Slices and Other Array Features

In this lesson, we will discuss slices and other array features like array.length, .dup etc.

WE'LL COVER THE FOLLOWING

- Slices
- Using `$`, instead of `array.length`
- Using `.dup` to copy
- Assignment

We have seen earlier in this chapter how elements are grouped as a collection in an array. This lesson further elaborates on that concept by explaining the features of the arrays.

Before going any further, here are a few brief definitions of some terms that happen to be close in meaning:

- **Array:** the general concept of a group of elements that are located side by side and are accessed by indexes
- **Fixed-length array (static array):** an array with a fixed number of elements; this type of array owns its elements and doesn't allow one to change the length of the array
- **Dynamic array:** an array that can gain or lose elements; this type of array provides access to elements that are owned by the D runtime environment
- **Slice:** another name for the dynamic array

When we use *slice*, it will mean a dynamic array. On the contrary, when we use the term *array* we mean either a slice or a fixed-length array with no distinction.

Slices

Slices have the same feature as dynamic arrays. Slices are called dynamic arrays since we can change their length at runtime, and they are called **slices** for providing access to portions of other arrays. They allow using those portions as if they are separate arrays.

Slices are defined by a *number range syntax* that corresponds to the indexes that specify the beginning and the end of the range:

```
beginning_index .. one_beyond_the_end_index
```

In the number range syntax, the beginning index is a part of the range, but the end index is outside of the range:

```
/* ... */ = monthDays[0 .. 3]; // 0, 1, and 2 are included; but not 3
```

> **Note:** Number ranges are different from Phobos ranges. Phobos ranges are about struct and class interfaces.

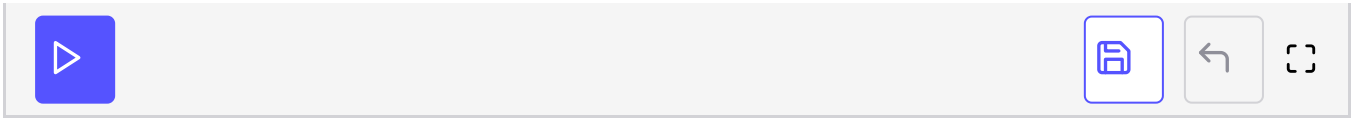
As an example, we can slice the monthDays array to be able to use its parts as four smaller sub-arrays:

```
int[12] monthDays =
[ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ];

int[] firstQuarter = monthDays[0 .. 3];
int[] secondQuarter = monthDays[3 .. 6];
int[] thirdQuarter = monthDays[6 .. 9];
int[] fourthQuarter = monthDays[9 .. 12];
```

The four variables in the code above are slices; they provide access to four parts of an already existing array. An important point worth stressing here is that those slices do not have their own elements. They merely provide access to the elements of the actual array. Modifying an element of a slice modifies the element of the actual array. To see this, let's modify the first elements of each slice and then display the actual array:

```
import std.stdio;
```

```
import std.stdio;

void main() {

    int[12] monthDays =[ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ];

    int[] firstQuarter = monthDays[0 .. 3];
    int[] secondQuarter = monthDays[3 .. 6];
    int[] thirdQuarter = monthDays[6 .. 9];
    int[] fourthQuarter = monthDays[9 .. 12];

    firstQuarter[0] = 1;
    secondQuarter[0] = 2;
    thirdQuarter[0] = 3;
    fourthQuarter[0] = 4;

    writeln(monthDays);

}
```



Slicing example

Each slice modifies its first element, and the corresponding element of the actual array is affected.

We have seen earlier that valid array indexes are from 0 to one less than the length of the array. For example, the valid indexes of a 3-element array are 0, 1 and 2. Similarly, the end index in the slice syntax specifies one beyond the last element that the slice will be providing access to. For that reason, when the last element of an array needs to be included in a slice, the length of the array must be specified as the end index. For example, a slice of all elements of a 3-element array would be array[0...3].
An obvious limitation is that the beginning index cannot be greater than the end index:

```
int[3] array = [ 0, 1, 2 ];
int[] slice = array[2 .. 1]; // ← run-time ERROR
```

It is legal to have the beginning and the end indexes to be equal. In that case, the slice is empty. Assuming that the index is valid:

```
int[] slice = anArray[index .. index];
writeln("The length of the slice: ", slice.length);
```

The output of the code will be:

```
The length of the slice: 0
```

Using `$`, instead of `array.length`

When indexing, `$` is a shorthand for the length of the array:

```
import std.stdio;

void main() {
    int[3] array = [ 0, 1, 2 ];
    writeln(array[array.length - 1]); // the last element
    writeln(array[$ - 1]); // the same thing
}
```

Using `.dup` to copy

Short for “duplicate,” the `.dup` property makes a new array from the copies of the elements of an existing array:

```
double[] array = [ 1.25, 3.75 ];
double[] theCopy = array.dup;
```

As an example, let’s define an array that contains the number of days of the months of a leap year. One method is to take a copy of the non-leap-year array and then increment the element that corresponds to February:

```
import std.stdio;

void main() {
    int[12] monthDays =
        [ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ];

    int[] leapYear = monthDays.dup;

    ++leapYear[1];   // increments the days in February

    writeln("Non-leap year: ", monthDays);
    writeln("Leap year    : ", leapYear);
}
```

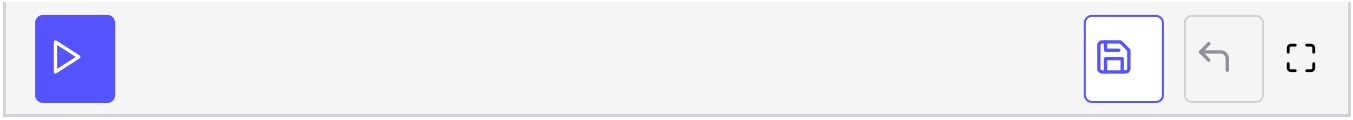
Using .dup to copy

> **Note**: If `.dup` property is not used, then the change will reflect in both the arrays.

Assignment

We have seen so far that the assignment operator modifies the values of variables. It is the same with fixed-length arrays:

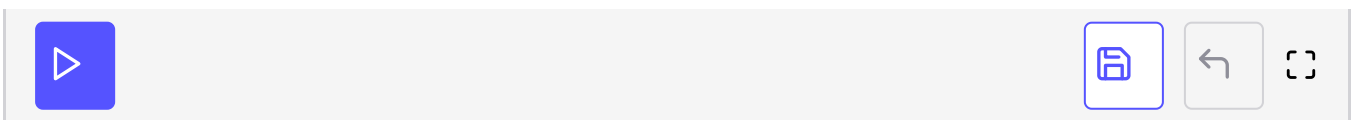
```
import std.stdio;

void main() {
    int[3] a = [ 1, 1, 1 ];
    int[3] b = [ 2, 2, 2 ];

    writeln(a);

    a = b; // the elements of 'a' become 2
    writeln(a);
}
```



Assignment operator and arrays

The assignment operator has a completely different meaning for slices: it modifies the slice to start providing access to the elements of the `evens` array:

```
import std.stdio;

void main() {
    int[] odds = [ 1, 3, 5, 7, 9, 11 ];
    int[] evens = [ 2, 4, 6, 8, 10 ];
    int[] slice; // not providing access to any elements yet

    slice = odds[2 .. $ - 2];
    writeln(slice);

    slice = evens[1 .. $ - 1];
    writeln(slice);
}
```



Assignment operator and slices

Above, `slice` does not provide access to any element when it is defined. `slice` is then used to provide access to some of the elements of `odds` and later to some of the elements of `evens`.

In the next lesson, we will see the termination of sharing in slices.