also have variables that refer to characters and to strings.

All variables in C must be *declared* or defined, and are generally of types mentioned above, although we will have opportunity to use integer and floating point variables of different **precision**. For instance, we have two types of integers, single precision integers, referred to simply as **integer** and double precision integers, referred to as **long**. In addition, we have single and double precision floating point, **float** and **double**. Variables are declared at the beginning of the program as follows:

```
double x;      /* double precision floating point */
int i,j;      /* integers */
float y;      /* single precision floating point */
long n;      /* double precision integer */
```

```
char s;          /* character */
```

(Note – text inbetween `/*` and `*/` are comments and not part of the program. It is always a good idea to place plenty of comments in your program).

What is the difference between, for instance, a variable declared as a **float** and one that is declared as a **double**? Both are what we call floating point variables, but a double precision floating point variable has twice the amount of memory allocated to it than a single precision floating point variable. Since the computer uses twice as much memory to represent a double than a float, this means that the double is represented in memory with more precision (effectively, this means the number can be represented with more decimal places). One would thus use doubles instead of floats if one was concerned with the precision of the mathematics (for instance, if one needs to subtract two large numbers which are almost equal). Years ago, when computer memory was in short supply and was expensive, it was advisable to declare floating point variables as **float**, and reserve the **double** declaration only for variables which required the extra precision. At present, computers usually have so much memory that it is routine to declare one's floating point variables as **double** unless one is working with exceptionally large amounts of data.

It can be helpful to use certain **unsigned** types as well. For instance, on some machines and compilers, the maximum range for an integer is from -32768 to 32768. An **unsigned int** variable will have the range 0 to 65536.

Arrays of variables may be declared similarly. One-dimensional arrays can be defined as follows:

```
float r[5];
```

and then the components of this array (usually called a *vector* in C), can be accessed as `r[0]`, `r[1]`, `r[2]`, `r[3]` and `r[4]`. Notice that vectors in C generally begin with index 0, going up to  $n - 1$  where in the above example,  $n = 5$ .

Another way to declare a vector and allocate memory for it is as follows. First, declare a *pointer* to a vector:

```
float *r;
```

then, use the function `vector` to allocate memory for the vector:

```
r = vector(0,4);
```

These two steps will have exactly the same effect as the statement `float r[5];`. We will discuss later why using the second method may be preferable to the first. Note that if we want to use the function `vector`, this is not a built-in C-function, such as `sin` or `cos` (see § 1.3). However, this function is included in a file called `comphys.c` that your instructors have provided you. This means that whenever you want to use a function out of `comphys.c`, you will have to remember to put the following statement at the top of your program —

```
#include ‘‘comphys.h’’
```

and then when you compile your program, to include `comphys.c` in the `gcc` statement, for instance:

```
gcc -o myprogram myprogram.c comphys.c
```

but more on that later!

Two-dimensional arrays may also be defined in a similar way, thus

```
float A[5][4];
```

will yield a *matrix* with the following components:

```
A[0][0], A[0][1], A[0][2], A[0][3],  
A[1][0], A[1][1], A[1][2], A[1][3],  
A[2][0], A[2][1], A[2][2], A[2][3],  
A[3][0], A[3][1], A[3][2], A[3][3],  
A[4][0], A[4][1], A[4][2], A[4][3]
```

and likewise there is a function in `comphys.c` called `matrix` which you can use to do the same thing, if you want.

A single character variable may be declared as follows:

```
char c;
```

although a string, i.e. a variable containing a word or multiple words of text will be declared as follows:

```
char str[80];
```

Here the variable `str` will be able to store text up to 80 characters long.

Occasionally one may have a value that is used commonly in a program that never gets changed, such as the value for  $\pi$ , or the value of the acceleration of gravity. In such cases, it is useful to employ a **macro** as follows:

```
#define PI 3.141592
#define G 9.8
```

Such macro definitions occur at the very beginning of the program. Another way of doing essentially the same thing is by using the `const` declaration:

```
const double pi = 3.141592653589793;
const int MaxOrbitals = 10;
```

## 1.2 Input and Output

Reading input from the screen is simple in C. Let us suppose we have declared two variables, `x` and `k` with the following statements:

```
double x;
int k;
```

and now want to input some values from the keyboard for these variables. We might use the following lines of code:

```
printf(“\nEnter a value for x > ”);
scanf(“%lf”,&x);
printf(“\nEnter a value for k > ”);
scanf(“%d”,&k);
```

The function `printf` prints a line to the screen – notice that we began the string “Enter a value for x >” with the line feed/carriage return control character `\n`. The program will wait for you to enter a value on the screen. Once you hit <Enter>, the function `scanf` will read the value you entered. Note that the arguments for `scanf` include a formatting statement (`%lf`) and the *address* of the variable you are reading in (`&x`). The function `scanf` always requires the address of the variable (designated using `&` in front of the variable name), and not the variable name itself. The format statement `%lf` simply states that the variable is a long floating point (double) variable. In the second `scanf` example, the variable is an integer, and so the formatting statement `%d` is used. Below, find some typical format statements for different types of variables:

```
%f    float
%lf   double
```

```

%d      integer
%ld     long  (double precision integer)
%e      floating point entered in scientific notation - 1.523e-13
%le     double entered in scientific notation
%g      free format float
%lg     free format double
%s      string

```

The `printf` statement may be used, with formatting, to output values of variables to the screen. Let us suppose we wish to output the value of `x`, a variable that has been declared as a `double` and `k`, an integer. We may use the following statements:

```
printf('The value of x is %f and k is %d\n',x,k);
```

Notice in the `printf` statement, one need not differentiate in the formatting between a `double` and a `float`. Also, `printf` takes the variable name, not the address, unlike `scanf`.

Now, if the value of `x` is 5.0 and the value of `k` is 4, the above statement will print as:

```
The value of x is 5.000000 and k is 4
```

It might look nicer if the statement were printed as

```
The value of x is 5.0 and k is 4
```

This can be accomplished with the following `printf` statement:

```
printf('The value of x is %3.1f and k is %d\n',x,k);
```

Notice in the formatting statement `%3.1f` the “3” stands for the total number of characters in the output (“5”, “.”, “0”), including the decimal point, and the “1” stands for the number of digits after the decimal point.

**Example:** In the first lab, you learned how to compile a simple program, called “Hello World”. The C-code for this program is as follows:

```

#include <stdio.h>

int main()
{
    printf('\nHello World!\n');
    return(0);
}

```

The significance of the different lines (for instance, the one with `#include <stdio.h>`) will be discussed in the next section. Let's now use our new-found knowledge of input/output to modify this simple "Hello World" program in an interesting way. Type the following into emacs, and save it under `hello2.c`:

```
#include <stdio.h>

int main()
{
    char name[30];

    printf('\nEnter your first name > ');
    scanf('%s',name);

    printf('\nHello %s!\n',name);

    return(0);
}
```

compile it with the following command line:

```
gcc -o hello2 hello2.c
```

and then run it with the following command:

```
./hello2
```

**Exercise 1.1:** Carry out the previous example with the modified Hello World.

### 1.3 Mathematics

The basic arithmetical operations are defined in the manner that you would expect, using, for instance, the operators `+`, `-`, `*`, `/`, `%`. The last operator, "`%`" is the modulus operator which gives the remainder from integer division. For instance,

```
z = 13 % 7          /* The result is 6 */
```

Table 1: Commonly Used Math Functions in C

<code>pow(x,y)</code>	Raising to a power $x^y$
<code>fabs(x)</code>	Absolute value of $x$
<code>sqrt(x)</code>	Square root
<code>sin(x)</code> , <code>cos(x)</code>	Sine and Cosine of $x$
<code>tan(x)</code>	Tangent of $x$
<code>asin(x)</code> , etc.	Inverse trig functions
<code>exp(x)</code>	$e^x$
<code>log(x)</code> , <code>log10(x)</code>	Natural log, $\ln x$ and Common log, $\log_{10} x$
<code>floor(x)</code>	Round down to nearest integer
<code>ceil(x)</code>	Round up to the nearest integer

Exponentiation, however, is different in C than in many other computer languages. To raise a variable `x` to a power `y` and place the result in another variable `z`, the function `pow` is used, thus:

```
z = pow(x,y);
```

in addition, if one wants to add, subtract, multiply or divide and put the result back into the same variable, one can use some special C shortcuts:

```
x += 2.5;    /* same as x = x + 2.5; */
y -= 3;     /* same as y = y - 3;   */
z *= x;     /* same as z = z * x;    */
y /= z;     /* same as y = y/z;      */
```

A number of predefined mathematical functions can be used in C; some of the more common are listed in Table 1.

It is important to remember that when carrying out arithmetic with integers the computer *truncates* (i.e. rounds down to the nearest integer) the result. Thus, in floating point,  $3.0/2.0 = 1.5$  but in integer arithmetic  $3/2 = 1$ . If you mix types in an arithmetical statement, it is a good idea to *cast* the variables to a consistent type. For instance, let us suppose we have the following declarations:

```
double x=30.5,y;
int a=3;
```

and we want to calculate  $y = x/a$ . It is best to write this statement as

```
y = x/(double)a;
```

in this example, we have *cast* the type of **a** to be a double, that is to say, in this expression, **a** is represented as 3.0, not 3.

## 1.4 A Simple Program

The following program may be typed into your programming environment:

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

int main()
{
    double wave,T;
    double p = 1.19106e+27;
    double p1;

    printf('\nEnter the wavelength in Angstroms > ');
    scanf('%lf',&wave);
    printf('\nEnter the temperature in Kelvin > ');
    scanf('%lf',&T);

    p1 = pow(wave,5.0)*(exp(1.43879e+08/(wave*T)) - 1.0);

    printf('The Planck function at %7.2f A and %7.2f K is %e\n',
           wave,T,p/p1);
    getchar();
    return(0);
}
```

Save this program as `planck.c`. Before we learn how to compile and run this program, let us look at each line in detail.

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

These two lines include *header* files in the program. In the C language, each function (such as `exp`, `pow`) must be declared beforehand (we will discuss later

what these declarations involve); for built-in functions, these declarations are contained in header files. In the case of math functions, the declarations are found in the header file `math.h`. In the case of input and output functions, such as `printf` and `scanf`, the declarations are found in `stdio.h`. Other common header files that you may have to use are `stdlib.h` and `string.h`. We will discuss which function declarations are found in those header files in class.

```
int main()
{
```

Every C program is composed of a number of *functions* which perform certain tasks. These functions (also known as routines or subroutines) always have a name and a *return type*. All C programs must have a `main` function. The `main` function by standard convention has a return type of `int`, which means that it returns an integer to the operating system. By convention, if a “0” is returned, then the program is assumed to have terminated normally without error. If the program returns a “1”, this indicates to the operating system that an error has occurred. Some compilers accept a `void` type for `main`. Some other functions (see lecture 3) will return a character, others a double, etc. Some functions have a `void` return type which means that they don’t return anything! We will discuss this in more detail later. The bracket immediately below the main function declaration indicates the beginning of the function. The function will be ended with a close bracket.

```
    double wave,T;
    double p = 1.19106e+27;
    double p1;
```

These statements declare a number of variables used in the program. Note that the `p` variable is *initialized* with a certain value. Note also that in a program, all statements end in a semicolon “;”. The exceptions to this rule are the `#include` statements and macro statements.

```
    printf(“\nEnter the wavelength in Angstroms > ”);
    scanf(“%lf”,&wave);
    printf(“\nEnter the temperature in Kelvin > ”);
    scanf(“%lf”,&T);
```

These are input statements similar to those we discussed before.

```
p1 = pow(wave,5.0)*(exp(1.43879e+08/(wave*T)) - 1.0);
```

This is the mathematical statement that actually calculates the Planck function.

```
printf('The Planck function at %7.2f A and %7.2f K is %e\n',  
      wave,T,p/p1);
```

This statement prints to the screen the value of the Planck function for the input values of the wavelength and the temperature.

```
getchar();
```

This statement requires you to press the Enter key to exit the program. You may find that it is necessary to include two instances of this line for the program to work correctly.

```
return(0);
```

This is the *return* statement which returns the *return value* from the main function. Since `main` is an *int* function, an integer (0) is returned, which indicates normal termination of the program.

```
}
```

Finally, all C functions end with a close-bracket.

This program may be compiled with the following command:

```
gcc -o planck planck.c -lm
```

The flag `-lm` must be included in the `gcc` command whenever a program includes a math function, such as, in this case, `exp` and `pow`. This flag “links” in the math library. Run the program using the command `./planck` and enter some reasonable values such as `wave = 5000` and `T = 6000`.

**Exercise 1.2:** Write a short program that will prompt the user for an initial velocity  $v_0$  and final time, and then calculate and output to the screen the velocity at the final time for an object subject to a gravitational acceleration of  $g = -9.8\text{m/s}^2$ . The following is the relevant equation, familiar to you from introductory physics:

$$v = v_0 + gt$$

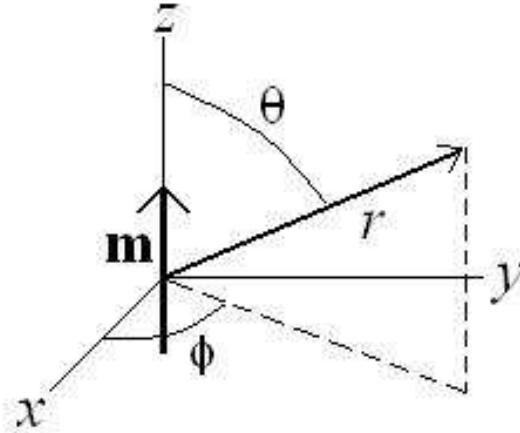


Figure 1: Magnetic Dipole

**Exercise 1.3:** Magnetic Dipole

The magnetic field of the "pure" dipole in SI units is given as:

$$B(r) = \left( \frac{\mu_0 m}{\pi r^3} \right) (2 \cos \theta \hat{r} + \sin \theta \hat{\theta})$$

where  $\mu_0 = 4\pi \times 10^{-7} [NA^{-2}]$ . Assume that the magnitude of the magnetic dipole moment ( $m$ ) is 1 [ $Am^2$ ]. Query the user to provide a distance,  $r$  [in meters], and an angle,  $\theta$ , [in degrees] and output the magnitude of the  $\hat{r}$  and  $\hat{\theta}$  components [in meters] to the screen. Do you get what you'd expect for  $\theta=0, 90, 180$ , and  $270$  degrees?

Here is some code to get you started. Notice that there is code that handles the divide-by-zero error by asking the user to re-enter a value for  $r$ . This uses "loops" and "conditionals" which we will talk about very soon...just complete the code with the math statements required. HINT: math.c uses radians in its sin and cos functions. The user is entering degrees – you need to do a conversion before computing the results.

```

/* Bdip exercise
YOUR NAME HERE */
#include <stdio.h>
#include <math.h>

```

```

const double pi=3.141592653589793;

int main()
{
    double r=0.0;

    /* MAKE THE REST OF YOUR DECLARATIONS HERE */

    printf("\nThis program will provide the r(hat) and theta(hat)
           components of the B field for a pure dipole.\n");

    /* PROMPT THE USER FOR  r AND theta HERE */

    /* THE FOLLOWING CODE HANDLES THE DIVIDE BY ZERO POSSIBILITY */

    while (r == 0.0) {
        printf("\nEnter the distance, r > ");
        scanf("%lf",&r);
        if (r == 0.0) {
            printf("\nr can not be zero, try again...\n\n");
        }
    }

    /* PUT THE REST OF YOUR CODE HERE */

    return(0);
}

```