PRIFYSGOL BANGOR UNIVERSITY

Imperative Programming in C

Laboratory 7: Multi-dimensional Arrays

Dr. Llyr ap Cenydd

Introduction to 2D Arrays

An array is just a collective name given to a group of values. So far we have only used one-dimensional arrays, but arrays can have any number of dimensions – 2D, 3D, 4D etc.

In this lab we are going to be working with two-dimensional arrays. To understand what a 2D array actually looks like, consider the following matrix:

a	b	c
d	e	f
g	h	i

In C, one-dimensional arrays provide a mechanism to access a sequence of data using a single index, such as myArray[0], myArray[1], myArray[2], ... Similarly, C supports the organization of data in a table by specifying a row and a column.

The above matrix A is 3 x 3 – that is, it has three columns and three rows. We can represent an index to each element in the array by specifying a column and row number, using the formula A[m,n], where m is the row number and n the column. So, the first element (a) is stored at A[0,0]. Similarly:

A[0,1] = b
A[0,2] = c
A[1,0] = d
A[1,1] = e and so on.

Declaring a 2D array

A 2D array is basically a list of one-dimensional arrays. As with one-dimensional arrays, 2D arrays can contain data of any type, although each entry in a table must have the same type. To declare a 2D integer array of size x, y we would write something like this:

```
type arrayName[ x ][ y ];
```

Where type is any valid C data type (int, float, double, char, struct etc.), arrayName the array's label and x and y the dimensions. The size of a 2D array must be declared when it is first defined.

Multidimensional arrays can be initialized by specifying bracketed values for each row. For example the following array has three rows and four columns:

```
int myArray [3][4] = {
                          {0, 1, 2, 3},
                          {4, 5, 6, 7},
                          {8, 9, 10, 11}
                         };
```

Note that the actual formatting (where we put the new lines) is completely up to us - here we've placed the rows on different lines just to aid readability. This code would compile ok because the C compiler doesn't care about white spaces.

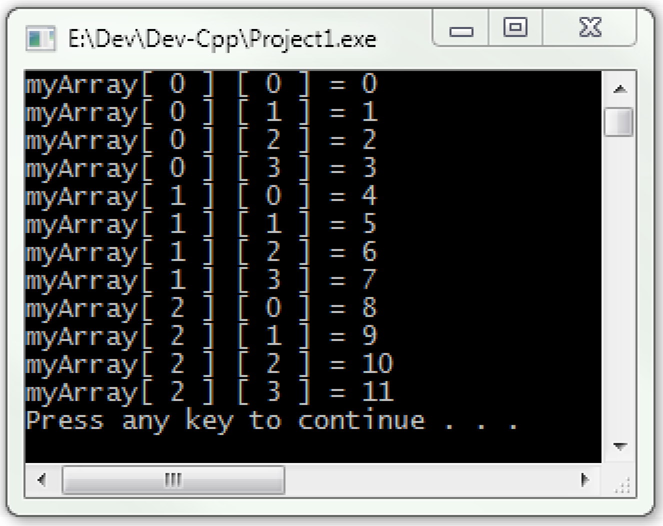
The nested braces are also optional, so the following array initialisation would work exactly the same:

```
int myArray[3][4] = {1,2,3,4,5,6,7,8,9,10,11};
```

Accessing 2D array elements

As with the 9 cell example above, we can access elements in the array using the subscripts - i.e. [row,column]. For example to print out myArray above, we can write the following code:

```
int main(int argc, char *argv[])
{


int myArray[3][4] = {
                         {0,1,2,3},
                         {4,5,6,7},
                         {8,9,10,11}
                         };


int i,j;


for(i=0; i<3; i++)
    for(j=0; j<4; j++)
        printf("myArray[ %d ] [ %d ] = %d \n", i, j, myArray[i][j]);

  system("PAUSE");
  return 0;
}
```


E:\Dev\Dev-Cpp\Project1.exe

```
myArray[ 0 ] [ 0 ] = 0
myArray[ 0 ] [ 1 ] = 1
myArray[ 0 ] [ 2 ] = 2
myArray[ 0 ] [ 3 ] = 3
myArray[ 1 ] [ 0 ] = 4
myArray[ 1 ] [ 1 ] = 5
myArray[ 1 ] [ 2 ] = 6
myArray[ 1 ] [ 3 ] = 7
myArray[ 2 ] [ 0 ] = 8
myArray[ 2 ] [ 1 ] = 9
myArray[ 2 ] [ 2 ] = 10
myArray[ 2 ] [ 3 ] = 11
Press any key to continue . . .
```

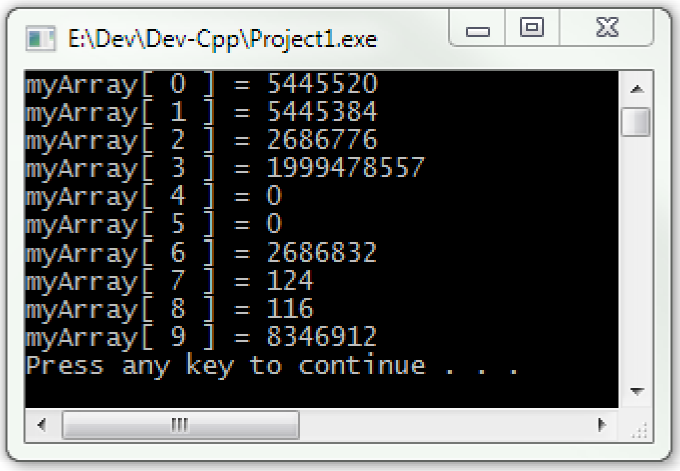
Notice that we are using nested for-loops here to work our way through the contents of the array. Try to visualise how the array is being accessed one element at a time.

Initialising arrays

Some of you might have run into a problem in the Lab 4 assignment where you thought you were declaring a character array of size 256, but later found out strange things were happening in your code, such as symbol characters being printed, if-statements triggering when they shouldn't etc. The most likely explanation for this is that you had not *initialised* your array correctly. For example, the following code creates an array of size 10, then prints out the contents.

```
int main(int argc, char *argv[])
{

  char myArray[10];
  int i;

  for(i=0; i<10; i++)
    printf("myArray[ %d ] = %d \n", i, myArray[i]);

  system("PAUSE");
  return 0;
}
```

However the output looks like this:

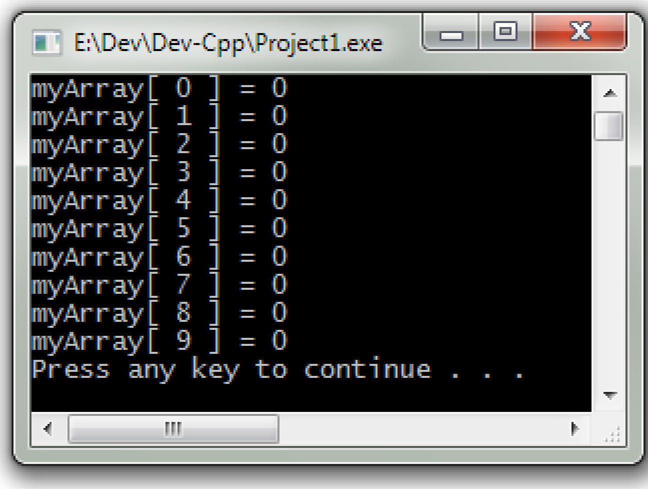


```
myArray[ 0 ] = 5445520
myArray[ 1 ] = 5445384
myArray[ 2 ] = 2686776
myArray[ 3 ] = 1999478557
myArray[ 4 ] = 0
myArray[ 5 ] = 0
myArray[ 6 ] = 2686832
myArray[ 7 ] = 124
myArray[ 8 ] = 116
myArray[ 9 ] = 8346912
Press any key to continue . . .
```

The reason for this behaviour is that while we have declared myArray, we have not initialised it - i.e. we have not told the compiler to set the array elements to anything! The consequence of this is that myArray will still contain what was previously stored at its memory locations. However if we change one line of code:

```
int myArray[10] = {0};
```

We can ensure that all array elements are set to zero:



```
myArray[ 0 ] = 0
myArray[ 1 ] = 0
myArray[ 2 ] = 0
myArray[ 3 ] = 0
myArray[ 4 ] = 0
myArray[ 5 ] = 0
myArray[ 6 ] = 0
myArray[ 7 ] = 0
myArray[ 8 ] = 0
myArray[ 9 ] = 0
Press any key to continue . . .
```

In the previous example we declared an array of size 12 containing the numbers 0-11. While we can initialise arrays by specifying the value of each element like this, what happens if we want to declare a large array, say 256 x 256?

The following two pieces of code do the same thing - they initialise an array of size 256 x 256 and set all the elements to zero.

```
int myArray[256][256] = { {0 } };


int myArray[256][256];
int x, y;
for(x = 0; x < 256; x ++)
{
    for(y = 0; y < 256; y ++)
            {
            myArray[x][y] = 0;
            }
}
```

While both sets of code produce identical 256 x 256 two-dimensional arrays, in the first example we are setting the array elements to zero at *initialisation*, while in the second example we are iterating through and setting each value one by one - which we can do anywhere in our code, not just at initialization. So you could create the array, and then thousands of lines of code later initialise its values.

Also notice that in the second piece of code we are once again using nested for-loops to walk through the array - this time in order to initialise every array element individually. Whenever you are working with 2D arrays, you will most likely be using the same technique to access every element in turn.

The C Standard requires that any partially-complete array initialisation is to be padded out with zero for the remaining elements (by the compiler). This goes for all data types. For example, consider the creation of an array of characters:

```c
char buf[10] = "";
```

is the same as:

```c
char buf[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

```c
char buf[10] = " ";
```

is the same as:

```c
char buf[10] = {' ', 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

```c
char buf[10] = "a";
```

is the same thing as:

```c
char buf[10] = {'a', 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

The same rules hold for multi-dimensional arrays.

It is a good idea whenever you are working with arrays in C to print out the contents of the array once it's been created - just to make sure that it contains what you expect. A large amount of bugs people encounter in C happen because of some oversight or incorrect allocation of memory when creating data, and this is especially true of arrays where we are potentially working with much larger chunks of memory.

Malloc and Free

There are times when we don't know exactly how much memory will be needed at compile time, so we have to get memory during the program's execution. For example in all previous examples of creating arrays, we have explicitly defined how big the arrays are when creating them:

```
int myArray[256][256] = { { 0 } };
```

```
char buf[10]
```

```
int myArray[3][4] = {1,2,3,4,5,6,7,8,9,10,11};
```

What happens if we don't know how big these arrays will be? For example what if we are asking the user how big an array they would like, or we want to change the size of the array while the program is running?

It is possible to initialize pointers (and therefore create arrays) using free memory. This allows for dynamic allocation of memory.

The function **malloc**, residing in the **stdlib.h** header file, is used to initialize pointers with memory from free store (a section of memory available to all programs). malloc works just like any other function call. The argument to malloc is the amount of memory requested (in bytes), and malloc gets a block of memory of that size and then returns a pointer to the new block of memory allocated:

```
//Dynamically create an integer array of 2000 elements

int *a;
int size = 2000
a = (int *) malloc(size * sizeof(int));
```

Here we created a pointer called a, created an int called size of size 2000 (values) * 4 (bytes), and then asked the operating system to allocate 8000 bytes of memory for us. We can then go ahead and do stuff with this memory, for example as an array:

```
//set all the memory locations to hold the value 5
for(i=0; i<size; i++)
    a[i] = 5;

//print out all the array values
for(i=0; i<size; i++)
    printf("\ni is : %d", a[i]);

//later
free(a);
```

One thing to remember when allocating memory using malloc is to always remember to free the memory after your program has finished using it using free(). If you don't do this, the memory will be reserved on the heap and the operating system won't be able to reallocate it, leading to a build up of non-reusable memory - known as a memory leak.

Exercise 1 - Passing by Reference

Take a look at the following piece of code:

```
#include <stdio.h>

void square(int num) {
    num = num * num;
}

int main() {
    int x = 4;
    square(x);
    printf("%d\n", x);

    system("PAUSE");
    return 0;
}
```

This program is supposed to be calculating 4^2 and then printing out the result. However when we run the program it prints out the number 4 - and not 16 like we expect. Why do you think this is?

By default, function arguments in C / C++ are **passed by value**. When arguments are passed by value, a copy of the argument is passed to the function, rather than the variable itself. Therefore, when we pass arguments into a function in this way, the only way to return the value back to the caller is via the function's return value.

In most cases, pass by value is the best way to pass arguments to functions — it is flexible and safe. It has two main advantages:

- Arguments passed by value can be variables (e.g. x here), literals (the number 8) or expressions (x+1)
- Arguments are not changed in the function, which prevents side effects when we return to the caller

However, there are cases where it's useful to **pass by reference**, **where we pass a pointer to a variable into a function** (and therefore its memory address) - thereby allowing that function to directly modify the variable's value. This can be useful for a number of circumstances - for example copying large data structures into functions can cost a lot of time and performance, especially if the function is called many times! It's easier just to modify the contents directly.

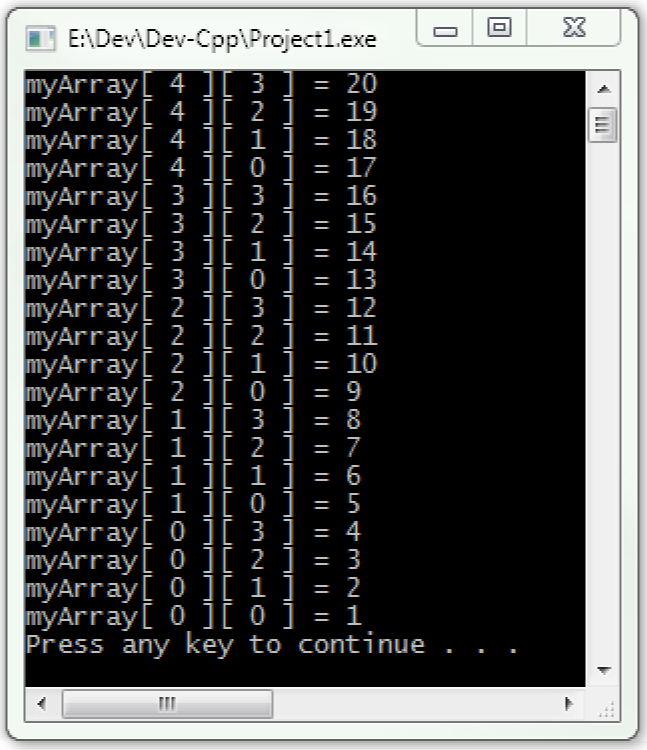
Your task for this exercise is to modify the code so that it passes by reference.

Hints

- In C, to pass by reference you need to declare the function parameter to be a pointer
- All you will need to add are & and * symbols
- Take it line by line - think in terms of computer memory about what you are doing when you use & (address of) and * (value at address). Check last week's lab / lecture notes if stuck.
- Ask a demonstrator in the lab if you are very stuck!

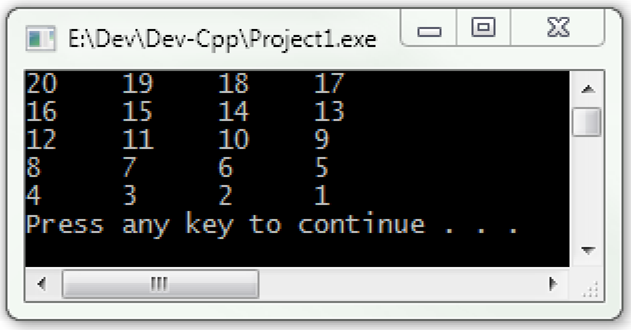
Exercise 2 - Columns and Rows

Create a 2D array (5 x 4) that stores the numbers from 1-20. Print out the contents of the array backwards, so that the output looks like the following:



```
myArray[ 4 ][ 3 ] = 20
myArray[ 4 ][ 2 ] = 19
myArray[ 4 ][ 1 ] = 18
myArray[ 4 ][ 0 ] = 17
myArray[ 3 ][ 3 ] = 16
myArray[ 3 ][ 2 ] = 15
myArray[ 3 ][ 1 ] = 14
myArray[ 3 ][ 0 ] = 13
myArray[ 2 ][ 3 ] = 12
myArray[ 2 ][ 2 ] = 11
myArray[ 2 ][ 1 ] = 10
myArray[ 2 ][ 0 ] = 9
myArray[ 1 ][ 3 ] = 8
myArray[ 1 ][ 2 ] = 7
myArray[ 1 ][ 1 ] = 6
myArray[ 1 ][ 0 ] = 5
myArray[ 0 ][ 3 ] = 4
myArray[ 0 ][ 2 ] = 3
myArray[ 0 ][ 1 ] = 2
myArray[ 0 ][ 0 ] = 1
Press any key to continue . . .
```

Then, print out the contents of the 2D array by row and column, where a new line is introduced after each row has been printed:



```
20     19     18     17
16     15     14     13
12     11     10     9
8      7      6      5
4      3      2      1
Press any key to continue . . .
```

Hints

- You will need to use curly braces { } to split up the two for-loops as we will want to run different printf statements depending on where we are in the nest
- Use something similar to "%-5d" to left justify text into columns like this

Exercise 3 - Row after row after row

What do you think the output will look like if we run the following snippet of code?

```c
int array[2][3] = { 1, 2, 3, 4, 5, 6 };
int r, c;

for (r = 0; r < 2; r++)
     for (c = 0; c < 3; c++)
          printf ("array[%d, %d] = %d\n", r, c, array[r][c]);
```

Run the code and see if it works like you expected.

Now consider the following alternative display loop:

```c
for (c = 0; c < 3; c++)
        for (r = 0; r < 2; r++)
             printf ("array[%d, %d] = %d\n", r, c, array[r][c]);
```

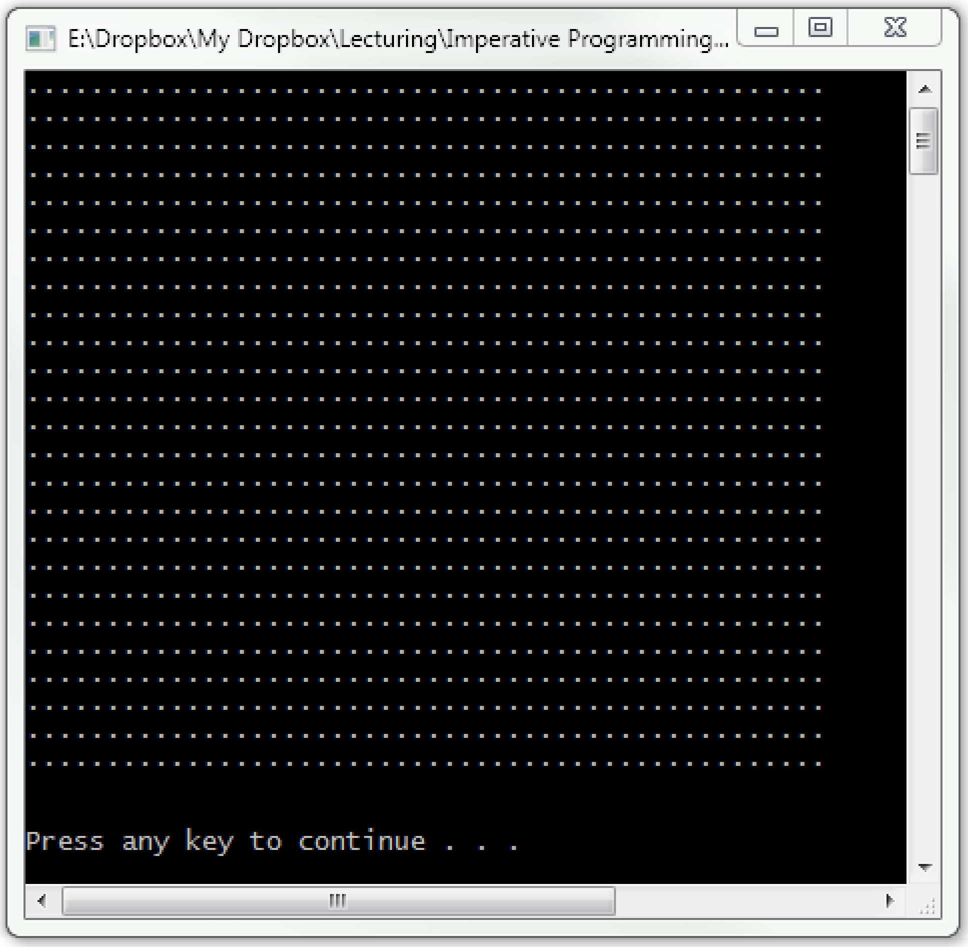
What do you think this code will print out?
Run the code - Can you explain what's going on here?

The answer is that two-dimensional (or any multidimensional) arrays are stored in memory as one dimensional arrays. These are laid out row by row - called the row-major form. If you guessed incorrectly - Look back over the code and run it again with this information in mind.

Exercise 4 - Random Crosses

Create a 2D character array of size 25 * 50 and fill it with '.' characters. You will need to initialise each array element in turn using two for loops, similar to that shown in the introduction.

Ensure that the array is being created and initialized properly by printing out the contents of using the row, column method you developed for exercise 2. You results should be similar to this:



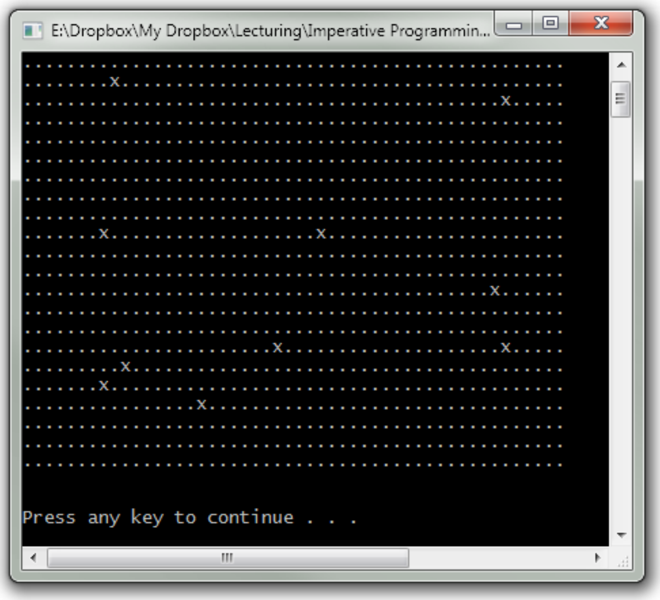
```
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................


Press any key to continue . . .
```

Your task for the first part of this exercise is to generate 'x' characters at random x and y coordinates in this grid.

You should also ensure that no 'x' characters are generated right at the edge of the grid (around the border). You might want take this one step further and define a margin, by which no 'x' characters can appear (2 from edge, 3 from edge etc.).

Test that your program works correctly by randomly generating a large number of x characters (e.g. 10000). You should see very clearly if your program is generating a margin correctly. If your program crashes, the reason is likely because you are stepping outside the boundary of the array. **The key for beginners working with 2D arrays in C is to take things one step at a time, testing thoroughly as you go. You will save yourself a lot of hassle if you spot bugs and potential problems when they first appear!**

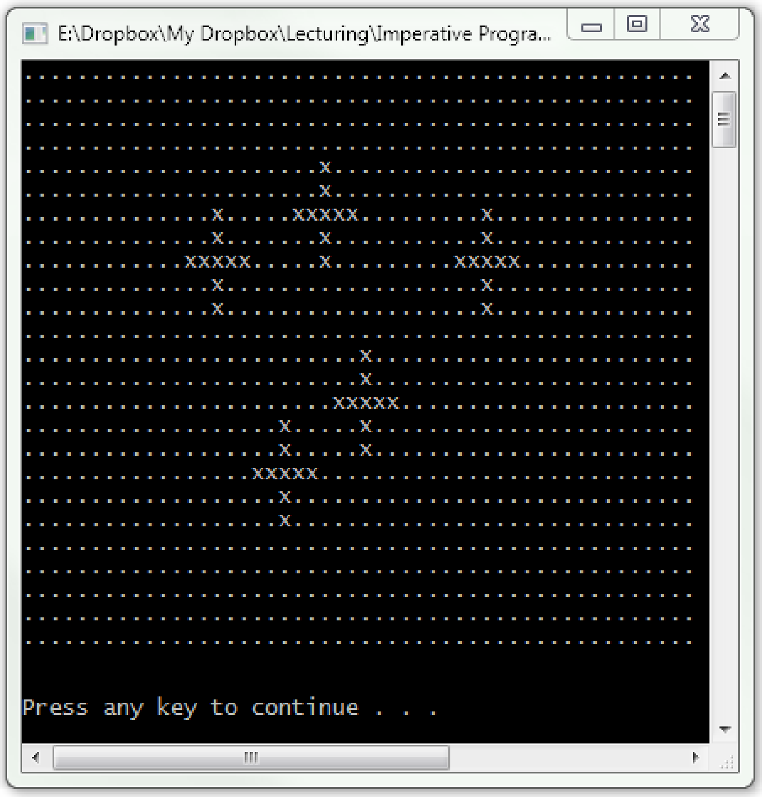
Once complete your output should resemble image on the right.


E:\Dropbox\My Dropbox\Lecturing\Imperative Programmin...

```
..................................................
........x.........................................
............................................x.....
..................................................
..................................................
..................................................
..................................................
..................................................
..................................................
.......x...................x......................
..................................................
..................................................
...........................................x......
..................................................
..................................................
.......................x....................x.....
.........x........................................
.......x..........................................
................x.................................
..................................................
..................................................
..................................................


Press any key to continue . . .
```

Extend your program so that instead of placing a single 'x' character, your program draws a cross pattern at each random location (two 'x' above, below, left and right of the original 'x'). You might find it easier to explicitly store the width and height of your grid at the start of your program (int width = 25; int height = 50) so that your index calculations become generic to any dimensions of grid.

Once finished, your program should produce results similar to this:


E:\Dropbox\My Dropbox\Lecturing\Imperative Progra...

```
..................................................
..................................................
..................................................
..................................................
......................x...........................
......................x...........................
..............x.....xxxxx.........x...............
..............x.......x...........x...............
............xxxxx.....x.........xxxxx.............
..............x...................x...............
..............x...................x...............
..................................................
.........................x........................
.........................x........................
.......................xxxxx......................
...................x.....x........................
...................x.....x........................
.................xxxxx............................
...................x..............................
...................x..............................
..................................................
..................................................
..................................................
..................................................
..................................................


Press any key to continue . . .
```

Passing arrays by value

To pass a 1D array into a function by value, we do something similar to this:

```
float average(float a[]);  //declaration

main()
{
  float myArray[]={12, 55, 22.6, 31.5, 20.5, 56};
   float avg = average(myArray);
   printf("Average : %.2f", avg);
}


float average(float a[]){
     int i;
     float avg, sum=0.0;

     for(i=0;i<6;++i)
       sum+=a[i];

     avg =(sum/6);
     return avg;
}
```

In the code above we are creating an array containing 6 floats, then passing the array into the average function **by value**. This will create a copy of myArray for use inside the average function.

Similarly, for 2D arrays we pass by value like this:

```
void printArray(int c[3][3]); //declaration

main()
{
    int myArray[3][3] = {1,2,3,4,5,6,7,8,9};
    printArray(myArray);
}



void printArray(int c[3][3])
{
int i,j;
for(i=0;i<3;++i)
      for(j=0;j<3;++j)
          printf("%d\n", c[i][j]);
}
```

Here the main() function is passing myArray **by value** into the printArray function. The function will print out the contents of **a copy** of the myArray array.

Passing arrays by reference

There are times where we want to pass a reference to an array into another function - recall from earlier that a disadvantage of passing by value occurs when we are passing large amounts of data, as we need to copy all the information into the function - which is commonly the case when we are working with arrays.

It is said that arrays 'decay' into pointers. An array declared as numbers[5] cannot be re-pointed - i.e. you can't set it to a different memory location. When we pass arrays into a function, either directly or using a pointer to that array, it has decayed functionality - for example we lose the ability to call sizeof() on the array, because it has essentially become a pointer inside the function. Therefore we need to actually pass the size of the array into the function along with the array.

In C, arrays are passed as a pointer to the first element. They are the only element that is not really passed by value (the pointer is passed by value, but the array is not copied). This allows us to modify the values of the array directly inside the function:

```
#include <stdio.h>

void reset( int *array, int size)
{
   int i=0;
   for(i=0; i<size; i++)
        array[i] = 0;
}


int main()
{
    int i;
    int myArray[10] = {0,1,2,3,4,5,6,7,8,9};

    for(i=0; i<10; i++)
        printf("%d ", myArray[i]);

    reset( myArray, 10 ); // sets all elements to 0

    printf("\n--------------------\n");

    for(i=0; i<10; i++)
        printf("%d ", myArray[i]);


    printf("\n\n\n");
    system("PAUSE");

    return 0;
}
```

For the function reset above, we are passing the array as a reference to the function - notice that the parameter is *array, signifying a pointer type. Also notice we don't need to include the & symbol when passing in main (like we had to do in Exercise 1) - this is because arrays work similarly to pointers - i.e. they already hold the address of the first element.

As we are passing by reference, the reset function is working directly on the original array, so when it sets each element to 0, it is resetting the values of the actual array, rather than working from a copy.

Malloc and 2D arrays

Recall that we can use the **malloc** function to dynamically allocate memory at runtime. We previously looked at how you can use malloc to allocate a 1D array. How does this extend to multi-dimensional arrays?

```
int num_rows = 10;
int num_cols = 5;
int *array = malloc(sizeof(int) * num_rows * num_cols);

//later...
free(array)  //remember to free when finished!
```

This code simply allocates enough memory to store num_rows * num_cols integer values - which is all the memory we need to store a 2D array of this size.

We can then access any element within the array like this:

```
array[(num_rows * col) + row] = 101;
```

So for example if we want to change column 0, row 0 it would look like this:

```
array[(num_rows * 0) + 0] = value;   //[0,0] = 101
```

and column 2, row 3:

```
array[(num_rows * 2) + 3] = value;   //[3,2] = 101
```

Essentially what we are doing here is jumping forward row number of rows, and then getting the appropriate column using the row-major form. In a 2D grid, this would be the equivalent of going:

$$grid[(x * width) + y]$$

So, to initialise every item in the array to the value 101, we could write:

```
int x,y;
for(x=0; x<rows; x++)
    for(y=0; y<cols; y++)
        array[(x * cols) + y] = 101;
```

Notice that in the previous example we have lost the ability to refer to an array like this:

array[x][y]

The reason being we were actually creating a 1D array, and then treating it like a 2D one. This is perfectly valid, but if you'd prefer to work with real 2D arrays then we need to do some modifications into how we allocate the memory:

```c
int **array;
int i, j;
int rows = 10;
int cols = 5;

//allocate the rows
array = (int **)malloc(rows * sizeof(int *));

//allocate the columns
for (i = 0; i < rows; i++)
{
    array[i] = (int *)malloc(cols * sizeof(int));
}

//set everything to 101
for (j=0;j<rows;j++)
{
    for (i=0;i<cols;i++)
    {
        array[i][j] = 101;
    }
}
```

We start with a pointer to a pointer called array (indicated by the double asterisk **). We then use malloc to allocate memory for a row number of arrays, and then we use a for-loop to move through these arrays and allocate memory to store all the columns. We end by setting all the values to 101.

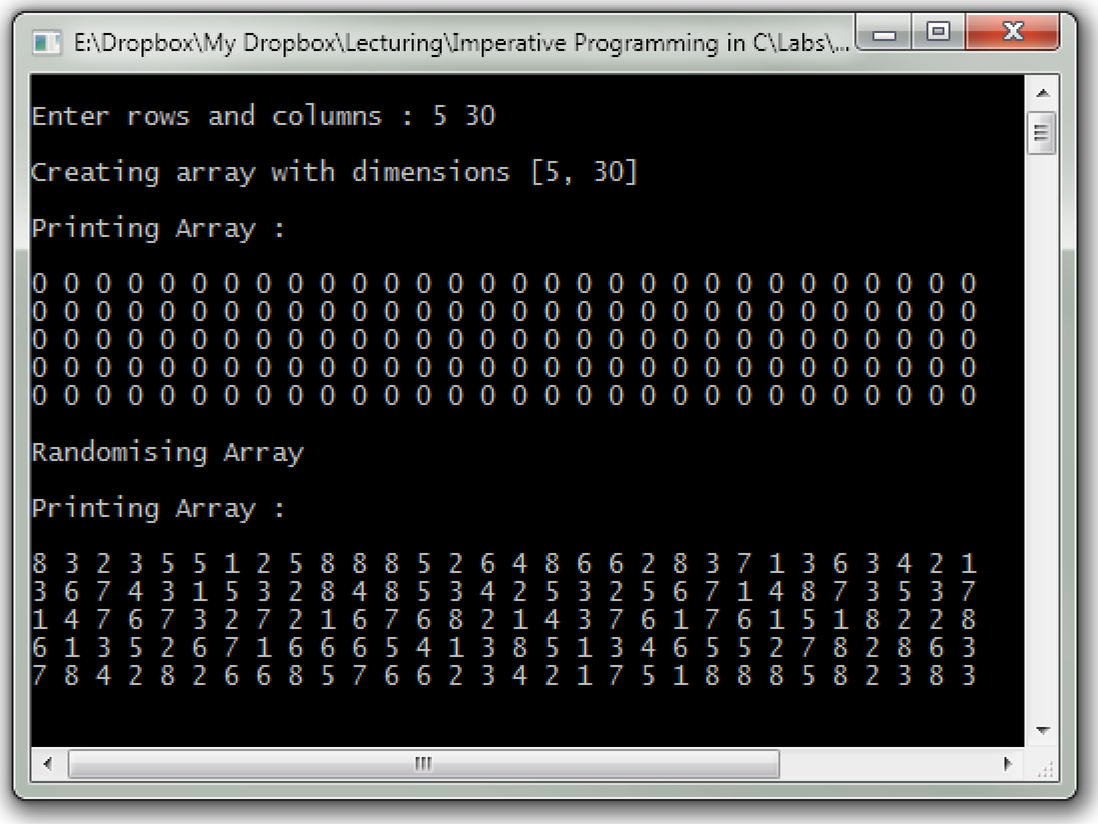
It is essentially up to you which version you prefer here - both techniques use malloc to declare a 2D array and then allocate enough memory to store rows * cols number of integers.

Exercise 5 - Create, Modify, Print, Free

You can think of this exercise as being practice for your assignment.

Your task for this last exercise is to write three functions - one to create and return a dynamic 2D array, one to print the contents of a 2D array, and another to add random numbers [1-9] into an array. The function declarations should look like this:

```
/* dynamically creates an array of size rows * cols, returns a pointer to the array */
int* createArray(int rows, int cols);

/* takes array and dimensions, prints out the contents formatted by row and column */
void printArray(int* array, int rows, int cols);

/* takes array and dimensions, randomly allocates each element to 1-9 */
void randomiseArray(int *array, int rows, int cols);
```


E:\Dropbox\My Dropbox\Lecturing\Imperative Programming in C\Labs\...

```
Enter rows and columns : 5 30

Creating array with dimensions [5, 30]

Printing Array :

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Randomising Array

Printing Array :

8 3 2 3 5 5 1 2 5 8 8 8 5 2 6 4 8 6 6 2 8 3 7 1 3 6 3 4 2 1
3 6 7 4 3 1 5 3 2 8 4 8 5 3 4 2 5 3 2 5 6 7 1 4 8 7 3 5 3 7
1 4 7 6 7 3 2 7 2 1 6 7 6 8 2 1 4 3 7 6 1 7 6 1 5 1 8 2 2 8
6 1 3 5 2 6 7 1 6 6 6 5 4 1 3 8 5 1 3 4 6 5 5 2 7 8 2 8 6 3
7 8 4 2 8 2 6 6 8 5 7 6 6 2 3 4 2 1 7 5 1 8 8 8 5 8 2 3 8 3
```

Here is a skeleton program to get you started. All you have to do is fill out the functions.

```
/* Function declarations */

int* createArray(int rows, int cols);
void printArray(int *array, int rows, int cols);
void randomiseArray(int *array, int rows, int cols);

/* Exercise 5 function */

exercise5()
{
   int rows, cols;

   //Get dimensions from user

   printf("\nEnter rows and columns : ");
   scanf("%d %d", &rows, &cols);

   //Create array

   int *myArray = createArray(rows, cols);

   //print, modify, print array

   printArray(myArray, rows, cols);
   randomiseArray(myArray, rows, cols);
   printArray(myArray, rows, cols);

   //free up space

   free(myArray);
}


/* dynamically creates an array of size rows * cols, returns a pointer to the array */

int* createArray(int rows, int cols)
{
   int *temp;
   return temp;
}


/* takes array and dimensions, prints out the contents formatted by row and column */

void printArray(int* array, int rows, int cols)
{

}


/* takes array and dimensions, randomly allocates 1-9  to each element */

void randomiseArray(int *array, int rows, int cols)
{

}
```

Appendix

Code Snippets – Updated with example C code

Online C Programming Resources

Complete C Reference Library

C Programming IDE’s

Dev-C++ (Windows)

Code::Blocks (Windows, Mac, Linux)

Visual Studio/C++ Express (Windows)

Netbeans C/C++ (Windows, Mac, Linux)

Codelite (Windows, Mac, Linux)