3 Introduction to C – part III

3.1 Functions

We have already learned in class that a C-program is made up of a number of functions. Every C-program has at least one function – the `main` function. C-programs can also make use of built-in functions such as math functions and input/output functions, and we have seen examples of how to use functions that are included in comphys.c (see exercise 2.7). In this lecture, we are going to learn how 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 Lecture 2.

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

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

  printf("\nEnter the temperature in Kelvin > ");
  ni = 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;
  }
  return(0);
}
```

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

  p1 = p/(pow(wave,5.0)*(exp(1.43879e+08/(wave*T)) - 1.0));
  return(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 function that called it (in this case 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 actual function code is enclosed in brackets. Note that the code for `planck` uses the parameters passed to it by the `main` function, does a calculation, and then returns a number `p1` to the `main` function. In the `main` function, 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 calling 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 other functions 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 pass their addresses, such as in the following 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;
  int ni;

  printf("\nEnter the temperature in Kelvin > ");
```

```
  ni = 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;
  }
  return(0);
}

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

  p1 = p/(pow(wave,5.0)*(exp(1.43879e+08/(wave*T)) - 1.0));
  return(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 multiply T by 2.5 and pass it back to `main`. Print the variable `T` out in `main` to verify that it has indeed been changed to 250.

`Exercise 3.2:` Rewrite your program in Exercise 3.3 and pass `T` by *value*. Show that even though `T` is multiplied by 2.5 in the `doit` function, its value remains the same in the main function.

`Exercise 3.3:` Rewrite your program from `Exercise 3.1`, but now declare `T` as an external variable. Verify that after calling `doit` the value of `T` in `main` is 250.0.

`Exercise 3.4:` In our traditional mathematical world, we permit numbers with an infinite number of ndigits. Real-world arithmetic *defines* 1/3 as that unique positive number that, when multiplied by 3, yields the integer 1. 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 1/3, it approximates it so that when it is multiplied by 3, the result is so close to the integer 1 that it is acceptable. The computer *error* associated with approximation arises from *truncation* or *rounding*. For example, the actual decimal value of 1/3 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 the 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 $E_n = C^nE_0$, 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

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

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:

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

Run this program using the command:
`./program_name tolerance > ex34.out`
where program_name is the name that you assign the executable program at compile time,
`gcc -o program_name ex34.c -lm`
and tolerance is either a decimal or scientific notation value. Plot the results using gnuplot:

```
gnuplot> plot "ex34.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,ni;
  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) > ");
      ni = 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);
}
```

Read: The C Programming Language: Sections 4.1 – 4.7

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 data from and output data to, respectively, 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:

```
#include <stdlib.h>  /* Needed for exit() */

double x,y;
int i,ni;
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++) {
  ni = 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 that check 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 <stdlib.h> /* Needed for exit() */
#include <math.h>
double planck(double wave, double T);

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

  printf("\nEnter the temperature in Kelvin > ");
  ni = scanf("%lf",&T);
  printf("\nEnter the name of the output file > ");
  ni = 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 = p/(pow(wave,5.0)*(exp(1.43879e+08/(wave*T)) - 1.0));
  return(p1);
}
```

`Exercise 3.5:` 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 condition 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.

`Exercise 3.6:` Because it is so *boring* to have to always check whether the file pointers returned by `fopen` have the value NULL, it is possible to write one's own version of `fopen` that does that check for you. Call that function `ffopen`. It will be called exactly like `fopen`, and its definition is identical to that of `fopen`:

```
FILE *ffopen(char *file, char *mode)
```

The function `ffopen` can call `fopen`, but include in the `ffopen` code the check to make certain that the file pointer returned by `fopen` is not a NULL. If it is a NULL, use `exit(1)` to exit the program. To use this `ffopen` function in future programs, simply copy the code to the bottom of your program, and include the definition at the top of the program.

Read: The C Programming Language: Sections 7.5 – 7.7

3.3 An Aside: Incorporating GNUPLOT in your program

So far we have employed `gnuplot` as a stand alone 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. 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);
if(system("gnuplot gnuplot.rsp") == -1) {
   printf(``\nCommand could not be executed\n'');
   exit(1);
}
return(0);
```

`Exercise 3.7:` In the shared directory and on the class website you will find a data file called `vostok_co2.dat` which records, in two columns, the date (in years before the present) in the first column and the CO_2 concentration, in parts per million (ppm) in the second column. These data are

from the Vostok ice core drilled in Antarctica. Write a program that will read in these data using a while loop, and in that loop, convert the CO_2 data into a rough global temperature relative to today (ΔT), using the following equation

$$\Delta T = 2.5(\mathrm{pCO_2} - 365.0)/280.0$$

where $\mathrm{pCO_2}$ is the concentration of CO_2 in ppm. In that same loop, write the date and temperature data into a new file. Then, use `gnuplot` to plot the temperature as a function of the date.

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 I, 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 statements right at the beginning of our programs:

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

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.

If you use `vector` or `matrix` or any other method of dynamically allocating memory in your program, be certain to *free* that space before exiting the program. For instance, for every version of `vector` there is a corresponding version of `free_vector`. For instance, let us suppose that we allocated space in the following way:

```
float *x;

x = vector(1,500);
```

You should then, at the end of your program or function, say right before the `return` statement, free that space with the following code:

```
free_vector(x,1,500);
```

We have already seen in Lecture 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}};
```

Unfortunately, it is not so easy to initialize a vector or matrix which has been defined using the functions `vector` or `matrix`. One way of doing so is to stuff the vector or matrix with data read in from a file (see Exercise 3.9).

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.

One can in principle do the same thing for matrices, but the implementation of the offset is quite a bit more complex.

`Exercise 3.8:` Modify your program from Exercise 3.5 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. Make certain that you free the space allocated for `wave` at the end of the program with `free_dvector`.

`Exercise 3.9:` On the class website, and also in the shared directory, you will find the datafile `vostok_co2.dat` which consists of two columns, the first the number of years before the present, and the second the concentration of CO_2 in the atmosphere in units of parts per million by volumne (ppmv). You may wish to plot these data out with `gnuplot` before you write your program. Write a program that reads in these data into two vectors. The program should count the number of data points as the data are being read in. In a `for` loop determine the maximum and minimum concentrations of CO_2 over the past 414,000 years, and print to the screen those values plus the corresponding dates. In addition to that, calculate the average CO_2 concentration during the *Eemian* (that is, the period between 112,000 and 130,000 years ago) and during the Illinoian ice age (from 138,000 – 175,000 years ago).

Read: The C Programming Language: Chapter 5

3.5 Best Practices in C Programming

As in every field, there are certain "best practices" in C-programming that can help you to avoid mistakes and to make it easier for others to read your code. Here are some, not necessarily in order of importance. Pay attention to them, as they will help you to achieve a better grade in this class and make you a better programmer!

1) Always declare your variables at the top of the function, *never* in the body of the code.
2) Use indentation to make it easier to read and debug a code. For instance, the code inside a `while` or `for` loop should be indented, as should the code inside an `if` or `else` block. Look at the examples in these notes and in The C Programming Language book for examples of proper indentation. Certain editors, such as `emacs`, will help you to achieve proper indentation; indeed, using an editor like `emacs` or `gedit` will help ensure against common errors such as unmatched parentheses, forgotten semicolons, etc.
3) If your variables require initialization via a mathematical formula, initialize those variables at the beginning of the code, not as part of the variable declaration. One of the most common errors is to use an equation to initialize a variable that contains variables that are not yet declared or initialized themselves!

4) Put the `main` function at the top of the file; this makes it easier for others to find it without searching through miles of code.
5) Document your code with comments. At the top of every program that you write, include your name, the purpose & function of the code, and the date it was written. If it has gone through numerous versions, number the version and include that version number in the comments. Throughout the code place comments, especially in sections that are not self-explanatory. Avoid pedantic comments; strive for a happy medium!
6) Make certain that if you allocate memory for a vector or array, that you *free* that space at the end of the program or function. Likewise, if a file is opened, make certain that it is closed before the end of the program or function.
7) Always check a file pointer returned by `fopen` to make certain it is not `NULL`.
8) Give a user-written function only as much power as it requires to do the job. For instance, if it is not necessary to pass parameters to a function by reference (which gives the function the power to modify those parameters), pass those parameters by value only. Avoid external variables unless absolutely needed.
8) Write your code in a *modular* way, i.e. most of the meat of the program should be carried out in user-written functions, not in `main`. The `main` function should, within reason, only be concerned with input and output and the calling of functions that do most of the work. This "rule" obviously is more important for long and complex programs than for short programs where most of the work is reasonably carried out in `main`.

Some Rules of Programming Etiquette.

1) Never use someone else's code without permission or appropriate attribution! This includes code that you find on the internet.
2) Whenever you distribute code, as much as possible make that code open source so that others can benefit from your work.
3) In contradiction to rule (2), in this class do not share code with your classmates, and certainly do not filch code from others! Unless otherwise indicated, you are expected to work independently.