stack will allocate the amt of memory during declaration
- so long a[10] makes 10 slots of long

ALWAYS BE AWARE OF
1) array initialisation
2) avoiding the use of variable-length arrays
3) determining the size of the array
4) how to read arrays with the cs1010 i/o library

Unit 15: Array

Learning Objectives

After this unit, students should:

- understand the concept of a compound data type
- be able to manipulate arrays in C, including:
  - declaring an array of a particular type and length
  - accessing the elements within an array
- be aware that when declaring arrays, they are done so with a fixed-length
- be aware that array and non-array variables are treated differently in C and specifically, the concept of the array decay operation.
- be aware of how array variables are specifically used to reference different elements in the array
- be aware that we cannot reassign array values through a typical assignment statement
- be able to define an array as a function parameter
- be aware of the difference of pass by value and pass by reference, and that arrays, as parameters correspond to the latter, and consequently, the need to also indicate the length of an array as another parameter
- be aware of potential errors when array variables are treated as non-array variables and vice versa
- be aware that we cannot change the length of an array, only declare a new one of different length
- be aware that we may check the length of an array using the sizeof operator
- be aware that VLA should be avoided
- be able to use the CS1010 library functions to read arrays of longs or doubles

Your First Compound Data Type

We now look at the first of the two compound data types in C -- arrays.

A variable can be declared to be of an array, in which case it can hold one or more values. An array variable can only store values of the *same* type T. We say that the array variable is an array of T. For instance, we can declare a variable `marks` to be an array of `long`, in which case, `marks` can hold one or more `long` values.

Array Declaration

The declaration syntax for an array in C takes the following form:

```
long marks[10];
```

We use the square bracket `[` and `]` to indicate that the variable `marks` is an array. The number `10` indicates that `marks` holds 10 `long` values.

Once declared, the variables in the array are uninitialized and will contain whatever value happened to be in the memory at that time.

Accessing the Array Elements

We can access the array elements using the *index* of the element, starting from 0. For instance, to initialize the marks for the first three students to 1, 2, 4, respectively, we can write:

```
long marks[10];
marks[0] = 1;
marks[1] = 2;
marks[2] = 4;
```

Same syntax, two different meanings

Beginners tend to confuse with the following:

```
long marks[10];
marks[10] = 1;
```

`marks[10]` appears twice but it has two different meanings. In the first line, we are *declaring* an array called `marks` of size 10. In the second line, we are *using* the array `marks`, and accessing the element with index 10 (which is not valid).

element in array is simply &a[i] == starting address of array + (i * unit (eg: long))

Array Initialization

Initializing a large array using the method above could be tedious. Alternatively, we can initialize an array using a list of values when we declare the array.

```
long marks[10] = {1, 2, 3, 1, 5, 10, 10, 4, 5, 3, };
```

If we do not specify a value during initialization, it will be set to 0 by default.

```
long marks[10] = {1, 2, 3, 1, 5, 10, 10, 4, };
// marks[8] and marks[9] are both initialized to 0
```

Note that, after the declaration, we can no longer using this technique to reinitialized or initialize the array.

```
long marks[10];
marks = {1, 2, 3, 1, 5, 10, 10, 4, 5, 3, };  // error
```

Example 1: Array As Lookup Table

One way the array is useful is that it can be used as a lookup table.

Consider the following function `days()`, which, given a month, return the number of days from the 1st of January until the 1st of the month. So `days(1)` returns 0, `days(2)` returns 31 (since January has 31 days), `days(3)` returns 31 + 28 = 59 (assuming non-leap year), etc.

```
long days(long month)
{
  long days_since = 0;
  if (month == 2) {
    days_since = 31;
  } else if (month == 3) {
    days_since = 31 + 28;
  } else if (month == 4) {
    days_since = 31 + 28 + 31;
  } else if (month == 5) {
    days_since = 31 + 28 + 31 + 30;
  } else if (month == 6) {
    days_since = 31 + 28 + 31 + 30 + 31;
  } else if (month == 7) {
    days_since = 31 + 28 + 31 + 30 + 31 + 30;
  } else if (month == 8) {
    days_since = 31 + 28 + 31 + 30 + 31 + 30 + 31;
  } else if (month == 9) {
    days_since = 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
  } else if (month == 10) {
    days_since = 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
  } else if (month == 11) {
    days_since = 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
  } else if (month == 12) {
    days_since = 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30;
```

```
26    }
27    return days_since;
28  }
```

The code is ugly and bug-prone. Consider an alternative solution using an array.

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

The array above is initialized with the number of days in a month. `days_in_month[0]` stores the number of days in January, `days_in_month[1]` stores the number of days in February, etc.

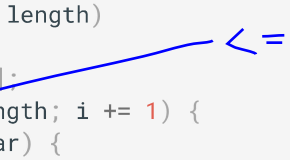
The code above can then be written as:

```
1  long days(long month)
2  {
3    long days_in_month[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
4  31};
5    long days_since = 0;
6    for (long i = 0; i < month - 1; i += 1) {
7      days_since += days_in_month[i];
8    }
9    return days_since;
   }
```

Example 2: Array As List

We can now revisit the flowchart for *max* and write the corresponding code. The code would look like this:

```
1  long max(long list[], long length)
2  {
3    long max_so_far = list[0];
4    for (long i = 1; i != length; i += 1) {
5      if (list[i] > max_so_far) {
6        max_so_far = list[i];
7      }
8    }
9    return max_so_far;
10 }
```



Please see the Appendix for the complete code.

Note that, in the type of the array passed into the function above, we only need to use `[]` without specifying the length. It is also almost always necessary to pass in the number of elements in the array together with the array so that we know how many elements are there to process. To understand why, we have to understand something called *array decay*.

Array and Pointers

The relationship between array and pointer can be confusing to beginners. Some might even think that arrays are pointers due to the way they are passed around and used. But *arrays are not pointers*. The relationship between the two, boils down to a very simple rule, as follows.

In C, the name of the variable of an array is treated differently from a non-array variable. If we declare an array

```
1  type name[num_of_elems];
```

then any reference to `name` is a synonym to `&name[0]` whenever we need the value of the array. In other words, `name` is the pointer to the first element in the array. This is known as the "array decay" operation.

There are several implications to this.

First, it is not possible to compare two arrays or assign one array to another.

```
1  long a[2] = {0, 1};
2  long b[2] = {0, 1};
3
4  if (a == b) {  // always false
5    :
6  }
7
8  b = a; // not possible
```

Line 4 above is equivalent to comparing `&a[0]` to `&b[0]`, due to array decay, and therefore is always false (since the array elements do not have the same memory address).

Line 8 above is equivalent to assigning `&a[0]` to `&b[0]`, as we seen in (Unit 14)[14-pointers.md], we cannot change the memory address of a variable since this is determined by the OS.

Second, the expression `name[i]` is actually the same as `*(name + i)`. Although we should always write `name[i]` as it is easier to understand, internally, this translates to accessing the value stored at the i-th location after the first element of the array. (Revisit Unit 14 if you are unfamiliar with pointer arithmetic).

Third, when we pass an array into a function, we are only passing in the address of the first element.

Consider the example below:

```
1  long max(long list[], long length)
2  {
3    long max_so_far = list[0];
4    for (long i = 1; i != length; i += 1) {
5      if (list[i] > max_so_far) {
6        max_so_far = list[i];
7      }
8    }
9    return max_so_far;
10 }
11
12 int main()
13 {
14   long a[10] = {1, 2, 3, 4, 1, 9, 10, 44, -1, -5};
15   cs1010_println_long(max(a, 10));
16 }
```

On Line 14, we pass `a` into the function `max`. Due to array decay, we are not passing in the whole array, but only the `&a[0]`. Inside `max`, we no longer have access to the whole array, but only the address of the first element. Luckily, due to pointer arithmetic and equivalence of `list[i]` to `*(list + i)`, we can still access the elements of the array.

Due to this, the size of `list` in the function parameter does not matter, and we need to pass in the size of the array `length` to `max` so that inside `max` we know the size of the array that we are dealing with.

Array decay also means that, when passing in an array as an argument to a function, we can very well write it as:

```
1  long max(long *list, long length)
2  {
3      :
4  }
```

For readability, however, we should convey to the reader of the code that `list` is an array and not just a pointer to `long`, thus *we should still use the `[]` notation.*

Another implication of array decay when passing an array into a function, is that it is possible to write programs that behave incorrectly, without any compiler error or warning. Consider the following:

```
1  int main()
2  {
3    long a = 0;
4    cs1010_println_long(max(&a, 10));
5  }
```

The compiler `clang` would happily compile this and generate an executable. When executed, it would print a gibberish to the standard output.

Array Index Out of Bound

A common bug when we work with arrays is accessing a location beyond what is allocated to the array. Unlike other languages like Java, which checks the bound for you, C does not. So we could write the following code and it would compile perfectly.

```
1  int main()
2  {
3    long a[10];
4    for (long i = 0; i <= 10; i += 1) {
5      a[i] = 1;
6    }
7  }
```

must be less than size of array

Running this, however, would lead to *memory corruption*, since we are writing to `a[10]` (the 11-th element) but we only asked for 10 elements for the array `a`. So we are writing to a memory that we are not supposed to and thus causing your program to behave incorrectly.

Similarly, reading from a memory location that we are not supposed to could lead to a misbehave program. You have seen this in Problem 1.1(d).

Other Facts About Arrays

Variable Length Array

C also allows declaration of an array where the number of elements depends on the value of a variable.

```
1  long num_of_students = 10;
2  long marks[num_of_students];
```

Such arrays, where the number of elements (or length) depends on the value of a variable, are sometimes called *variable-length array*, or VLA. This is a misnomer since once the array is created, the length is fixed. Changing the value of the variable `num_of_students` above will not change the length of `marks`.

VLA *should not be used* in CS1010. Even outside of CS1010, VLA should be used with extreme care. A VLA is allocated on the stack, which typically has very limited memory. If the stack runs out of memory, your program would crash! The situation is worse if the size

of the VLA is read as an input from the users -- this implies that an external user could enter a malicious input to crash your program. You should use dynamically allocated array instead (see below for a teaser and a later unit for details).

Skipping Elements During Initialization

If we have a large array, and we want most of it to be initialized to 0, and only some non-zero, we can use *element designators*, putting the index of the element we want to initialize to non-zero in square brackets `[` and `]`.

```
long vector[100] = {1, [5] = 2, 3, [99] = -1};
```

This statement initializes `vector[0]` to 1, `vector[5]` to 2, `vector[6]` to 3, and `vector[99]` to -1. The rest of the elements will be 0.

Skipping the Size in Declaration

If you supply an initialization list, the number of elements already indicates the length to the compiler, so you can skip the length.

```
long marks[] = {1, 3, 2, 8, 5,}
```

This makes it easy to add or remove items from the array, without having to remember to keep the array length consistent.

Determining the Number of Elements in the Array

C provides a `sizeof` operator, which returns the number of bytes allocated to a type. We can use `sizeof long` for instance, to determine the number of bytes allocated to `long` on a platform. We can also use `sizeof` on a variable instead of the type. This becomes useful to determine, programmatically, the length of an array (esp if the array length is skipped in the array declaration). We can calculate the number of elements in `marks` with

```
long num_of_elem = sizeof marks/sizeof marks[0];
```

Note that array decay does not apply for only two operators: the `sizeof` operator, and the address-of `&` operator. So the expression above works as intended.

Dynamically Allocated Array

It is often not possible to determine the length of the array beforehand. So, it is useful to be able to allocate an array with a length that is determined during runtime (not hard-coding the length of the array in the program).

For instance, if I want to keep a `marks` array for a module, it is unclear how big I should set the array to. How big is big enough? While I can use a variable-length array for this purpose, it is not ideal -- if the system does not have enough memory to store the array, the program would simply crash with a segfault and there is no way to recover from this.

For this reason, it is useful to request memory from the OS which we will manage ourselves in our program. Unlike memory space on the call stack which is managed entirely by the OS, there is another region of memory called the *heap*, which we can use. We can request for memory from *heap* using the method `calloc()` and return the memory to the heap when we are done with `free()`. We will visit these in more detail later, but this shallow understanding is enough for now.

CS1010 I/O Library

To wrap up this unit, we will look at the CS1010 library functions that help us read in an array of either `long` values or `double` values. These functions, `cs1010_read_long_array` or `cs1010_read_double_array` takes in a parameter, which is the number of elements to read, and it returns a pointer to the array allocated within the function.

For instance, to read in an array of 100 integers, we can write:

```
long *marks;
marks = cs1010_read_long_array(100);
for (long i = 0; i < 100; i += 1) {
  cs1010_println_long(marks[i]);
}
```

This should be straightforward enough. There are, however, two cases to consider. What if the OS failed to allocate the memory for our array? In this case, marks would be `NULL` and access `marks[i]` would cause a segfault. Second, we must return the memory allocated to us back to the OS once we are done. To let go of this memory, we call the function `free`. The complete code looks like this:

```
long *marks;
marks = cs1010_read_long_array(100);
if (marks == NULL) {
  // signal error and return
}

for (long i = 0; i < 100; i += 1) {
  cs1010_println_long(marks[i]);
```

```
}
// do other things to marks
  :
free(marks);
```

Problem Set 15

Problem 15.1

Write the function `average` that takes an array of k integers and k and returns the average of the k values in the array.

Problem 15.2

Explain why the following would lead to senseless output:

```
int main()
{
  long a = 0;
  printf("%ld\n", max(&a, 10));
}
```

How about the following? Would the output be correct?

```
int main()
{
  long a = 0;
  printf("%ld\n", max(&a, 1)); // change 10 to 1
}
```

Problem 15.3

Explain how the following code iterates through every element in the list when called with an array of length `length` as the first argument.

```
long max(long *list, long length)
{
  long max_so_far;
  long *curr;

  max_so_far = *list;
  curr = list + 1;
  for (long i = 1; i != length; i += 1) {
    if (*curr > max_so_far) {
      max_so_far = *curr;
    }
    curr += 1;
  }
```

```
  return max_so_far;
}

```

Appendix: Complete Code Example

```
#include "cs1010.h"

long days_till_beginning_of(long month)
{
  long days_in_month[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31};
  long num_of_days = 0;
  for (long i = 0; i < month - 1; i += 1) {
    num_of_days += days_in_month[i];
  }
  return num_of_days;
}

int main()
{
  long month = cs1010_read_long();
  long day = cs1010_read_long();
  cs1010_println_long(day + days_till_beginning_of(month));
}
```

```
long max(long list[], long length)
{
  long max_so_far = list[0];
  for (long i = 1; i != length; i += 1) {
    if (list[i] > max_so_far) {
      max_so_far = list[i];
    }
  }
  return max_so_far;
}

int main()
{
  long a[10] = {1, 2, 3, 4, 1, 9, 10, 44, -1, -5};
  cs1010_println_long(max(a, 10));
}
```