Laboratory 5

Arrays and Strings

5.1 Overview

This laboratory introduces you to *arrays* which are a way to create a variable that has the capacity to store many values. Each of the values in the array can be accessed by using an index.

You will learn how to create and use arrays, and how to make sure that they are initialised with specific values when your program starts. You will also learn about *multidimensional arrays* which can be used to store data in a similar way to tables.

You have already been using text enclosed in quotation marks (`"..."`). In this laboratory you will be learn more about these *text* or *character strings*, which are actually a special form of array. You will learn how to create and manipulate them.

In the graphics option you will make use of arrays to make your graphics program more like a computer game. In the music option you will find that arrays are very useful for defining musical relationships, like the pitch relationships between notes in a scale.

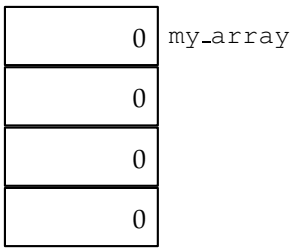
5.2 Introducing Arrays

The variables you have been working with so far are a bit like scalar values in maths. This laboratory is about *arrays* which are more like mathematical vectors.

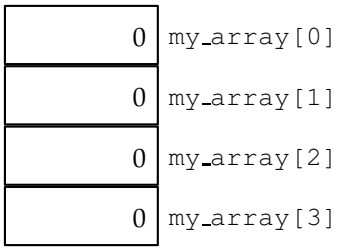
Each variable you declare defines a space in the computer's memory in which you can store one value of the correct type. We often show this as a diagram like this:

The variable declaration on the left is equivalent to the diagram on the right. In the diagram, the box represents the space in the computer's memory in which the value of the variable is stored. In this case you can see that the box contains the value 20 to indicate that the variable is assigned the value 20. The text to the right of the box (`my_integer`) identifies the name of the variable: it provides a name for that space in the computer's memory.

Every variable you declare has a space in memory, and the name of that variable identifies the memory space. An array allows you to reserve *more than one* memory space for a single variable. All the memory spaces you reserve will have the same type (for example `int`). When you use the variable you have to specify which one of the spaces you wish to access. Diagrammatically, an array looks like this:



This shows that one variable name identifies multiple spaces in the computer's memory. These spaces are more properly called *elements*. To identify which element we are talking about we reference it by number. In C **elements are numbered from zero**. This 'element number' is called a *subscript* or an *index*. A subscript is placed after the variable name, in square brackets, like this: `my_array[3]`. We can show the way that subscripts work on a diagram:



Obviously the subscript must be an integer; it wouldn't make sense to ask for element number 2.3859.

You declare an array by specifying how many elements you want in that array. For example:

```
int my_array[4];
```

declares a new array variable called `my_array`. Each element in the array contains an integer value. The array has four elements in it. Note that the last element is called `my_array[3]`. Try to remember that when you are declaring an array, the number in the square brackets is the number of elements, **not** the subscript of the last element. The subscript of the last element will be the size of the array minus one.

Syntax: Array Declarations

type name`[`*size*`]` [, more variable delcarations]^{n} ;

e.g.

```
int an_integer_array[10];
double a_real_array[6];
int an_int_variable, an_int_array[22];
```

Arrays are variables which have space for more than one value. The number of values that the array has space for is specified as a number in square brackets (*size*).

When the array variable is used, an integer must be included specifying the index of the element which is being referenced. This index is called a *subscript*. Subscripts are numbered **starting from zero**. For example:

```
an_integer_array[6]
```

is the integer value which occupies the 7th element in the the array called `an_integer_array`.

5.3 Working with Arrays

You are now going to try using arrays in a program. The "lab5" program uses arrays to collect information. The program asks the user for the heights of up to ten people and then displays the mean height, calculated from the data that were entered. The heights that the user enters are stored in an array. It would be very difficult to produce a program like this without using arrays.

Open "clab" project and set the target to "lab5". Try building and executing the program. Run it a few times to make sure you are familiar with what it does. The source code for "lab5" is shown below.

```
/*
 *  A program to demonstrate arrays
 *  C Programming laboratory 5
 */

#include <stdio.h>

/* Function prototypes */
void welcome_message(void);
double mean(double values[], int num_values);

/*
 * Program starts here
 */
int main(void)
{
    /* Declare an array which can store up to 10 doubles */
    double data_values[10];

    /* Keep track of the number of items we have here */
    int num_data_values;

    /* Some other useful variables */
    int curr_data_value, done;
    double mean_height;

    /* Set initial conditions */
    num_data_values = 0;
    done = 0;

    /* Display message to user */
    welcome_message();

    /* Loop to get all data values, stop when we have 10 values */
    /* Or when the user enters a zero height */
    for (curr_data_value = 0;
            (curr_data_value < 10) && (done == 0);
            curr_data_value++)
    {
        /* Get the data into the array */
        printf("Please enter the height for person %d (or 0 if done): ",
            curr_data_value + 1);
        scanf("%lf", &data_values[curr_data_value]);

        /* If the data was non-zero we have another height */
        /* If not, we are done */
        if (data_values[curr_data_value] != 0)
            num_data_values++;
        else
```

```
            done = 1;
    }

    /* Calculate the mean height */
    mean_height = mean(data_values, num_data_values);

    /* Display it */
    printf("\nThe mean height is %lf\n", mean_height);

    return 0;
}

/*
 * Displays welcome message
 */
void welcome_message(void)
{
    printf("Mean Height Calculator\n");
    printf("Enter the height of each person when prompted\n");
    printf("Enter 0 when you are done. You can enter up to 10 values\n\n");
}

/*
 * Calculates the mean of a set of values
 */
double mean(double values[], int num_values)
{
    double mean;
    int current_value;

    /* Initialise the mean to zero */
    mean = 0;

    /* Calculate the sum of all of the values */
    for (current_value = 0; current_value < num_values; current_value++)
    {
        mean += values[current_value];
    }

    /* Divide by the number of values, if there are any */
    if (num_values > 0)
        mean /= num_values;

    return mean;
}
```

The program uses the `main` function to collect information from the user. It defines two other functions: the `welcome_message` function displays useful text to the user when the program starts; the `mean` function calculates the mean of the values that the user has entered.

The program will accept up to 10 data values, so it declares an array of 10 `double` elements, like this:

```
    double data_values[10];
```

The program allows the user to enter less than ten values if they want, so we need to keep track of how many values they *have* entered. This is done using another variable:

```
    int num_data_values;
```

After the welcome message has been displayed the program enters a `for` loop. This loop counts through the elements in the `data_values` array. Each time the loop runs (each *iteration*) the user is asked for a height. The `scanf` function call places the result into the correct element in the array:

```
scanf("%lf", &data_values[curr_data_value]);
```

This `scanf` call is just like the ones you have seen in previous labs, except that instead of a scalar variable this call puts the result into an element of the array. The element it puts the result in is decided by the value of the `curr_data_value` variable. The value of the `curr_data_value` variable is set by the `for` loop. The `for` loop will stop when it gets to the tenth element in the array (`data_values[9]`). It will also stop if the `done` variable is assigned a value other than zero. This is controlled by the condition part of the `for` loop.

To calculate the mean, the array is passed to another function, called `mean`. Note how the parameter which receives the array is declared. The prototype for the `mean` function looks like this:

```
double mean(double values[], int num_values);
```

The array is declared with empty square brackets because we don't need to limit this function to only working with arrays of a certain size. In this case, the function just needs to know how many values out of the array to use. Even if the program always used the whole array, we would still need to pass the size of the array as a parameter to the function. **There is no way to determine the size of an array in C**. You must store it in a variable somewhere, or you must fix it at the time when you write the program.

Exercise 5.1: Understanding the Array Program

Make sure you understand how the "lab5" program works. Try stepping through some or all of it if you are unsure. If you only want to step through some of it (perhaps you only want to step through the `mean` function) then use a breakpoint to interrupt the program when it reaches the part you are interested in.

If you do not understand anything about the program, ask one of the demonstrators for help.

When you are sure that you understand how the program works, change the source code to allow the program to accept up to 15 values, rather than 10. Rebuild the program and test it to make sure that your changes have worked.

Exercise 5.2: Add a Standard Deviation Function

The standard deviation of a sample of values measures how 'spread out' the values are and may be calculated using the following formula:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2 - (\bar{x})^2} \tag{5.1}$$

Where:

- σ is the standard deviation;
- x_i represents the ith data value;
- $\bar{x}$ is the mean of all of the data values;
- N is the number of data values.

The equation may also be written as the difference between the mean of the squared data values and square of the mean:

$$\sigma = \sqrt{(\bar{x^2}) - (\bar{x})^2} \tag{5.2}$$

Add a function to "lab5" called `standard_deviation` which takes the array of data values and the number of data values as arguments. It should return the standard deviation. You should add lines to the `main` function to call your new function and display the standard deviation on the screen.

Hints:

- Use equation 5.2 to calculate the standard deviation.
- Create a second array of the same maximum size as the `data_values` array (i.e. 15 elements).
- You should copy the original data items into this new array, but square them.
- Use the `mean` function to calculate the two means.
- You might want to look back at the last laboratory to remind yourself about the `pow` and `sqrt` functions.
- You will need to include `math.h`.

5.4 Initialising Arrays and Multi-Dimensional Arrays

Sometimes you will want to make sure there are specific values in elements of an array before you do anything with it. Imagine a situation where you want to know the number of days in a month. If you have the number of the month (from 0 to 11) in a variable called `month`, you could use an array to find out the number of days the month has:

```
int days_in_month[12];
...
printf("The number of days in January = %d", days_in_month[0]);
```

For this to work you would need to make sure the array `days_in_month` was set up with the right values. We could do that like this:

```
int days_in_month[12];
```

```
days_in_month[0] = 31;
days_in_month[1] = 28;
days_in_month[2] = 31;
days_in_month[3] = 30;
days_in_month[4] = 31;
days_in_month[5] = 30;
days_in_month[6] = 31;
days_in_month[7] = 31;
days_in_month[8] = 30;
days_in_month[9] = 31;
days_in_month[10] = 30;
days_in_month[11] = 31;
```

But this is a pain. Fortunately, C provides us with a way to initialise the value of an array when we declare it. Like this:

```
int days_in_month[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

The values on the right-hand side of the assignment operator (=), in the curly brackets, are known as *initialisers*. There must not be more initialisers that elements in the array.

The next program you will write will use initialisers to set numbers in an array. The difference from the arrays we have just been looking at, is that this next program will need a *multidimensional array*. The arrays you have seen so far have a single dimension; they work like a table with a single column. The subscript is like the row number, telling the program where to look in the table. A two dimensional array has two subscripts; it works like a table with multiple columns. The two subscripts act as the row and column numbers, selecting a single cell in the table. Three and more dimensions are also possible.

A two dimensional array is declared in C like this:

```
int data_table[5][3];
```

You could think of the `data_table` array as being a table with 5 rows and 3 columns. You would access the `data_table` array like this:

```
printf("The first element in the table is %d", data_table[0][0]);
```

Notice that you need to specify both subscripts.

Initialising a two dimensional array is like treating each row as a one-dimensional array, and then collecting all of the rows together. Look at this example:

```
int data_table[5][3] = { {1, 2, 3}, {2, 3, 4}, {3, 4, 5}, {4, 5, 6},
                         {5, 6, 7} };
```

If we treat the first subscript as the column index, and the second as the row index, `data_table` is equivalent to the following table of values:

1	2	3
2	3	4
3	4	5
4	5	6
5	6	7

Change the target to "lab5a" and open "lab5a.c" in the editor. "lab5a" is intended to be a program which informs the user of the distance between a small selection of major world cities. However, at the moment the program isn't finished. The best way for the program to work out the distance between cities is to look it up in a table (i.e. a two-dimensional array). You will need to create and initialise a suitable array to get the program to work. You should initialise it with data from the table below. The distances are in miles.

	Cape Town	Hong Kong	London	New York	Rio de Janeiro	Tokyo
Cape Town	0	7,375	6,012	7,764	3,773	9,156
Hong Kong	7,375	0	5,982	8,054	11,021	1,794
London	6,012	5,982	0	3,458	5,766	5,940
New York	7,764	8,054	3,458	0	4,817	6,740
Rio de Janeiro	3,773	11,021	5,766	4,817	0	11,533
Tokyo	9,156	1,794	5,940	6,740	11,533	0

Exercise 5.3: Using an Initialised Two-Dimensional Array

Add a two dimensional array to the "lab5a" program and initialise it with the data shown in the table. You can then use the indices that the user enters to look up the correct distance.

Some things to look out for:

- Remember that array subscripts are indexed from zero.
- Using a subscript that is beyond the range of the array can cause big problems. Validate the data that the user enters to make sure it is valid.

If you don't understand how to do this, ask for help.

5.5 Strings: A Special Kind of Array

You have already used pieces of text many times in your C programs. They have usually been in `printf` statements and have been enclosed in double quotation marks `"like this"`. A piece of text like this is called a *string* because it is treated by the programming language as a *string of characters*. The name *string* is just an abbreviation.

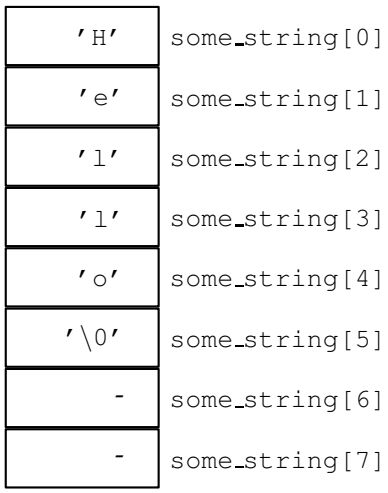
In C, strings are treated as arrays of type `char`. Each element in the array holds a single character. For example, we could declare an array to hold strings like this:

```
char some_string[8];
```

Initialising a character array like this is quite easy, for example:

```
char some_string[8] = { "Hello" };
```

When a character array is initialised like this each letter is stored in an element of the character array `some_string`. Like this:



You will notice that the element immediately after the last character in the string (`some_string[5]` in this case) contains a the character `'\0'` (that's a backslash character and a zero character, not the letter `'o'`). This is an *escape sequence* just like the special sequence for a new line (`'\n'`). `'\0'` is the escape sequence for the *null* character. It is used to signal the end of the string. A string that ends in a null character is said to be *null terminated*. All strings in C are null terminated. When you initialise a string the null termination is added to the array for you automatically. Because the null termination takes up an extra character, you **must** make sure that any character array you declare has one more space in it than the maximum number of characters that you want it to hold. For example, if you wanted to declare an array for holding strings of up to 20 characters, you should declare it as:

```
char another_string[21];
```

to make sure that there is room for the null termination character.

You can use the null termination to find the end of a string in an array. For example, you could check the first element in the array to detect whether a string is empty (i.e. it has no characters, like this: `""`). You would check for the null termination like this:

```
if (another_string[0] == '\0')
{
    /* The string is empty */
}
else
{
    /* The string is not empty */
}
```

The example program "lab5b" contains an example of how to do some basic string handling. Change the target to "lab5b" and open "lab5b.c" in the editor. Try building and executing the program. What does it do?

The source code for "lab5b.c" is shown below.

```
/*
*  A program to demonstrate strings
*  C Programming laboratory 5
*/

#include <stdio.h>

/*
* Program starts here
*/
```

```
int main(void)
{
    /* Declare a character array to store the string in */
    char string_data[20];

    /* Ask the user for a string */
    printf("Enter a word: ");
    scanf("%19s", string_data);

    /* Show them the string */
    printf("The word you entered is: %s\n", string_data);

    /* Tell them what the first character was */
    printf("The first character of the word is %c\n", string_data[0]);

    return 0;
}
```

You should see that the program is quite simple; it uses a `scanf` statement to read in the string. It then displays the string using `printf` and then displays just the first character using another `printf` statement.

The `scanf` and `printf` statements use a placeholder that you haven't seen before for handling strings: `%s`. For example, the `printf` statement uses the `%s` placeholder to substitute the characters out of the string variable and into the string which is going to be displayed.

You will notice that the `scanf` statement also uses a `%s` placeholder, to read in a string. However, in the case of `scanf` a number appears between the percentage sign (`%`) and the `s` character. The number is the maximum number of characters to read into the string. It ensures that the program does not try and put too many characters into the array.

When using an array with `scanf` you **do not** need to use the ampersand character (`&`) before the variable name. The reason for this is a little complicated and you will study it in more detail in laboratory 8.

Function Reference: `%s` — Using the `%s` placeholder with `scanf` and `printf`

```
scanf("%lengths", string_variable);
printf("%s", string_variable);
```

The `%s` placeholder allows `scanf` and `printf` to handle strings. When using `scanf` it is good programming practice to make sure the length of the string that can be read is limited using a number (*length*) between the percentage sign and the `s` character. This number should be one less than the length of the array to allow room for the null termination character.

As the argument which matches the `%s` placeholder is an array. It does not need to be preceded by an ampersand.

Exercise 5.4: Understanding Character Arrays

Test the "lab5b" program with different input. You should find that `scanf` prevents you from being able to enter nothing at all. It also only ever uses the first word that you enter.

- Why is the `scanf` placeholder `%19s`? (Why is it not `%20s`?)
- What happens if you enter a word longer than 19 characters?
- What happens if you change the placeholder to have a number less than 19?

Exercise 5.5: Handling Null Terminated Strings

Add to "lab5b" to create a function with the prototype:

```
int string_length(char string[]);
```

The function should count the number of characters in the string. It should stop counting when it reaches the null character. It should be able to handle situations where there are no characters in the string (i.e. the first character is the null character). Call your new `string_length` function from within the `main`function so that you can display the length of the word that the user enters.

If you don't understand how to do this, ask for help.

Exercise 5.6: Character Array Handling

Add to "lab5b" to add some code into the `main` function which copies the characters from the `string_data` array into another array, but in reverse order. You will need to use the length of the string (which you determined in your `string_length` function) to do this. Display the reversed string using `printf`. You must make sure that the new string you create (the reversed copy) has a null termination character in the right place.

If you don't understand how to do this, ask for help.

5.6 Forming Strings Using `sprintf`

Sometimes you want to form a more complicated string which will be stored in a character array, rather than displayed on the screen. The `sprintf` function works exactly like `printf` except that, rather than displaying the string it makes onto the screen, in places the result into a character array which you specify. For example, look at the following code snippet:

```
char number_string[10];
int an_integer;

an_integer = 20;
sprintf(number_string, "%d", an_integer);
```

You can see that the `sprintf` function call looks very like a `printf` function call, except that there is a new first argument. The first argument to `sprintf` is the string variable that results will be placed into. After the `sprintf` statement executes, the string `number_string` contains the characters `'2'`, `'0'` and `'\0'`

Function Reference: `sprintf` — Forming strings using `printf`-like placeholders

```
int sprintf(char string_variable[], char format_string[], ...);
```

e.g.

```
sprintf(number_string, "%d", int_variable);
sprintf(answer_string, "The answer is %d", integer_answer);
num_chars = sprintf(display_string, "%lf", double_variable);
```

The `sprintf` function works like the `printf` function except that, instead of outputting its results onto the screen, it places them into the character array `string_variable`. It is called in exactly the same way as `printf` but with an extra first argument (`string_variable`).

`sprintf` takes at least two arguments but, like `printf`, it can take more, depending on how many variables you want to include.

- `string_variable` is the character array which will receive the output of the `sprintf` function as a string;
- `format_string` is a string containing placeholders, just like with `printf`;
- after the first two arguments (where the `...` is) you should include variables whose values will be substituted for the placeholders in the `format_string`.

The placeholders you will use most often are:

- `%d` for integer (`int`) variables;
- `%lf` `double` variables;
- `%c` for single characters (`char`);
- `%s` for strings (`char[]`).

These are the same as for `printf`.

The return value of `sprintf` is an integer, which specifies how many characters were written into the `string_variable` array (not including the null termination).

The `sprintf` function is defined in `stdio.h`

Exercise 5.7: Using `sprintf`

Add to the `main` function in "lab5b" to prompt the user for an integer number. Use the `sprintf` function to create a string version of the integer in a character array (you could re-use one of the arrays you already have, if it is big enough).

Determine the length of the string it produces in two ways:

- use your `string_length` function;
- use the return value from the `sprintf` function call.

Display both answers and make sure that they are the same!

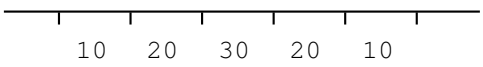
If you don't understand how to do this, ask one of the demonstrators for help.

There is a library that is part of the C standard library which contains a number of functions especially designed for handling strings. All the functions are defined in `string.h`. If you want to find out more about these functions you should look in a book on C, or consult documentation on the internet.

5.7 Graphics Option

Change the target to "graphics1" program and open "graphics1.c" in the editor. You will be adding even more functionality to your program now. By the end of this session it should reach the stage where you can play it as a simple game.

Your graphics program is beginning to get quite sophisticated. The user now has control over the horizontal position of the stick person, and the launch angle of the projectile. You are now going to add a target for them to throw the object at. The target should be at ground level and have numbers on it identifying the score associated with a particular area. It could look something a bit like this:



How many areas there are, and what the scores are for each area is entirely up to you.

The best way to produce the target is using a loop. Each iteration of the loop should produce one of the small vertical lines which separate the scores. You could store the associated scores in array a bit like this:

```
int scores[5] = {10, 20, 30, 20, 10};
```

The code that you write to produce the target should have a structure a bit like this:

```
int line;
char label[3];

for (line = 0; line < 5; line++)
{
    /* Calculate position of line based on the value
    * of the line variable */

    /* Draw the line in the right place */

    /* Calculate position of label based on the value
    * of the line variable */
```

```
        /* Create a string with the label in it */
        sprintf(label, "%d", scores[line]);

        /* Use outtextxy to display the text */
        outtextxy(x, y, label);
    }

    /* Draw the last line here */
```

This is obviously just a skeleton, you will need to fill in the missing sections with code of your own.

Exercise 5.8: Drawing a Target

Create a function called `draw_target` which uses code based on the skeleton to draw a suitable target on the far right-hand side of the graphics window. Feel free to decide:

- how many areas your target will have;
- what scores the different areas will have;
- how big the different areas of the target will be.

Exercise 5.9: Giving the User a Score

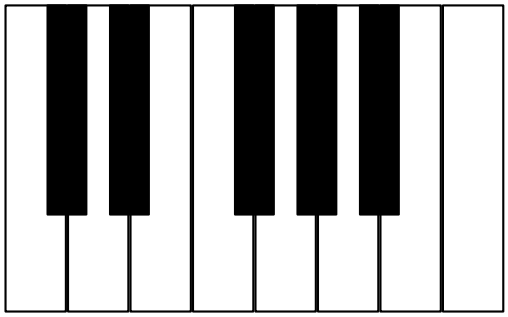
Now the user has a target to throw the object towards, the computer should reward them with a score that corresponds with the position in which the projectile path ends.

Your program calculates the vertical position of the projectile at each horizontal location. It should stop when the vertical location is not above ground level. The horizontal position at which the projectile is no longer above ground level will determine the score.

When the projectile reaches ground level (or below) your program should assign the corresponding horizontal position to a variable. You can then use that variable (which will be an x-coordinate) to calculate where on the target the object fell. You should then be able to use the values in the `scores` array to give the user a score for their throw.

5.8 Music Option

The improvisation program you have been working on has been limited to producing whole tone scales or random notes. It has been difficult to produce other scales as the differences in pitch between notes in say, a major scale, are not easily calculated mathematically. As an example, let's look at a simple C major scale. If the scale starts on middle C the first note of the scale has a pitch value of 60. To get the other notes in the scale we could add small pitch values onto this starting value, like this:



If we chose a major scale in a different key then the pitch offsets would still be the same. But the pitch offsets are not related in any obvious mathematical way.

A way to approach this problem is to store these pitch offsets in an array. For example, you could define a major scale like this:

```
int major_scale[8] = {0, 2, 4, 5, 7, 9, 11, 12};
```

A loop like the following one would play a major scale:

```
int scale_note;
int key;

key = 60;

for (scale_note = 0; scale_note < 8; scale_note++)
{
    midi_note(key + major_scale[scale_note], 1, 64);
    pause(500);
    midi_note(key + major_scale[scale_note], 1, 0);
}
```

Arrays like these are very powerful for representing musical relationships such as scales and chords.

Exercise 5.10: Making Your Improvisation Use Non-Whole Tone Scales

Begin by opening your "music2" program (remember to change the target to "music2"). Add a function which uses as array to play a single ascending octave of a major scale. Try using it in place of the function which produced whole tone scales.

Now is your opportunity to innovate! Try and improve your improvisation program focussing on using scales other than whole tone scales.

- Try some different scales, other than a major scale.
- Try choosing which scale will be played at random.
- How about representing chords using an array? Can you get your program to improvise a chord line to accompany a melody built out of scales?
- Can you use the value of an array to predefine some `pause` values? This would allow you to predefine certain rhythms. Try playing these rhythms on MIDI channel 10 (this is the drum channel).

5.9 Summary

Now that you have finished this laboratory you should understand what arrays are and how they can be used. You should be able to declare both single and multidimensional arrays and initialise them with data.

You should know that strings are character arrays and that all strings in C are null terminated; so they end with the null character (`'\0'`). You should be able to manipulate strings and create them using the `scanf` and `sprintf` functions.

In the graphics option you added to your program to make it much more like a computer game. It should now have a target towards which the user throws the object. The user should then be given a score for their throw.

In the music option you should have found that arrays are very useful for defining sections of music by representing numerical relationships. You should understand how to use an array to represent a scale. You may also have explored how to use arrays for chords and rhythms.