Laboratory 8

Arrays and Strings Revisited and Allocating Memory

8.1 Overview

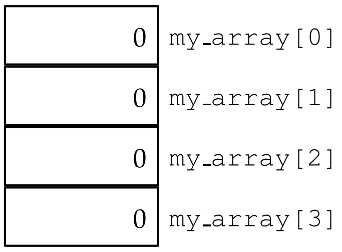
In lab 7 you were introduced to pointers. This lab begins by examining arrays, showing that there is a strong relationship between arrays and pointers. An array is essentially a pointer to a block of memory locations which the C compiler has set aside. Array and pointer notation can therefore be used interchangeably.

A pointer on its own is not very useful, as it has no memory to refer to. This lab introduces the `malloc` function which you can use to allocate blocks of memory whilst your program is running. Any memory you allocate with `malloc` must be de-allocated before your program ends using the `free` function. C provides an operator called `sizeof` which helps you determine how much memory is required by different variable types.

Finally the lab looks at allocating memory for structures, and arrays of structures. You will alter the graphics or music program you have been working on to use the memory allocation functions `malloc` and `free`.

8.2 More About Arrays

Lab 5 introduced you to arrays. You may remember diagrams of arrays with boxes representing the elements of the array. Like this:



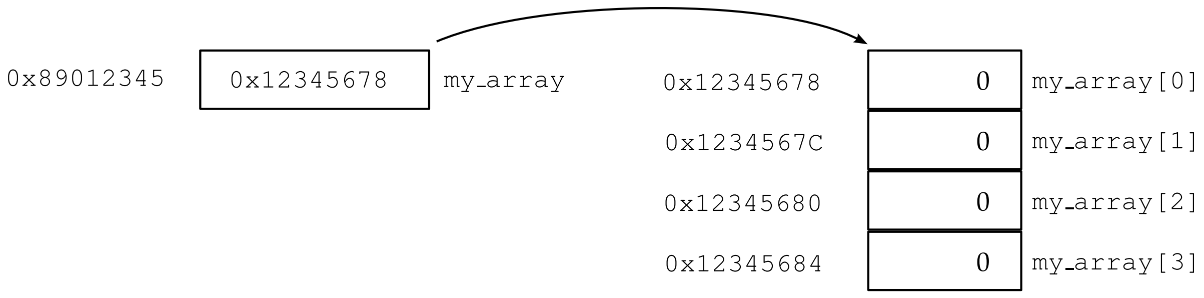
The boxes in array diagrams like this one represent memory locations in just the same as the boxes in the diagrams representing pointer/variable relationships in lab 7. These diagrams simply the way in which arrays work. An array actually has two parts to it:

- A block of memory locations in which the information for each element is stored;
- The array variable is actually a pointer which references the first memory location in the block.

An array that is declared as:

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

Can be drawn diagrammatically as:



The variable name `my_array` is actually a pointer, this is shown on the left-hand side of the diagram. This pointer references the first location of a block of four integer-sized memory locations. The block of memory locations is shown on the right-hand side of the diagram.

When you use the square brackets (`[ ]`) after an array name it is very similar to using the dereference operator (`*`). In fact, writing:

```
my_array[0] = 5;
```

is exactly the same as writing:

```
*my_array = 5;
```

Accessing any element other than the first element in an array requires placing a non-zero number in the square brackets. The pointer equivalent to this is to add a number to the pointer. Like this:

```
*(my_array+3) = 20;
```

Which is exactly the same as writing:

```
my_array[3] = 20;
```

Using mathematical operations, such as addition, on pointers is called *pointer arithmetic* and must be used carefully. Pointer arithmetic changes what the pointer is referencing. There is nothing in C to stop you changing a pointer so that it does not reference a valid memory location. This mistake, often called a *floating pointer*, is very difficult to spot and generally causes your program to crash completely.

In general the array notation, using the square bracketed subscript, is much clearer and easier to read than pointer arithmetic. **Use pointer arithmetic very carefully**, try and avoid it if possible, there are usually clearer ways to write the same thing.

The fact that arrays are just 'pointers with some memory attached' is often reflected in functions which accept arrays as parameters. In lab 5 you used a `mean` function to calculate the mean of an array of values. The prototype for this function was:

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

This prototype could equally have been written using pointer notation, as:

```
double mean(double *values, int num_values);
```

The two prototypes are effectively identical. The body of the `mean` function used the square bracket notation for accessing the `values` parameter as an array. Like this:

```
mean += values[current_value];
```

Even with the function prototype that declares the `values` parameter as a pointer, this line **does not** need to be changed. Just as an array variable is actually a pointer, it is perfectly legitimate to treat a pointer as an array.

If you look at other people's code, including standard C functions, you will find that the pointer notation for array parameters to functions is very common.

Exercise 8.1: Investigating Arrays as Pointers

When you pass an array to a function, you are passing a pointer. You can think of arrays as always being passed by reference. The program "lab8" contains a skeleton for a program which might use an array. Open "lab8.c" in the editor and change the target to "lab8".

Based on the very simple programs you used in laboratory 7, create a program which demonstrates that arrays are effectively always passed by reference. You can make the program as simple or complicated as you like.

8.3 More About Strings

In lab 5 you saw that character strings are also arrays: arrays of type `char`. Each element is a character and the last element in the string is always the null character, `'\0'`. Since string variables are arrays, they are also pointers. You will often find that functions that operate on strings expect a parameter of type `char *` i.e. a character pointer. For example, the prototype for the standard C function `strlen` is as follows:

```
int strlen(char* str);
```

The `strlen` function calculates the length the string it was passed by finding the null-termination character. You wrote a function to do this in lab 5.

You can assign strings to pointers that are specified in your program, like this:

```
char *message;
message = "It's a long way to Tokyo\n";
```

You could then display the message using a `printf` statement, like this:

```
printf(message);
```

You can access individual characters of the message using the array subscript square brackets, for example:

```
message[5]
```

would be equivalent to the character `'a'`. However, you **can not** change the string by writing to it. The string is pre-specified so it may be in memory which you are not allowed to write to. For example, you **should not** do the following:

```
message[5] = '1';
```

If you want to be able to change the message, you must put its contents into an array, like this:

```
char message[] = {"It's a long way to Tokyo\n"};
```

This will allocate writeable memory especially for the message and copy the message into it. You can now change its contents.

Exercise 8.2: Investigating Strings as Pointers

Open the "lab5b" program. This program does some operations on a word that you enter. It should count the length of the word and then reverse the characters in the word and display it. Currently, the reversal operation takes place in the `main` function.

- create a function called `reverse_string`, which reverses the string that you give it;
- for both the `reverse_string` function and the `string_length` function change the prototypes so that the string parameters are declared as pointers, rather than arrays.

So, your prototypes should be:

```
    int string_length(char *string);
```

and:

```
    void reverse_string(char *string);
```

You should find that declaring these parameters as pointers rather than arrays makes no difference to the operation of your program. However, when handling strings it is far more common to use the pointer notation than the array notation.

8.4 Allocating Memory

In the last two sections you found that an array is a pointer to a block of memory locations. The block of memory is allocated by the C compiler when your program is running. The C compiler then sets up the array pointer so that you can use it. There are a number of circumstances under which you might not want the C compiler to allocate the memory for you. You might want to do it yourself. The most common of these reasons are:

- You don't know how big an array you need until the program is running. When you specify an array size, it must be fixed when you write the program. If you allocate the memory yourself you can control how big an array is whilst the program is running, perhaps in response to user input.
- The amount of memory that can be set aside by the compiler for an array is limited. Sometimes you may wish to allocate a large block of memory to use as an array, for example to manipulate a bitmap image or a sound sample. Allocating memory whilst the program is running is more flexible, and the blocks of memory which can be allocated are bigger.

When the compiler allocates array memory for you, you must have specified the size when you wrote the program. This size is fixed, so this is called *static memory allocation*. Allocating memory whilst the program is running is more flexible. This is called *dynamic memory allocation* because the amount of memory that is allocated can be changed every time the program is run.

The standard C library provides a number of functions for working with memory. The normal way to allocate a block of memory is to use the `malloc` function (which is an abbreviation for memory allocation). When you are finished with the block of memory, your program **must** hand it back, using the `free` function.

To demonstrate the use of `malloc` and `free` we will look at a very simple program snippet. The bit of code we will look at simply allocates some memory, we imagine that it uses it, and then the memory is freed. When the memory is allocated we will use the pointer `my_array` to reference it. When the program starts the pointer does not yet have a meaningful value.

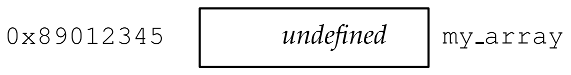
```
int main(void)
{
    int *my_array;

    my_array = malloc(size);

    ...

    free(my_array);

    return 0;
}
```



The call to `malloc` allocates a block of memory of a specified size. The code snippet has the word *`size`* where you would specify the size of the block of memory you require. We will look at how to specify memory sizes in a moment. Imagine that we specified that we wanted room for four integers.

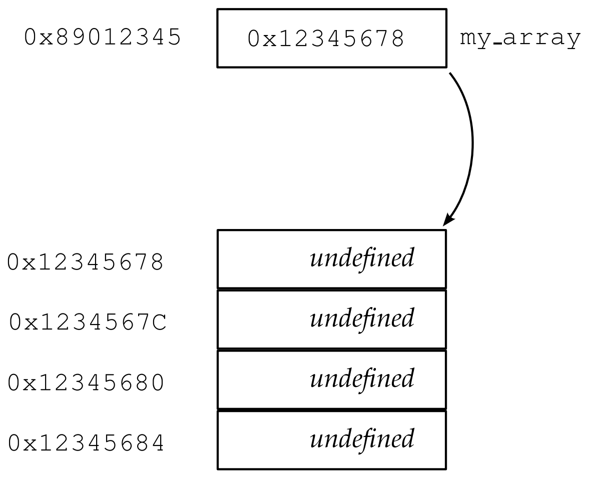
```
int main(void)
{
    int *my_array;

    my_array = malloc(size);

    ...

    free(my_array);

    return 0;
}
```



The diagram shows that we have allocated a block of memory, and that the pointer `my_array` references it. Notice that the block of memory has no variable name. It is not a variable; the only way we have to access this memory is using the `my_array` pointer.

When we have finished using the memory we must tell the system that we no longer need it. This allows the memory to be reused. If we did not do this the system would use up memory every time we ran the program. Allocating memory and forgetting to free it is called a *memory leak*.

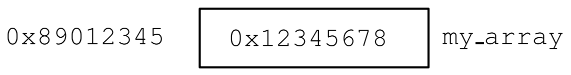
```
int main(void)
{
    int *my_array;

    my_array = malloc(size);

    ...

    free(my_array);

    return 0;
}
```



Notice that when you free a block of memory, the pointer that you used to access the memory (in this case `my_array`) does not change. However, the memory that this pointer references no longer belongs to the program. **You must not attempt to access memory you have freed**.

A call to the `malloc` function may not always succeed. You are asking the system for a block of memory. If it does not have enough memory to give you the `malloc` function will fail. If it does, the `malloc` function returns the special pointer value `NULL`. If a pointer is `NULL` it indicates that the pointer is invalid. If you try to use a `NULL` pointer your program will crash. It is good practice to always check to see if `malloc` returned a `NULL` or a valid pointer.

You have just seen how the `malloc` and `free` functions are used to allocate and free up blocks of memory. A call to the `malloc` function takes one argument: the size of the block you wish to allocate. C has a special operator called `sizeof` to help you calculate how much memory different things take up. The `sizeof` operator looks like a function call but is in fact built in to the C language. To use `sizeof` you specify the type name for a variable in brackets after the keyword `sizeof`. For example, if you want to know the size of an integer, you would write:

```
sizeof(int)
```

The size of four integers would be:

```
sizeof(int) * 4
```

Therefore, to allocate space for four integers using `malloc` you would write:

```
my_array = malloc(sizeof(int) * 4);
```

The code snippet we have been looking at is in the "lab8a" program. Open "lab8a.c" in the editor and change make sure the target is "lab8a".

Exercise 8.3: Investigating Memory Allocation

Have a look through the source code for the "lab8a" program. Do you understand what it is doing? Once you have tried executing it, try stepping through it and watching the variables. If you cannot understand what is happening, ask one of the demonstrators to explain it to you.

Function Reference: `malloc` — Memory Allocation

```
pointer = malloc(size);
```

e.g.

```
    some_pointer = malloc(sizeof(int));
    real_array = malloc(sizeof(double) * 20);
    string = malloc(sizeof(char) * 256);
```

The `malloc` function allocates a block of memory of the requested size and returns a pointer to the memory. Once the memory is not longer required, it must be deallocated using the `free` function.

Should the `malloc` function be unable to grant the request for memory, for example if there is insufficient remaining free memory on the system, the function will return the `NULL` pointer.

The `malloc` function is defined in `stdlib.h`

Syntax: The `sizeof` Operator

`sizeof(type)`

The `sizeof` operator calculates the amount of memory a type requires for storage. For example, the following:

```
sizeof(int)
```

calculates how much size is required by one integer variable. `sizeof` evaluates to a number, so it may be combined with mathematic operations such as multiplication. For example:

```
1024 * sizeof(char)
```

The most common use for the `sizeof` operator is when using the `malloc` memory allocation function.

Function Reference: `free` — Memory Deallocation

`free(pointer);`

The `free` function deallocates memory that was previously allocated using the `malloc` function. The `free` function takes one argument: a pointer to the block of memory to be deallocated.

- The `free` function should always be used to deallocate memory as soon as it is no longer required.
- You should **never** attempt to free a block of memory twice, or to free statically allocated memory such as arrays.
- Once a block of memory referenced by a pointer is deallocated, the pointer should **not** be used again, unless it is assigned a new value.

The `free` function is defined in `stdlib.h`

Exercise 8.4: Arrays and Memory Allocation

The "lab5" program allowed the user to enter data which the program placed in an array. Two statistical operations are then carried out: mean and standard deviation. At the moment, the memory in the array is statically allocated. The number of data items the user can entered is therefore fixed when the program is compiled.

Change the "lab5" program so that the user is asked how many data items they wish to enter at the beginning. You should then use `malloc` to dynamically allocate the data array. You can put a maximum limit of the number that the user enters if you want.

Remember to free the memory before the program ends.

Exercise 8.5: Strings and Memory Allocation

In a similar way to Exercise 8.4, change the "lab5b" program so that the user can decide how long a word they wish to be able to enter. You will need to be careful in how the `scanf` function call is constructed.

8.5 Dynamic Memory Allocation for Structures

Dynamically allocating memory for structures is very similar to allocating memory for arrays. The structure name is used with the `sizeof` operator to calculate the size of the block of memory required. For example, to allocate a block of memory for one `student_type` structure you would call `malloc` in the following way:

```
    student = malloc(sizeof(student_type));
```

Allocating memory for an array of structures is similar. For example:

```
    student_array = malloc(sizeof(student_type) * 20);
```

As you can see, this follows the same pattern as allocating memory for standard C types such as `int` and `double`.

Exercise 8.6: Structures and Memory Allocation

If you have been following the graphics option, open the "graphics3" program; if you have been following the music option, open the "music3" program (remember to change the targets appropriately). These projects currently use a statically allocated array of structures. Alter the program so that the array memory is dynamically allocated and freed.

Exercise 8.7: Reallocating Memory

If the user of your "graphics2"/"music3" program wanted to a a line/note whilst the program was running, how would you accommodate this (it would require more memory to be allocated). If the user removed a line/note could you free memory up? Have a go at implementing this into your program.

If the user is adding/removing lines/notes continuously allocating and freeing memory can make your program slow. How could you make this process more efficient?

8.6 Audio Programming - Part 2

Having learned about pointers and memory allocation you are now ready to continue with more advanced audio programming concepts such as audio effects and real-time audio processing. The following example code and exercises will take you through the creation of skeleton code that you can eventually use in your assignment.

8.6.1 Applying gain changes

This example demonstrates how to apply gain changes into an audio file. The program reads a file into memory, halves the amplitude of all samples and outputs the new data into a new file on disk. The code is exactly the same as that of example `audio01.c` apart from the memory allocation step and the processing step. In the previous example we didn't have to modify the signal in any way so no extra memory was needed. In this example though we want to modify our initial signal so we have to allocate some memory to hold the processed signal. The following line in file `audio02.c` does exactly this:

```
output.data = (float *) malloc (sizeof(float) * input.frames * input.channels);
```

The above line uses `malloc` to allocate a block of memory of size:

```
sizeof(float)* input.frames*input.channels}.
```

The `malloc` function return a pointer to that block of memory which we store in the 'data' field of the 'output' `SIGNAL` structure. The processing step iterates through each sample of the input data, applies a gain change and stores the result in the output data. After this simple processing the new file is written to disk using `wavwrite`.

The program `audio02` contains code that shows you how to change the amplitude of an audio file. Set the target to `audio02` and open the source code.

Exercise 8.8: Applying gain changes - add error checking

Modify the program `audio02.c` so that it includes error checking as shown in `audio01.c` example.

Exercise 8.9: Applying gain changes - inspect original and processed audio files

Run the program and compare, inspect and listen the original and the processed audio file using an audio editor (e.g Audacity).

Exercise 8.10: Applying gain changes - write processing function

Write a function that carries out the same processing found in as STEP4 in `audio02.c`. The function should have at least two input arguments (the input and output SIGNAL structures). What other parameter could be useful for this processing? Finally replace the code in 'STEP 4 PROCESSING' with your function and make sure your program works as expected.

8.6.2 Reversing a signal

This example demonstrates how to reverse a signal. The only thing that is different from previous examples is the processing step. Open `audio03.c` and inspect the code. Do you understand the processing part? If not ask a demonstrator for help. Run the program and compare, inspect and listen the original and the processed audio file using an audio editor (e.g Audacity).

Exercise 8.11: Reversing a signal - add error checking

Modify `audio03.c` so that it includes error checking.

Exercise 8.12: Reversing a signal - write a processing function

Write a function that carries out the same processing found in as 'STEP4 -PROCESSING' in `audio03.c` and make sure your program works as expected.

8.6.3 Delay effect

This program demonstrates how to implement a simple delay effect. In particular this is a feed-forward delay which creates a single echo (there is no feedback parameter). Open file `audio04.c` and inspect the code. You should notice that we have declared a couple of extra parameters namely M and g. M is the amount of delay we require in samples and g is gain parameter (the feed-forward parameter also known as the feed-forward coefficient). The most interesting part of this code is the processing block that implements the delay. The actual mathematical equation that we use is the following:

$$y[n] = x[n] + g \cdot x[n - M] \tag{8.1}$$

where $y[n]$ is the output signal, $x[n]$ the input signal, $M \in Z$ the amount of delay we require in samples and $g \in R$ the feed-forward coefficient (gain factor).

The role of the feed-forward coefficient is clear; when $g = 0$ the second term in equation (8.1) becomes zero so what we end up is $y[n] = x[n]$ meaning that there is no delay at all. When $g > 0$ the second term of (8.1) starts to contribute to the output. Note the term $x[n - M]$ for a moment. This term is actually the signal $x[n]$ delayed by M samples. This is depicted in figure 8.1.

And as a sanity check we can also use table 8.1 to confirm this (for M=2):

We can clearly see the signal $x[n - M]$ being delayed! Now if carefully look back into equation (8.1) we can see what it actually does: the output signal is equal to the input signal plus a delayed and attenuated version of the input signal.

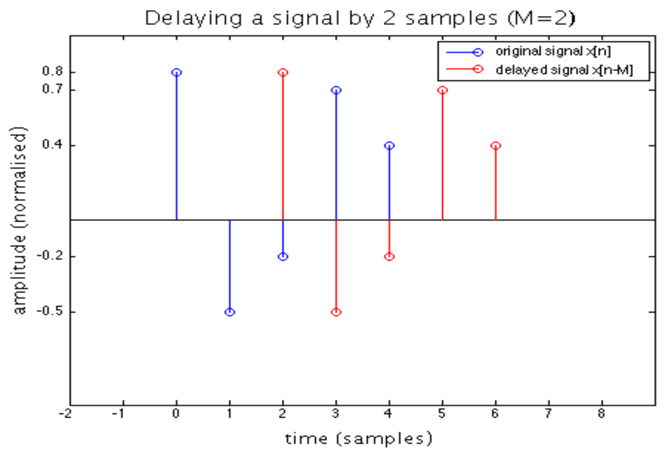


Figure 8.1: A signal x[n] (blue) and a delayed version (red)

Table 8.1: A signal delayed by 2 samples.

n	0	1	2	3	4	5	6	7
x[n]	0.8	-0.5	-0.2	0.7	0.4	0	0	0
n-M	-2	-1	0	1	2	3	4	5
x[n-M]	0	0	0.8	-0.5	-0.2	0.7	0.4	

Exercise 8.13: Delay effect - add error checking

Modify the `audio04.c` so that it includes error checking.

Exercise 8.14: Delay effect - convert samples to ms

The delay parameter M in this example is given in samples. Find the equation which converts this value into milliseconds (or seconds). Assume a known sampling rate (you will usually get this by the file information from within the `SIGNAL` structure or if you generate sound from code you can specify your own system sampling rate; a good value is $44100\ Hz$).

Exercise 8.15: Delay effect - write processing function

Write a function which implements the delay effect explained in this example. Then replace the code in 'STEP 4 PROCCESSING' in `audio04.c` with your own function and make sure your program works as expected.

8.6.4 Tremolo effect

This example applies a tremolo effect to a signal. Tremolo can be created by simply modulating an incoming signal with a low frequency oscillator (LFO). For our LFO we will use a sine wave. The mathematical formula of a sine wave is the following:

$$lfo(t) = \sin(2\pi \cdot f \cdot t + \phi) \tag{8.2}$$

where f is the frequency of the oscillation in Hz and ϕ the phase of the wave in radians. Using equation (6.1) into equation (6.2) we get:

$$t = n \cdot T_s = \frac{n}{F_s} \tag{8.3}$$

Finally substituting equation (8.3) into equation (8.2) we get the discrete version of the sine wave:

$$lfo[n] = \sin(2\pi \cdot \frac{f}{F_s} \cdot n + \phi) \tag{8.4}$$

We know that the output of the sin function ranges from -1 to 1. For the tremolo effect to work we want the LFO to range from 0 to 1 so we have to do some scaling. The following line produces the required scaled output:

$$lfo[n] = 0.5 \cdot (\sin(2\pi \cdot \frac{f}{F_s} \cdot n + \phi) + 1) \tag{8.5}$$

Equation (8.5) is actually implemented in the code of this example (without the phase parameter). After creating an array that contains the values of our LFO the only thing left to do is to multiply the LFO signal with the amplitude of the input signal and save the result into a new file on disk.

Open file `audio05.c` and inspect the code. Make sure you understand the lines of code that generate the LFO signal. If not please ask a demonstrator to explain.

Exercise 8.16: Tremolo effect - add error checking

Modify the program `audio05.c` so that it includes error checking.

Exercise 8.17: Tremolo effect - write processing function

Write a function that implements an LFO, that is a function that given a rate value in Hz and an amplitude value, produces a LFO signal as the one expressed in equation 3.5. You could also add a phase parameter.

8.6.5 Real time audio processing. Generating a sine wave.

This example generates a sine wave and plays it through the speakers for a few seconds. There is no user interaction (adding user interaction is an exercise) but the example goes through the fundamental concepts of real-time audio processing. The theory and design pattern that follows, although simplified, is true for many audio systems including digital audio workstations, plug-in architectures like VST and Audio Units, audio engines for games, software packages like MaxMSP and PureData and pretty much all audio solutions that require real-time audio support.

8.7 Summary

This laboratory re-examined arrays and strings to show their close relationship with pointers. You should now know that an array is simple a block of memory, allocated by the compiler, which is referenced by a pointer. Memory that is allocated by the compiler is called *statically allocated* memory.

Now that you have finished this lab you should know how to *dynamically* allocate memory using the `malloc` function call. You should know that all dynamically allocated memory should be deallocated using the `free` function call.

In the last part of the lab you should have used dynamic memory allocation with an array of structures.