GLG 490 Numerical Methods Intro to C Programming – part 3

<u>Arrays</u>

There are essentially 2 ways for C to represent multidimensional sequences of data (vectors, matrices, tensors, etc). One is through the use of arrays and the other through pointers. We'll talk more about pointers later, but note that <u>arrays and pointers are intrinsically different creatures.</u> They are often utilized in the same fashion, but there are slight differences. In this course we'll mainly use pointers because a) they can be dynamic, b) they can be larger, and c) they are required for most functions (such as those in numerical recipes). But, for simple tasks, an array is easier to deal with.

Arrays can be of any data type (i.e. int, float, double, *fp, etc.) and they can have multiple dimensions (i.e. 3x2x10).

A very important difference between C and Fortran is that arrays in C start at position 0, not at position 1!

For example: If there is an array defined as

int vector_A[4]

Then it has the following positions: A[0], A[1], A[2], A[3]

To be honest, most of us C scientific programmers still follow the Fortran convention so we usually ignore the A[0] and just use the A[1], A[2], and A[3] positions. We'll typically follow the same convection here in this course. The reason that we will do this is because it will potentially cut down on possible errors when using mathematical formulas. When dealing with vectors, matrices, and tensors, we often mathematically use 1 as the lowest index.

Another important point – after declaring an array, you cannot assume that the values of its members will be 0!

tutorial9.c takes the dot product of 2 vectors:

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

/* this code will help learn the basics of arrays */ /* we will take the dot product of 2 vectors */

int main(void)
{

ł

/* Declare two double vectors of length four (it will have positions at 0,1,2,and 3) */ /* We will ignore the 0 positions */

double A[4]; double B[4]; int kk; double dot_product;

/* We'll be extra cautious and initialize the array with 0.0 values */

/* Lets fill A and B with some typical numbers */

A[1]= 5.0; A[2]= 2.0; A[3]= 3.5; B[1]= -1.0; B[2]= 3.0; B[3]= -1.5;

/* Lets print out the vectors (or at least the parts we're interested in)*/

```
for (kk=1;kk<=3;kk++)
{
fprintf(stdout,"A[%d] = %f B[%d] = %f\n",kk,A[kk],kk,B[kk]);
}
```

/* Take the dot product of A and B */

```
dot_product=A[1]*B[1]+A[2]*B[2]+A[3]*B[3];
```

fprintf(stdout,"The dot product of A and B is: %f\n", dot_product);

return 0;

}

Here is a code that multiplies 2 matrices. We also introduce an integer vector and n=3 dimensional array.

tutorial10.c

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

```
/* this code will help learn the basics of arrays */
/* we will multiply two matrices */
```

```
int main(void)
{
```

/* Declare two double matrices of length four by four (it will have positions at 0,1,2,and 3) */ /* We will ignore the 0 positions */

```
/* NOTE: remember, there is nothing special about the word matrix below, its */
/* just a name I've given it */
```

double matrix_A[4][4]; // Note the new style of comment here // We will use my_matrix_A as a 3x3 matrix // ignoring the 0 positions.

```
double matrix_B[4][4]; // ditto
double product_of_A_and_B[4][4];
```

int my_silly_little_integer_array[10]; double my_big_ass_matrix[10][16][21];

int irow,icol; int iz;

/* Now lets give them some meaningless values */

```
for (irow=1;irow<=3;irow++)</pre>
  {
  for (icol=1;icol<=3;icol++)
  {
      matrix_A[irow][icol]=(1.0*irow)*(1.0*icol);
      matrix B[irow][icol]=(2.0*irow)*(3.0*icol);
  }
  }
  for (irow=1;irow<=10;irow++)</pre>
  {
     my_silly_little_integer_array[irow]=irow;
  }
  for (irow=1:irow<=3:irow++)
  for (icol=1;icol<=3;icol++)
  for (iz=1;iz<=3;iz++)
  {
     my_big_ass_matrix[irow][icol][iz]=(1.0*irow*icol*iz);
  }
  }
  }
/* Lets multiply matrix A and matrix B */
/* There are algorithms we could make to do this, but we'll be simple here */
 product_of_A_and_B[1][1]=matrix_A[1][1]*matrix_B[1][1]+
                 matrix_A[1][2]*matrix_B[2][1]+
                 matrix_A[1][3]*matrix_B[3][1];
```

```
product_of_A_and_B[1][2]=matrix_A[1][1]*matrix_B[1][2]+
matrix_A[1][2]*matrix_B[2][2]+
matrix_A[1][3]*matrix_B[3][2];
```

- product_of_A_and_B[1][3]=matrix_A[1][1]*matrix_B[1][3]+ matrix_A[1][2]*matrix_B[2][3]+ matrix_A[1][3]*matrix_B[3][3];
- product_of_A_and_B[2][1]=matrix_A[2][1]*matrix_B[1][1]+ matrix_A[2][2]*matrix_B[2][1]+ matrix_A[2][3]*matrix_B[3][1];
- product_of_A_and_B[2][2]=matrix_A[2][1]*matrix_B[1][2]+ matrix_A[2][2]*matrix_B[2][2]+ matrix_A[2][3]*matrix_B[3][2];
- product_of_A_and_B[2][3]=matrix_A[2][1]*matrix_B[1][3]+ matrix_A[2][2]*matrix_B[2][3]+ matrix_A[2][3]*matrix_B[3][3];
- product_of_A_and_B[3][1]=matrix_A[3][1]*matrix_B[1][1]+ matrix_A[3][2]*matrix_B[2][1]+ matrix_A[3][3]*matrix_B[3][1];
- product_of_A_and_B[3][2]=matrix_A[3][1]*matrix_B[1][2]+ matrix_A[3][2]*matrix_B[2][2]+ matrix_A[3][3]*matrix_B[3][2];
- product_of_A_and_B[3][3]=matrix_A[3][1]*matrix_B[1][3]+ matrix_A[3][2]*matrix_B[2][3]+ matrix_A[3][3]*matrix_B[3][3];

/* yeah, it would have been nicer to do this in a more systematic way */

```
for (irow=1;irow<=3;irow++)
{
for (icol=1;icol<=3;icol++)
{
    fprintf(stdout,"AxB[%d][%d]: %f\n",irow,icol,product_of_A_and_B[irow][icol]);
}
</pre>
```

```
/* print this whole thing out */
```

return 0; }

One useful programming tip is that you can use a file pointer array to open many files at one time. Just a caution though – the number of files you can open at one time is limited. That number is usually machine dependent (16 is about the number I can get). You can see how to tell in the following example: tutorial11.c

tutorial11.c

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

int main(void)
{

/* Lets open a bunch of files and store them in directory temp_tutorial11/ */

/* We'll write to 10 files simultaneously */

```
int number_of_files=10;
FILE *fp[11];
int ifile;
```

/* We'll also have 10 file names to create */ /* We'll create a 2-d character array to do this */

```
char filename[11][200];
```

/* "FOPEN_MAX" is a macro in stdio.h which tells the safest number of files to have open at once */

fprintf(stdout,"I hope we don't have more than %d files open!\n",FOPEN_MAX);

```
/* make the subdirectory temp_tutorial11 */
```

```
system("mkdir -p temp_tutorial11");
```

```
/* now lets open the files */
```

}

```
for (ifile=1;ifile<=number_of_files;ifile++)
{</pre>
```

/* now lets give it a filename that includes a 3 digit number */

sprintf(filename[ifile],"temp_tutorial11/temporary_file_%.3d.out",ifile);

```
fp[ifile]=fopen(filename[ifile],"w");
```

/* now for some error control, check to make sure file could open */

```
if (fp[ifile]==NULL)
{
    fprintf(stdout,"Man, thats a problem - cant open %s\n",filename[ifile]);
    exit(10);
}
```

/* So now, all of our files are opened, we can write to them at will */

```
/* lets prints something dumb in each one */
```

```
for (ifile=1;ifile<=number_of_files;ifile++)</pre>
```

```
fprintf(fp[ifile],"Hi, I'm file number %d\n",ifile);
```

/* lets say something special to file number 2 */

```
fprintf(fp[2],"I am special because I am file number 2\n");
```

```
/* now, close all files */
/* also not, how we dont need brackets if we only have one command */
```

```
for (ifile=1;ifile<=number_of_files;ifile++)
fclose(fp[ifile]);</pre>
```

```
return 0;
}
```

{

}

Now, check the directory you created to see if your files are there. Use vi to check them out, paying particular attention to file number 2.

An additional note on file processing

Its not often that you will want to have all of your files open at the same time given the (typical 16) maximum, however, you may want to read in many files or write to many files. As you saw in tutorial11, having a character array can be very useful in this manner. Here is an example of a code that will produce many files with a systematic name. Keep the temp_tutorial11 directory for now – that's where we will put them.

tutorial12.c

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

```
int main(void)
{
```

/* we'll open the 10 files that we just created with tutorial 11 */ int number_of_input_files_to_read=10; FILE *fp_input_file;

```
/* we'll also write 100 files */
int number_of_output_files_to_write=100;
FILE *fp_output_file;
```

int ifile;

/* here is our character arrays for filenames */

char input_filename[11][200];

char output_filename[101][200];

```
/* make the subdirectory temp_tutorial11 */
```

system("mkdir -p temp_tutorial11");

/* now lets open the files we already created in tutorial 11*/ /* We'll append information to them */

```
for (ifile=1;ifile<=number_of_input_files_to_read;ifile++)
{</pre>
```

/* We'll have to have the filename perfectly */

sprintf(input_filename[ifile],"temp_tutorial11/temporary_file_%.3d.out",ifile);

/* notice the "a" for append in the following statement */

```
fp_input_file=fopen(input_filename[ifile],"a");
```

/* now for some error control, check to make sure file could open */

```
if (fp_input_file==NULL)
{
    fprintf(stdout,"Man, thats a problem - cant open %s\n",input_filename[ifile]);
    exit(10);
}
/* lets write something to the file and close it */
fprintf(fp_input_file,"Tutorial 12 just touched you\n");
fclose(fp_input_file);
```

}

}

/* Now, lets open 100 new files and just write hi */

```
for (ifile=1;ifile<=number_of_output_files_to_write;ifile++)
{
    sprintf(output_filename[ifile],"temp_tutorial11/new_file%.3d.out",ifile);
    fp_output_file=fopen(output_filename[ifile],"w");
    if (fp_output_file==NULL)
    {
        fprintf(stdout,"Man, thats a problem - cant open %s\n",output_filename[ifile]);
        exit(10);
    }
    fprintf(fp_output_file,"Hi, I just made you and now I will close you \n");
    fclose(fp_output_file);
    }
return 0;
</pre>
```

Getting out of loops, goto or break

Sometimes you won't know how long you will want to loop through something, so it may be convenient to get out of the loop before its finished given some condition is satisfied. The "break" statement will kick you out. Another possibility is the "goto" statement which will take you somewhere else (that you define) in the function. A note about "goto" – I use it all the time, but with criticism from others because most programmers consider it bad form and it goes against their coding philosophy. I kind of agree with them, but it doesn't stop me from using goto in a careful manner.

The "break" command immediately kicks you out of the current loop.

The "goto" command takes you where you declare.

(example – the statement **goto here_man;** takes you anywhere in the function where you have written **here_man:** Note the colon (not semicolon) in this last part!

Lets look at how to use goto and break in the following example: *tutorial13.c*

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

/* tutorial13 */ /* shows how to use break and goto to kick out of a loop*/

/* The example is from kinematics */

/* An apple is falling from a tree, and we don't */

/* know how many timesteps it will take to hit the ground */

```
/* we'll use the equation x=0.5*g*t^2+initial_distance */
```

```
/* we don't know how many timesteps to use, so we use a lot of them */
```

```
int main(void)
{
```

```
double g= -9.98;  // acceleration of gravity
double initial_distance=3.0; // distance from ground in meters
```

```
int maximum_timesteps=10000000;
int istep;
double time;
double time_step=1.0e-6; // very small timestep
double x;
```

```
time=0.0;
x=initial_distance;
for (istep=1;istep<=maximum_timesteps;istep++)
{
```

x=0.5*g*time*time + initial_distance;

// check if it hit the ground yet

```
if (x<=0.0)
{
  break;
}
```

```
// now increment the time
      time=time+time step;
```

}

fprintf(stdout, "The apple fell %f meters in %.8f seconds\n", initial_distance, time);

```
/* Now we'll do the exact same thing, but with a goto */
```

```
time=0.0;
x=initial distance;
for (istep=1;istep<=maximum_timesteps;istep++)</pre>
{
```

x=0.5*g*time*time + initial_distance;

// check if it hit the ground yet

```
if (x<=0.0)
{
  goto apple_hit_the_ground;
}
```

```
// now increment the time
      time=time+time step;
```

}

/* Heres where the goto goes to - note the semicolon */

```
apple hit the ground:
   fprintf(stdout,"The apple fell %f meters in %.8f seconds\n",initial_distance,time);
```

return 0; }

Note: You would never want to use the condition if (x==0.0) because the chances of that ever happening are extremely low!

While

Another convenient statement is the "while" which will keep you in a loop until a certain condition is satisfied. Personally, I don't use it that often because there is always the possibility of the infinite loop if the condition is never satisfied. We can modify the last tutorial, using the while statement instead.

Tutorial14.c

#include<stdio.h>

#include<stdlib.h>

```
/* tutorial14 */
/* shows how to use while*/
```

```
/* The example is from kinematics */
/* An apple is falling from a tree, and we don't */
/* know how many timesteps it will take to hit the ground */
```

```
/* we'll use the equation x=0.5*g*t^2+initial_distance */
```

```
int main(void)
```

```
{
```

```
double g= -9.98; // acceleration of gravity
double initial_distance=3.0; // distance from ground in meters
```

```
int maximum_timesteps=10000000;
int istep;
double time;
double time_step=1.0e-6; // very small timestep
double x;
```

```
time=0.0;
while (x>0.0)
{
    x=0.5*g*time*time + initial_distance;
    time=time+time_step;
}
```

/* note that we printed out "time-time_step" - can you guess why? */

fprintf(stdout,"The apple fell %f meters in %.8f seconds\n",initial_distance,time-time_step);

return 0;

}