Arrays

In this lesson, we will discuss arrays, how arrays are declared and how data is accessed in an array.

WE'LL COVER THE FOLLOWING

If we need five different values to perform some calculations, we will have to store those values in five different variables. The declaration of those five variables will look like this:

```
double value_1;
double value_2;
double value_3;
double value_4;
double value_5;
```

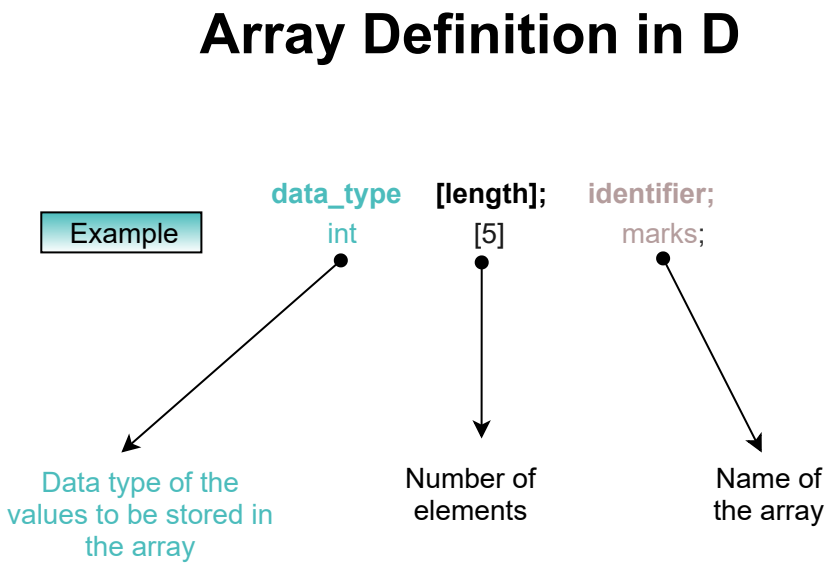
This method of defining variables individually does not scale to cases where even more variables are needed. Imagine needing a thousand values; it is almost impossible to define a thousand variables from `value_1` to `value_1000`.

Arrays are useful in such cases: the **array** feature allows us to define a single variable that stores multiple values of the same type together. Although simple, arrays are the most commonly used data structures to store a collection of values.

This chapter covers only some of the features of arrays. More features will be

introduced later in the slices and other array features lesson.

Definition



Array definition in D

The definition of array variables is very similar to the definition of normal variables. The only difference is that the number of values associated with the variable is specified in square brackets. We can contrast the two definitions as follows:

```
int singleValue;
int[10] arrayOfTenValues;
```

The first line above is the definition of a variable that stores a single value, just like the variables that we have defined so far. The second line is the definition of a variable that stores ten consecutive values. In other words, it stores an array of ten integer values. You can also think of it as defining ten variables of the same type.

Accordingly, the equivalent of the five separate variables above can be defined as an array of five values using the following syntax:

```
double[5] values;
```

That definition can be read as 5 double values. Note that I have chosen to name the array variable as plural to avoid confusing it with a single-valued variable. Variables which only store a single value are called **scalar**

variables.

In summary, the definition of an array variable consists of the type of values, the number of values, and the name of the variable that refers to the array of values:

```
type_name[value_count] variable_name;
```

Array definition

The type of the values can also be a user-defined type. For example:

```
/* An array that holds the weather information of all cities. Here, the bo
ol values may mean

false: overcast
true : sunny
*/
bool[cityCount] weatherConditions;

// An array that holds the weights of a hundred boxes
double[100] boxWeights;

// Information about the students of a school
StudentInformation[studentCount] studentInformation;
```

Containers and elements

Data structures that bring elements of a certain type together are called **containers**. According to this definition, arrays are containers.

Example

Consider that an array that holds the air temperatures of the days in July can have 31 double values together and form a container of elements of type `double`.
The variables of a container are called **elements**. The number of elements in an array is called the **length** of the array.

Accessing the elements

In order to differentiate the variables declared earlier in the lesson, we

appended an underscore and a number to their names as in `value_1`. This is not possible, nor necessary, when a single array stores all the values under a single name. Instead, the elements are accessed by specifying the element number within square brackets:

```
values[0]
```

That expression can be read as the element with the number 0 of the array named `values`. Here, 0 can be read as the index of the first element. In other words, instead of typing `value_1` one must type `values[0]` with arrays. There are two important points worth stressing here:

- **The numbers start at location zero:** Although humans assign numbers to items starting with 1, *the numbers in arrays start at 0*. The values that we have numbered as 1, 2, 3, 4, and 5 before are numbered as 0, 1, 2, 3, and 4 in the array. This variation can confuse new programmers.
- **Two different uses of the [] characters:** Don't confuse the two separate uses of the `[]` characters. When defining arrays, the `[]` characters are written after the type of the elements and specify the number of elements. When accessing elements, the `[]` characters are written after the name of the array and specify the number of the elements that is being accessed:

```
// This is a definition. It defines an array that consists of 12 element
s. This array is used to hold the number of days in each month.

int[12] monthDays;

// This is access. It accesses the element that corresponds to December an
d sets its value to 31.

monthDays[11] = 31;

// This is another access. It accesses the element that corresponds to Jan
uary, the value of which is passed to writeln.

writeln("January has ", monthDays[0], " days.");
```

> **Reminder:** The element numbers of January and December are 0 and 11 respectively, not 1 and 12.

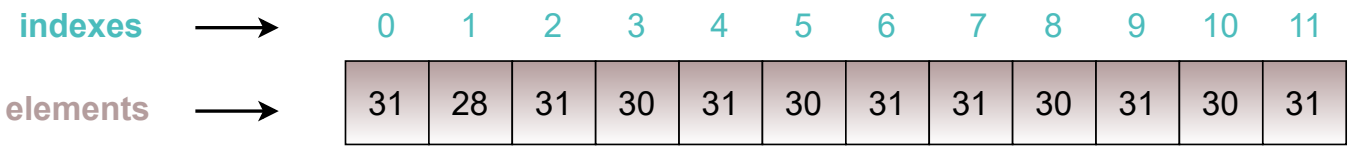
Index

The location of an element in the array is called its **index**, and the act of accessing an element is called **indexing**.
An index need not be a constant value; the value of a variable can also be used as an index, making arrays even more useful. For example, the month can be determined by the value of the `monthIndex` variable below:

```
int monthIndex = 2;
writeln("This month has ", monthDays[monthIndex], " days.");
```

When the value of `monthIndex` is 2, the expression above would print the value of `monthDays[2]`, which is the number of days in March.

Only the index values between zero and one less than the length of the array are valid. For example, the valid indexes of a three-element array are 0, 1 and 2. Accessing an array with an invalid index causes the program to be terminated with an error.
Arrays are containers where the elements are placed side by side (contiguously) in the computer's memory. For example, the elements of the array that hold the number of days in each month can be shown like the following (assuming a year when February has 28 days):



Array indexing

> **Note:** The indexes above are for demonstration purposes only; they are not stored in the computer's memory.
> The element at index 0 has the value 31 (number of days in January), the element at index 1 has a value of 28 (number of days in February) and so on.

Fixed-length arrays vs. dynamic arrays

If the length of an array is specified when the program is written, the array is a **fixed-length array**. Fixed-length arrays are also known as *static arrays*.

When the length can change during the execution of the program, that array is a **dynamic array**.

So far the arrays we have defined are fixed-length arrays because their element counts are specified at the time when the program is written. The lengths of those arrays cannot be changed during the execution of the program. To change their lengths, the source code must be modified and the program must be recompiled.

Defining dynamic arrays is simpler than defining fixed-length arrays because omitting the length makes a dynamic array:

```
int[] dynamicArray;
```

The length of such an array can increase or decrease during the execution of the program.

Using `.length` to get or set the number of elements

Arrays have properties as well. However, which we will see only `.length` here. `.length` returns the number of elements of the array:

```
writeln("The array has ", array.length, " elements.");
```

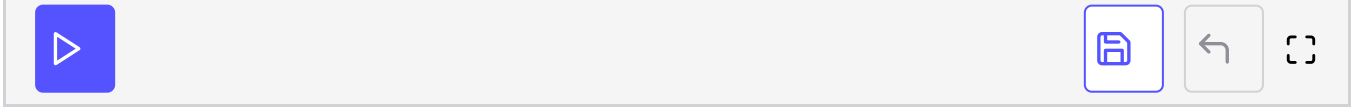
Additionally, the length of dynamic arrays can be changed by assigning a value to this property:

```
import std.stdio;

void main() {

    int[] array; // initially empty
    array.length = 5; // now has 5 elements
    writeln(array.length);

}
```



In the next lesson, we will see a simple use case of arrays that shows the

In the next lesson, we will see a simple use case of arrays that shows the significance of arrays.