Reading Material

From *Teach Yourself C*:
- Chapter 6: 6.5 – 6.7
- Chapter 12: 12.6 and 12.7 (especially example 3)
- Chapter 7: § 7.1
- Chapter 3: § 3.5

From *Numerical Recipes*:
- Chapter 1
- Chapter 8: sections 0, 1, 2
A. Pointers and Two-Dimensional Arrays

Just like in the case of one-dimensional arrays, pointers can be associated with two-dimensional arrays. For our purposes, there are two main advantages in doing this: (i) we can pass two-dimensional arrays to functions using pointers in a “call-by-reference” mode, and (ii) we can define and allocate memory to two-dimensional arrays even if we do not know the dimensions before the code runs (dynamic memory allocation).

There are several different ways of associating pointers with two-dimensional arrays. In what follows we describe one of those ways that will help us in using Numerical Recipes routines for the rest of the quarter.

Think of the image of a two-dimensional array as a collection of rows and columns of square boxes. We will use the concept of associating a pointer with one-dimensional arrays to associate a pointer with each of the rows of the two-dimensional array. Each of these pointers (let us call them row-pointers), points to the first element of each row. Then memory is allocated in such a way that each of these pointers can be associated with all elements in that row (across all columns).

We can also construct an array of pointers, each element of which holds the address of the row-pointers. This is an array of pointers and its length is equal to the number of rows in the two-dimensional array we would like to construct. In general an array of pointers can be declared as follows:

\[
data\text{-type}\ \star\ name\ of\ array\ of\ pointers[\ length\ of\ array] ;
\]

However, there is a problem with the above declaration: one has to know the length of the array of pointers, i.e., the number of rows of the two-dimensional array, when writing the program. The only way to avoid this problem is to use another pointer to point to this array of row-pointers! This is called a pointer to pointer-array structure and it is declared as follows:

\[
data\text{-type}\ \star\star\ name\ of\ pointer\ to\ pointer;
\]

Note the similarities and differences of the above notation compared to the declaration of a “simple” pointer. Note also that, just like “simple” pointers, at the declaration line it is not at all clear whether the pointer will pointer to an array of pointers (in this case) or just to another “simple” pointer. Remember that to associate a “simple” pointer with an array we have to allocate enough memory for all the array elements. The same is needed in the case of a pointer that points to a pointer array.
For example:

```c
int **p2D;
p2D=(int **) malloc (Nrow*sizeof(int*));
```

The above command allocates enough memory for `**p2D` to hold `Nrow` pointers, each of which can be accessed by `*p2D[index]`, where index runs from 0 to `Nrow-1`. But we are not done yet! Each of the pointers in this array must be allocated enough memory to hold all the elements (across columns) at a given row. This can be done with a loop that is repeated a number of times equal to the number of columns:

```c
for (i=0; i<Nrow; i++) {
p2D[i]=(int *) malloc (Ncol*sizeof(int));
}
```

Then the elements in this two-dimensional structure can be accessed as: `p2D[i][j]`, where the index `i` corresponds to rows and the index `j` corresponds to columns. This notation is identical to the notation used so far for two-dimensional arrays (see notes from lecture 3), with the only difference that the name of the array is now the name of a pointer-to-pointer-array. This notation can be used to access the array elements, assign values to it, print them, do arithmetics etc.

**Call-by-reference**

Once a two-dimensional array has been defined and memory has been allocated for it, it can be passed to another function using the pointer-to-pointer structure.

The function-calling line may be:

```c
function-name (pointer-name);
```

The function-declaration line may be:

```c
function-name (int **pointer-name);
```
B. Functions that return Pointers

We have already discussed that functions can return variables of any data types. C actually allows us to write functions that return pointers or pointers-to-pointers. The declaration of such functions follows all the rules we have learned so far:

```c
pointer-type *function-name (input parameters) { ... 
and
pointer-to-pointer-type **function-name (input parameters) ... 
```

Here is one example where functions returning pointers are useful: we can write-up the process of dynamic memory allocation for arrays in separate functions that we keep aside and we use again and again every time we need arrays in codes for the rest of the quarter (see problems 3.1 and 3.2). In this case we can construct functions that: take array dimensions (one-dimensional or two dimensional) as input; return a pointer or a pointer-to-pointer-array for which appropriate memory has been allocated. Note that different functions will have to be used for one- or two-dimensional arrays and for different data types (int, float, or double).

NOTE: All C routines from Numerical Recipes (NR) that deal with arrays follow a convention different than that of standard C: instead of using arrays where elements have indices from 0 to N-1, they use arrays where element indices run from 1 to N. The way we will deal with this is: always define arrays with one extra element: N+1 instead of N. For example, let us say that we need to use a NR routine with an array with elements from 1 to 5. If we define an array with length equal to 5, then only elements with the following indices exist: 0,1,2,3,4 but not 5. However we can define an array of length 6 (remember: N+1), which will have elements with indices: 0,1,2,3,4,5 and we just use part of this array (only the elements with indices 1,2,3,4,5). Obviously this leads to waste of memory, but for our purposes in this course, we do not have to worry about this.
C. Pointers to Functions

Even though in C functions are not variables, we can still define *pointers to functions*, which can be assigned, passed to functions, returned from functions, etc. In fact in C functions are similar to arrays in the following sense: the name of a function is the address of the function (or a pointer to the function), just like the name of an array is the address of the array (or a pointer to the array).

The most common use of pointers to functions comes up when we want to pass those pointers to other functions. In general-purpose routines like those in Numerical Recipes this is done very often as we will see soon. When we talk about “passing pointers-to-functions to functions”, we mean that we would like to specify a function as an input parameter to another function.

Consider the following example: Let us say that we want to print two columns of N rows, where the first column corresponds the number of the row and the second column corresponds to the inverse of the row, or the inverse-square of the row, or the square-root of the row. One way is to write three separate functions that print the two-column structures. Another way is to create a *single* function, let us call it `make2C`, that can produce either the above outputs depending on on what function is passed to `make2C`. In other words we can pass a function that calculates the inverse (inv), or the inverse-square (inv2), or the square-root (sqr) of a number as an input parameter to `make2C`.

What we need to learn is: (i) how to include a function in the “calling” statement of another function, and (ii) how to include a function as an input parameter in the declaration line.

(i) We can include a function in the calling line by just including its name; effectively a pointer-to-function is passed to `make2C` and the name of the function *is* the pointer. For example:

```
make2C(N,inv);
```
(ii) We can include a function as an input parameter by providing the data type of its output, the pointer function, and the data type of its input parameters. The following declaration says that function `make2C` takes two input parameters: one is an integer variable, the other is a pointer to a function, which takes an integer as input and returns a float value:

```c
void make2C (int N, float (*func)(int)) {
    int i;
    for (i=1; i<(N+1); i++) {
        printf("%i %.3f \n", i, (*func)(i));
    }
}
```

NOTE: `float (*func)(int)` is very different from `float *func(int)`. The first one represents a `pointer to a function` and the second represents a `function that returns a pointer`.

The functions `inv`, `inv2`, and `sqr` are declared as usual. For example:

```c
float inv(int i) {
    return (1.0/((float) i));
}
```

The result of the above exercise is that we have created a "general-purpose" function `make2C` that prints two columns of numbers, where the first column is the number of row, and the second column is the result of any function that was passed to `make2C` using a pointer to that function.
D. Function Prototypes, Header Files, and Multi-File Programs

For a number of different reasons, it is customary in C (and sometimes necessary) to use function prototypes separately from the function declarations and definitions. Function prototypes are just one-line statements that identify: (i) the data type of the function output, (ii) the name of the function, and (iii) the data type of the input parameters. Function prototypes are listed before the main function and just after the “include” lines. The actual declarations and definitions of the functions are listed after the details of the main function.

In the example from the previous section, we would have four function prototypes, one for each of the four functions: make2C, inv, in2, and sqr. The function prototype for make2C would be:

```c
void make2C (int, float (*func) (int) );
```

The function prototype for inv would be:

```c
float inv (int);
```

NOTE the semicolon at the end of the line!

Then the main function would follow and the actual functions would follow after the main function.

Often for clarity or convenience, we put a set of function declarations and definitions in a separate source file (e.g., functions.c) with the intention to compile it separately from the main source code (e.g., main.c), but then combine the two object files in the linking process to create a single executable file. However calls to these external functions may appear in the main source code and vice versa. What we need to do is include all all function prototypes in all source files. The most efficient way to do this is to write all function prototypes in a separate file and include it in all source codes as a header file:

```c
#include "functions.h"
```

NOTE the double quotes instead of < and > - this tells the code to search for the header file in the same directory where the source files are.

NOTE: to do all what is described above in the Visual C++ environment, you need to open new files within an existing workspace (see notes from lecture 1 for how we open files within a workspace) and choose the type of file (header or source).
Programming Hints

• User pointer names that can easily give you a hint about which variable they point to.

• Remember the sequence of steps when it comes to pointers and arrays: first we define the pointer with its type, then we allocate appropriate memory, and then we can assign values to its elements (“load” them).

• To avoid confusion, when passing variables or arrays to functions using pointers, using different names for them in the calling and declaration lines.