

3 Introduction to C – part III

3.1 Functions

We have already noted in class that a C-program is made up of a number of functions. Every C-program has at least a `main` function, and, as we have seen, C-programs can make use of built-in functions such as math functions and input/output functions. It is, however, often convenient to write our own functions to handle certain calculations or perform certain operations. As an example of how to introduce a user-written function into our program, let us modify our Planck function example from part II.

```
#include <stdio.h>
#include <math.h>
double planck(double wave, double T);

int main()
{
    double wave,T;
    double result;

    printf("\nEnter the temperature in Kelvin > ");
    scanf("%lf",&T);

    wave = 500.0;

    while(wave <= 12000.0) {
        result = planck(wave,T);
        printf("The Planck function at %7.2f A and %7.2f K is %e\n",
            wave,T,result);
        wave += 500.0;
    }
    getchar();
    return(0);
}

double planck(double wave, double T)
```

```

{
  static double p = 1.19106e+27;
  double p1;

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

```

Notice in this example we have placed all the mathematical calculations in a “function” called `planck`. Note as well that we have added a declaration statement to the beginning of the program:

```
double planck(double wave, double T);
```

This statement performs two functions. First, it declares `planck` to be of type `double` which means that it returns a double-precision floating point number to the `main` function. In addition, this statement informs the compiler that `planck` requires two parameters to be *passed* to it, both double-precision floating-point numbers (`wave` and `T`).

At the end of the C-program we have the actual code for the function `planck`, starting with what is essentially a repeat of the declaration statement. The code is enclosed in brackets. Note that the code for `planck` uses the parameters passed to it by the `main` program, does a calculation, and then returns a number `p/p1` to the main program. In the main program, this number is placed in the variable `result` in the following statement:

```
result = planck(wave,T);
```

and thus you can see that `planck` can now be used in the same way as other math functions such as `sin` or `cos`.

Parameters can be passed to functions in two different ways. They can be *passed by value* or *passed by reference*. In the example above, they were passed by value. What this means is that the function receives its own copy of the parameters, which it can change and modify to its heart’s content without changing the value of the parameters in the `main` function. If we want the function to make permanent changes to these parameters, then we must pass the *addresses* of these parameters to the function. Then whatever changes the function makes to these parameters take effect in the main function as well. This is a way of enabling a function to “return” more than one value. If we pass `wave` and `T` to `planck` by value, then we would use the statement

```
result = planck(wave,T);
```

On the other hand, if we wish to pass `wave` and `T` to `planck` so that `planck` can permanently modify them, then we must use the statement

```
result = planck(&wave,&T);
```

In this case, `wave` and `T` must be referred to as `*wave` and `*T` in `planck`, and the declaration for `planck` would look like:

```
double planck(double *wave, double *T)
```

Another way to pass information to functions is through the medium of *external variables*. There are two types of variables in C, local and external variables. In the example above, `result` is a variable *local* to `main`; it cannot be “seen” or referenced by `planck`, for instance. The variables `p` and `p1` are local to `planck` and cannot be seen by `main`. If, however, we place the declarations for `wave` and `T` at the top of the program, before `main`, then they are *external* variables which can be seen (and modified) by both functions. The next example shows a use of external variables:

```
#include <stdio.h>
#include <math.h>
double wave,T;      /* External Variables */
double planck();

int main()
{
    double result;

    printf("\nEnter the temperature in Kelvin > ");
    scanf("%lf",&T);

    wave = 500.0;

    while(wave <= 12000.0) {
        result = planck();
        printf("The Planck function at %7.2f A and %7.2f K is %e\n",
            wave,T,result);
    }
}
```

```

    wave += 500.0;
}
getchar();
return(0);
}

double planck()
{
    static double p = 1.19106e+27;
    double p1;

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

```

Note the changes in the declaration of `planck` and how `planck` is referenced in the main program.

Exercise 3.1: Write a program that will pass a variable `double T = 100.0` by reference to a function (declare it `void doit(double *T)`) which will then add 100 to `T` and pass it back to `main`. Print the variable `T` out in `main` to verify that it has indeed been incremented by 100.

Exercise 3.2: Do the same thing as **Exercise 3.1**, but now declare `T` as an external variable.

Exercise 3.3: In our traditional mathematical world, we permit numbers with an infinite number of nonperiodic digits. Real-world arithmetic *defines* $\sqrt{3}$ as that unique positive number that, when multiplied by itself, yields the integer 3. However, in a computer, we have only a finite number of bits available to represent all numbers. So although the computer can not accurately represent $\sqrt{3}$, it approximates it so that when it is squared, the result is so close to the integer 3 that it is acceptable. The computer *error* associated with approximation arises from *truncation* or *rounding*. For example, the actual decimal value of $\sqrt{3}$ can be written as 0.33333333... where there are an infinite number of 3's in the mantissa; however, the computer approximation can only place 24 3's in the mantissa (for floating point numbers on a 32-bit machine) - the computer truncates the rest of the 3's.

Suppose that E_n represents that magnitude of the *absolute error* of an iterative algorithm after n iterations. Absolute error is just defined as the absolute value of the difference between an actual value, p , and an approximated value, p^* : $E = |p - p^*|$. After n iterations, if $E_n = CnE_0$, where E_0 is the initial error, and C is a constant independent of n , then the error is said to be *linear*. If $E_n = C^n E_0$, then the error is said to be *exponential*. Linear error growth is usually unavoidable and when C and E_0 are small, the results are generally acceptable. Exponential growth should be avoided. A system exhibiting linear error growth is known to be *stable*; a system exhibiting exponential growth is known to be *unstable*.

The Taylor polynomial for $f(x) = \sin x$ about $x_0 = 1.0$ is

$$p(x) = \sum_{i=1}^N (-1)^{i+1} \left(\frac{x^{2(i-1)+1}}{(2(i-1)+1)!} \right).$$

Our goal is to find out how many of the terms in the Taylor series we need to keep in order to stay within a user-defined error. Below (and provided) is a program that finds the minimum number of iterations, N , that satisfies the absolute error equation:

$$\left| \sin\left(\frac{\pi}{4.0}\right) - p\left(\frac{\pi}{4.0}\right) \right| < 10^{-5}.$$

Run this program using the command:

```
./program_name tolerance > ex3.out
```

where `program_name` is the name that you assign the executable program at compile time,

```
gcc -o program_name ex3.c -lm
```

and `tolerance` is either a decimal or scientific notation value. Plot the results using `gnuplot`:

```
gnuplot> plot "ex3.out" using 1:2 with lines
```

Modify the program to do the same thing for $\exp x$, where the Taylor series expansion is:

$$p(x) = \sum_{i=1}^N \frac{x^{i-1}}{(i-1)!}.$$

Also, make the passed variables *external* instead of being passed by value. How do the rates of error-reduction compare between the two functions? You may find that this program may be of value to you in the future - add to to your code library.

```

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

double getp(int i, double x);
double geterror(double p,double pstar);
double factorial(int i);

const double pi=3.141592653589793;

int main(int argc, char *argv[])
{
    int i,N;
    int M=50; //set the maximum number of allowable steps
    double tolerance=0.0;
    double error=0.0;
    double p=0.0;
    double pstar;
    double x;

    x=pi/4.0;

    while (tolerance <= 0.0) {
        if (argc != 2) {
            printf("\nEnter the tolerance (scientific notation, e.g. 1.5e-13) > ");
            scanf("%le",&tolerance);
        } else tolerance=atof(argv[1]);

        if (tolerance == 0.0) {
            printf("\nTolerance can not be zero or negative, try again...\n\n");
            return(0);
        }
    }

    pstar=sin(x);

    for (i=1;i<=M;i++) {
        p=p+getp(i,x);
        error=geterror(p,pstar);
        printf("%d %le\n",i,error);
    }
}

```

```

    if (error<tolerance) {
        break;
    }
    if (i==M) {
        printf("\nThe number of iterations exceeds allowable maximum, retry\n");
        printf("\nwith new tolerance or change max limit.\n");
        break;
    }
}
return(0);
}

double getp(int i, double x)
{
    int term0;
    double term1,term2;

    term0=2*(i-1)+1;
    term1=pow( (-1.0),(i+1) );
    term2=( pow(x,term0) / factorial(term0) );
    return( term1*term2 );
}

double geterror(double p, double pstar)
{
    return ( fabs(pstar-p) );
}

double factorial(int i)
{
    int j;
    double result=1.0;

    if (i==0) {
        return(1.0);
    }
    else {
        for (j=1;j<=i;j++) {
            result=result*(double)j;
        }
    }
}

```

```
    }  
  }  
  return(result);  
}
```

3.2 File Input and Output

So far we have printed the output from our programs to the screen. This is useful only if there is a relatively small amount of output, but if we have pages and pages, it is more desirable to output to a file. This file can then be used by other programs, or we can import it into a graphics program such as `gnuplot` and plot our results. How do we do file input and output?

We must first define a “file pointer” and then use the `fopen` function. Consider this fragment of code:

```
FILE *in,*out;  
  
in = fopen("input.dat","r");  
out = fopen("output.dat","w");
```

here we have defined two file pointers `*in` and `*out` and then used the `fopen` function to open the file `input.dat` for reading (`r`), and the file `output.dat` for writing (`w`). The file pointers “point” to “streams” `in` and `out` which can be used to input and output data, respectively, to the files `input.dat` and `output.dat`. We can use the functions `fscanf` and `fprintf` to read and write data. Consider the following code fragment:

```
double x,y;  
int i;  
FILE *in,*out;  
  
if((in = fopen("input.dat","r")) == NULL) {  
    printf("\nCannot open file for input\n");  
    exit(1);  
}
```

```

if((out = fopen("output.dat","w")) == NULL) {
    printf("\nCannot open file for output\n");
    exit(1);
}

for(i=0;i<10;i++) {
    fscanf(in,"%lf",&x);
    y = x*1.34e+22;
    fprintf(out,"%e",y);
}
fclose(in);
fclose(out);

```

Notice that we have embedded the two `fopen` statements in `if` statements and have checked to see if the function `fopen` returns in either case a `NULL`. If `fopen` does return a `NULL`, then this means that something has prevented the program from opening the file. In the case of a file opened for reading, this might happen if the file does not exist, or if the file “permissions” do not allow reading. If `fopen` returns a `NULL`, then the program exits gracefully using the `exit(1)` statement. It is important to include such checks in your program. If you don’t, and `fopen` returns a `NULL` then nonsense will be input into your program, and you will get garbage out! Notice that at the end we closed both streams using the `fclose` function. Every `fopen` call should be accompanied by an `fclose` call. Let us modify our Planck function program to output the data to a file:

```

#include <stdio.h>
#include <math.h>
double planck(double wave, double T);

int main()
{
    double wave,T;
    double result;
    char outfile[80];
    FILE *out;

```

```

printf("\nEnter the temperature in Kelvin > ");
scanf("%lf",&T);
printf("\nEnter the name of the output file > ");
scanf("%s",outfile);

if((out = fopen(outfile,"w")) == NULL) {
    printf("\nCannot open %s for writing\n",outfile);
    exit(1);
}

wave = 500.0;

while(wave <= 12000.0) {
    result = planck(wave,T);
    fprintf(out,"%7.1f  %e\n",wave,result);
    wave += 500.0;
}

fclose(out);

return(0);
}

double planck(double wave, double T)
{
    static double p = 1.19106e+27;
    double p1;

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

```

Exercise 3.4: Use your editor to create a file called `inputwave.dat` in your working directory. The contents of this file should be:

500

```
1000
1500
2000
.
.
.
12000
```

i.e. with all the wavelengths between 500 and 12000Å (please fill in the dots appropriately!!). Modify the above program so that it reads in one line of this file for every passage through the loop. You can do this by changing the conditional in the `while` statement to something like this:

```
while(fscanf(in,"%lf",&wave) != EOF) {
```

This will read in one line per passage through the loop and exit the loop when the end of the file (EOF) is encountered.

3.3 An Aside: Incorporating GNUPLOT in your program

So far we have employed `gnuplot` as a standalone program, but it is possible to use `gnuplot` commands in your program and thus incorporate graphics into your program. As an example, let us modify the last example program to do this. Add the include file `<stdlib.h>` to the program preamble, modify the line `FILE *out;` to `FILE *out,*rsp;` and then, after the `fclose(out)` statement, add the following code:

```
if((rsp = fopen("gnuplot.rsp","w")) == NULL) {
    printf("\nCannot open gnuplot.rsp for writing\n");
    exit(1);
}
fprintf(rsp,"plot '%s' using 1:2 with lines\n",outfile);
fprintf(rsp,"pause mouse\n");
fprintf(rsp,"replot\n");
fclose(rsp);
system("gnuplot gnuplot.rsp");
return(0);
```

3.4 More on Vectors and Arrays

When we introduced the concept of variable (part I), we mentioned that arrays of variables could be declared easily, using statements like:

```
double r[10];
int x[200],y[3][5];
```

The entries of **r** then run as **r[0]**, **r[1]**, ..., **r[9]**, the entries of **x** as **x[0]**, **x[1]**, ..., **x[199]** and the entries of **y** as

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

This way of *allocating* memory for vectors and arrays is useful if 1) the vector or array is not very large or, 2) one knows the size of the vector or array before the program is run. If the program needs to be run with different sized vectors or arrays each time it is run (for instance, you might need to analyze different data sets with different numbers of data points), it is useful to *dynamically* allocate memory for your vector or array at runtime. This can be done using the built-in C-function `calloc`, but as we mentioned in lecture 1, two very useful functions `vector` and `matrix` have been provided for you in `comphys.c`. The `vector` and `matrix` functions can be used in the following way: Let us suppose that we want to dynamically allocate space for vectors **r**, **x** and array **y** so that they have the same dimensions as in the example above. We can do it as follows

```
double *r;
int *x,**y;
.
.
.
r = dvector(0,9);
x = ivector(0,199);
y = imatrix(0,2,0,4);
```

First, notice that there are a number of versions of `vector` and `matrix`. To allocate space for an integer vector, use `ivector`, a floating point vector, `vector` and for a double precision floating point vector, `dvector`. The same

goes for `matrix`. Notice as well that the functions `vector` and `matrix` give us the ability to define vectors and matrices which are not *zero-offset*, i.e. whose first index is not 0. For instance, if we want `x` to go from `x[1]` to `x[200]`, we could use the statement

```
x = ivector(1,200);
```

Unit offset vectors can be quite convenient. To use the functions `vector` and `matrix` in our programs, we must include the following statement right at the beginning of our programs:

```
include "comphys.h"
```

and then link the file `comphys.c` in with your program. Indeed, as we go through the semester, we will use a number of functions from `comphys.c`. Most of these functions are from the book *Numerical Recipes in C*, Second Edition by Press et al., published by Cambridge University Press. This is one of the best books ever written, and should be in the library of all physicists and astronomers.

We have already seen in part I how vectors can be *initialized*. We can similarly initialize arrays. For instance, the array `y` can be initialized with values in the following way:

```
int y[3][5] = {{ 1, 8, 3, 5, 2},
               { 0, 1, 4, 4, 2},
               { 99,456, 23, 12, 5}};
```

One does not have to use `vector` to create a unit offset vector. For instance, let us suppose we have declared and initialized a vector `c` in the following way:

```
double c[4] = {1.865,3.449,1.23e+04,0.0045};
```

which clearly goes from `c[0]` to `c[3]`. Let us suppose we want to define a vector that has the same entries, but is unit offset. This is easy to arrange, and can be done as follows

```
double c[4] = {1.865,3.449,1.23e+04,0.0045};
double *C;
```

```
C = c-1;
```

The vector `C` will now go from `C[1]` to `C[4]` with the same entries as `c`, i.e., `C[1] = 1.865`, `C[2] = 3.449`, etc.

Exercise 3.5: Modify your program from exercise 3.2 to read the values from the file `inputwave.dat` into the vector `wave` which has been allocated space using the function `dvector`. Then modify the *while loop* to use these values in the computation of the Planck function.