GLG 490 Numerical Methods
Intro to C Programming – part 6

Allocating memory to pointers

Chunks of memory can be allocated (reserved) to pointers. In this manner, pointers may act as very convenient arrays. 3 reasons why this method is preferred over explicitly using arrays are: a) Chunks of allocated memory may be much larger than an array can, b) pointers can be conveniently transferred to functions, and c) most functions (such as those in numerical recipes) require pointers rather than arrays to be used as function arguments.

To allocate a chunk of memory to a pointer, we use the **malloc** function. **malloc** has only one argument which is the size of the memory chuck to be allocated. **malloc** then returns the starting address of that memory chunk.

For instance, if we've declared a pointer to double named **my_pointer_to_double**, and we want to give it a chunk of memory such that it can store 100 doubles, we do this:

```
double *my_pointer_to_double;
my_pointer_to_double=malloc(100*sizeof(double));
```

In *tutorial25.c*, we see examples of how a double pointer and an integer pointer are given chunks of memory using **malloc**. They can then be used as arrays. We also see an example of how a pointer to a pointer can be used to create a 2-Dimensional array.

Tutorial25.c

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

// tutorial25.c

// This tutorial deals with allocating memory

int main(void)
{

int number_of_rows;
int number_of_columns;
int irow;

// declare a pointer to double
double *dpointer1;

// declare a pointer to integer
int *ipointer1;

// declare a pointer to a pointer to double
double **my_p_to_p;
```

```
//Now, lets allocate memory to these arrays using malloc:

//Lets say we want an array of 100 doubles
// (we'll add a position because we'll ignore the zero position)

     dpointer1=malloc((100+1)*sizeof(double));

//Now we can use "dpointer1" as if
// it were an array of size 101.

    dpointer1[1]=5.7;
    dpointer1[2]=3.2;

    fprintf(stdout,"Position 1 of dpointer1: %f\n",dpointer1[1]);
    fprintf(stdout,"Position 2 of dpointer1: %f\n",dpointer1[2]);

//Similarly, we can allocate memory to ipointer1
//  We'll make it have a size of 21 integers

    ipointer1=malloc(21*sizeof(int));

    ipointer1[1]=3;
    ipointer1[2]=4;
    ipointer1[3]=5;

    fprintf(stdout,"Position 1 of ipointer1: %d\n",ipointer1[1]);
    fprintf(stdout,"Position 2 of ipointer1: %d\n",ipointer1[2]);
    fprintf(stdout,"Position 3 of ipointer1: %d\n",ipointer1[3]);

// Now lets make a 2 dimensional array of size 10x10 doubles
// Again, we add an extra because we ignore the zeros

   number_of_rows=10;
   number_of_columns=10;

// First, we will make an array of pointers
// Note how we use "sizeof() with a pointer as the argument.
// This is because the size of each of these positions is the same
// as that of a pointer.

   my_p_to_p=malloc((number_of_rows+1)*sizeof(my_p_to_p));

// Now for each of these pointer positions, we'll make an array of doubles

   for (irow=1;irow<=number_of_rows;irow++)
   {
       my_p_to_p[irow]=malloc((number_of_columns+1)*sizeof(double));
   }

// Now, we can use "my_p_to_p" as a 2D array

   my_p_to_p[1][1]=1.0;
   my_p_to_p[1][2]=2.0;
   my_p_to_p[2][2]=3.0;

   fprintf(stdout,"Row 2, Column2: %f \n",my_p_to_p[2][2]);
```

```
// When the code finishes, it "frees" the memory.
// However, sometimes it is convenient to free the memory beforehand.
// We'll free our pointers here:

  free(dpointer1);
  free(ipointer1);
  free(my_p_to_p);

return 0;
}
```

Passing a pointer (representing an array) to a function

In *tutorial26.c* we pass 3 pointers to an array. One pointer represents a 1D array. Another is actually a pointer to a pointer, and represents a 2D array. The third is a pointer to pointer to pointer which represents a 3D array.

tutorial26.c

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

// tutorial26.c
// passing pointers to functions


int main(void)
{

int kk;
int pp;
int i,j,k;
void print_double_arrays();

// declare pointers
double *my_1D_array;
double **my_2D_array;
double ***my_3D_array;

// declare the number of positions in each direction
int num_i=3;
int num_j=3;
int num_k=3;

// Lets allocate the 1D array

  my_1D_array=malloc((num_i+1)*sizeof(double));

// give the 1D array some values

  my_1D_array[1]=1.0;
  my_1D_array[2]=2.0;
```

```
   my_1D_array[3]=3.0;

// Lets allocate the 2D array
// (note the sizeof() arguements

   my_2D_array=malloc((num_i+1)*sizeof(my_2D_array));
   for (kk=1;kk<=num_i;kk++)
   {
     my_2D_array[kk]=malloc((num_j+1)*sizeof(double));
   }

// Lets give the 2D array some values

   my_2D_array[1][1]=1.0;
   my_2D_array[1][2]=2.0;
   my_2D_array[1][3]=3.0;
   my_2D_array[2][1]=4.0;
   my_2D_array[2][2]=5.0;
   my_2D_array[2][3]=6.0;
   my_2D_array[3][1]=-4.0;
   my_2D_array[3][2]=1.5;
   my_2D_array[3][3]=9.0;

// Lets allocate the 3D array
// (again, note the sizeof() arguments
// Watch carefully how this is done!

   my_3D_array=malloc((num_i+1)*sizeof(my_3D_array));
   for (kk=1;kk<=num_i;kk++)
   {
     my_3D_array[kk]=malloc((num_j+1)*sizeof(my_3D_array));
     for (pp=1;pp<=num_j;pp++)
     {
        my_3D_array[kk][pp]=malloc((num_k+1)*sizeof(double));
     }
   }

// Lets give the 3D array some values
// We'll set them equivalent to i+j+k

   for (i=1;i<=num_i;i++)
   {
   for (j=1;j<=num_j;j++)
   {
   for (k=1;k<=num_k;k++)
   {
      my_3D_array[i][j][k]=(1.0*(i+j+k));
   }
   }
   }

// We'll pass the pointers to a function that
// will print out the values

   print_double_arrays(my_1D_array,my_2D_array,my_3D_array,num_i,num_j,num_k);
```

```
return;
}

/************************************************/
/*   FUNCTION PRINT_DOUBLE_ARRAYS       */
void print_double_arrays(A,B,C,n_i,n_j,n_k)
   double *A;
   double **B;
   double ***C;
   int n_i;  //number of positions in 1D
   int n_j; // number of positions in 2D
   int n_k; // number of positions in 3D
{
int i,k,j;

// print out 1D array

   fprintf(stdout,"\n1D array:\n");
   for (i=1;i<=n_i;i++)
   {
      fprintf(stdout,"%f\n",A[i]);
   }

// print out 2D array

   fprintf(stdout,"\n2D array:\n");
   for (i=1;i<=n_i;i++)
   {
   for (j=1;j<=n_j;j++)
   {
      fprintf(stdout,"%f\n",B[i][j]);
   }
   }

// print out 3D array

   fprintf(stdout,"\n3D array:\n");
   for (i=1;i<=n_i;i++)
   {
   for (j=1;j<=n_j;j++)
   {
   for (k=1;k<=n_k;k++)
   {
      fprintf(stdout,"%f\n",C[i][j][k]);
   }
   }
   }

return;
}
```

Passing memory addresses of integers or doubles to a function

Typically, if you pass integers and doubles to a function, the compiler makes a copy of those variables for use within that function. If you change the value within the function, it does not affect the values outside of that particular functions. However, when you pass a pointer, you are allowing for memory to be manipulated. Hence, any changes you make within the function will also affect values outside of the function. This is very useful if you would like functions to permanently modify your variables. You will need this for your homework, so pay attention to the following tutorial. The professor will explain further.

Tutorial27.c shows an example of this.

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

// tutorial 27.c

// passing memory to functions
// This tutorial will help with your homework
// Carefully analyze the printed output

// function1 does not modify values
// function2 does modify values

int main(void)
{

double my_real_number;
int my_integer_number;

void function1();
void function2();

// give values to variable

  my_real_number=3.5;
  my_integer_number=2;

// First, we'll pass to function 1
// where the numbers will not be modified here

  fprintf(stdout,"Before passing to function1\n");
  fprintf(stdout,"variable values: %f %d\n",my_real_number,my_integer_number);

  function1(my_real_number,my_integer_number);

  fprintf(stdout,"After returning from function1\n");
  fprintf(stdout,"variable values: %f %d\n",my_real_number,my_integer_number);

// Now, we'll pass to function 2
// Notice that we are passing pointers
// The values change upon returning
```

```
   fprintf(stdout,"Before passing to function2\n");
   fprintf(stdout,"variable values: %f %d\n",my_real_number,my_integer_number);

// Note the amperstands - we are passing memory pointers!

   function2(&my_real_number,&my_integer_number);

   fprintf(stdout,"After returning from function2\n");
   fprintf(stdout,"variable values: %f %d\n",my_real_number,my_integer_number);

// Note the my_real_number and my_integer_number changed!

return 0;
}

/*******************************************/
/* FUNCTION1                               */
void function1(real_number,integer_number)
  double real_number;
  int integer_number;
{

// multiply the number by 2

   real_number=real_number*2.0;
   integer_number=integer_number*2;

   fprintf(stdout,"Inside function 1\n");
   fprintf(stdout,"variable values: %f %d\n",real_number,integer_number);


return;
}

/*******************************************/
/* FUNCTION2                               */
void function2(real_number,integer_number)
  double *real_number;
  int *integer_number;
{

// To modify memory contents, we need a star in front
// real_number and integer_number are memory values
// *real_number and *integer_number are the contents of those memory values

   *real_number=*real_number*2.0;
   *integer_number=*integer_number*2;

   fprintf(stdout,"Inside function 2\n");
   fprintf(stdout,"variable values: %f %d\n",*real_number,*integer_number);

return;
}
```

Passing part of an array to a function

Often, you will not want to pass an entire multi-dimensional array to a function, and instead you may want to only send part of it. For example, if you have a matrix of vectors, you can send only one vector at a time to a function. As usual, this is best seen by example. In *tutorial28.c*, a 2D array contains many vectors. Some of these vectors are passed to a function to compute their dot product.

Tutorial28.c

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

// tutorial28.c

// An example of passing part of a mulidimensional array

// in this example, a matrix contains multiple vectors.
// We will take the dot product of some of these vectors

int main(void)
{

// Declare a large matrix full of vectors.
// Think of vectors making up the columns of this matrix.
// The vectors are 3-dimensional (3 positions)
// As usual, we will ignore all 0 positions (so we add 1)

int number_of_vectors=5;
double **matrix;
int kk;
double dot_product();
double result;

// allocate memory for the vector matrix

// first make an array of pointers - each representing a vector

  matrix=malloc((number_of_vectors+1)*sizeof(matrix));

// now allocate memory for each vector (4 positions each [ignoring the 0])

 for (kk=1;kk<=number_of_vectors;kk++)
 {
   matrix[kk]=malloc(4*sizeof(double));
 }

// Now lets give the vectors some values;

// vector 1
  matrix[1][1]=3.0;  //x-position
  matrix[1][2]=3.0;  //y-position
  matrix[1][3]=3.0;  //z-position
```

```
// vector 2
   matrix[2][1]=1.0;  //x-position
   matrix[2][2]=2.0;  //y-position
   matrix[2][3]=3.0;  //z-position
// vector 3
   matrix[3][1]=4.0;  //x-position
   matrix[3][2]=1.0;  //y-position
   matrix[3][3]=0.0;  //z-position
// vector 4
   matrix[4][1]=3.0;  //x-position
   matrix[4][2]=4.0;  //y-position
   matrix[4][3]=5.0;  //z-position
// vector 5
   matrix[5][1]=-1.0;  //x-position
   matrix[5][2]=4.0;  //y-position
   matrix[5][3]=9.0;  //z-position

// Take the dot product of vectors 2 and 3 using the function
// Look carefully how this is done - we are sending pointers!

   result=dot_product(matrix[2],matrix[3]);

   fprintf(stdout,"vector2 dot vector3 is %f\n",result);

// Lets try another one
// Take the dot product of vectors 1 and 5 using the function
// Look carefully how this is done - we are sending pointers!

   result=dot_product(matrix[1],matrix[5]);

   fprintf(stdout,"vector1 dot vector5 is %f\n",result);

return 0;
}

double dot_product(vecA,vecB)
    double *vecA;
    double *vecB;
{
double dot;

    dot=vecA[1]*vecB[1]+vecA[2]*vecB[2]+vecA[3]*vecB[3];

return dot;
}
```