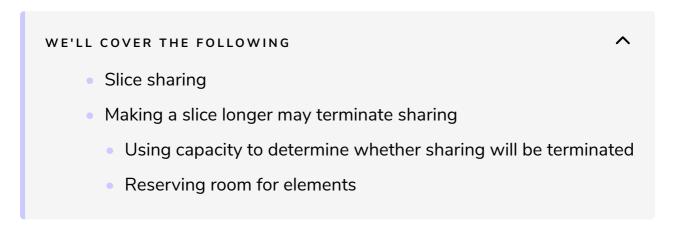
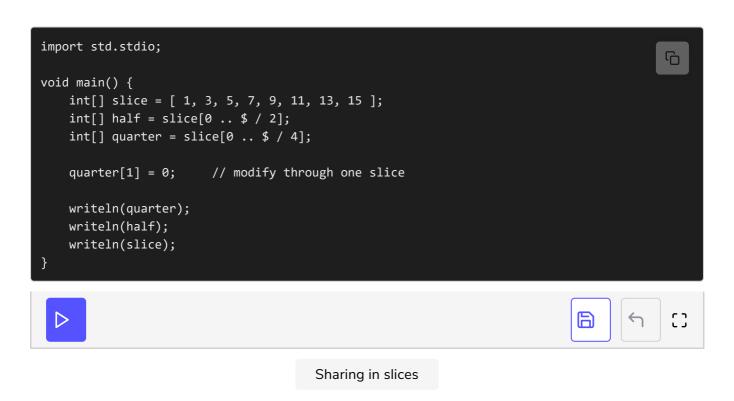
Slices: Termination of Sharing

In this lesson, we will discuss the concept of termination of sharing in slices.



Slice sharing

It is possible to access the same elements by more than one slice. For example, the first two of the eight elements below are being accessed through three slices:



The effect of the modification to the second element of quarter is seen through all slices.

Making a slice longer may terminate sharing

When viewed this way, slices provide shared access to elements. This sharing poses the question of what happens when a new element is added to one of the slices. Since multiple slices can provide access to the same elements, there may not be room to add elements to a slice without stomping on the elements of other slices of the same array.

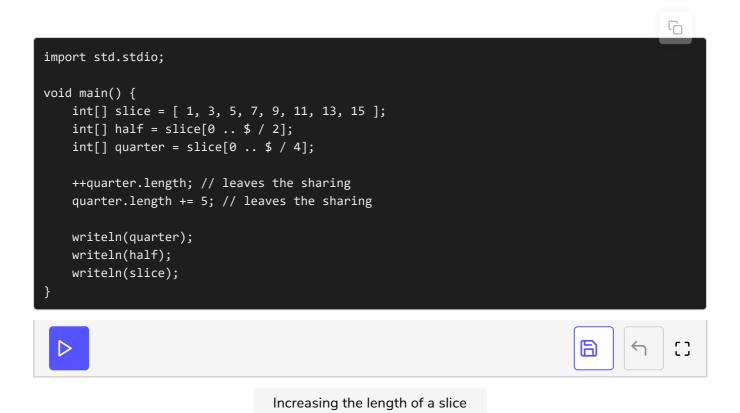
D disallows element stomping and answers this question by terminating the sharing relationship if there is no room for the new element. The slice that has no room to grow leaves the sharing. When this happens, all the existing elements of that slice are copied to a new place in memory automatically and the slice starts providing access to these new elements.

To see this in action, let's add an element to quarter before modifying its second element:

```
quarter ~= 42; // this slice leaves the sharing because there is no room f
or the new element
quarter[1] = 0; // for that reason this modification does not affect the o
ther slices
```

The output of the program shows that the modification to the quarter slice does not affect the others:

Explicitly increasing the length of a slice terminates the sharing as well:



On the other hand, shortening a slice does not affect sharing. Shortening the slice merely means that the slice now provides access to fewer elements:



As can be seen from the output, the modification through d is seen through a; the sharing is still in effect.

Reducing the length in different ways does not terminate the sharing either:

```
import std.stdio;

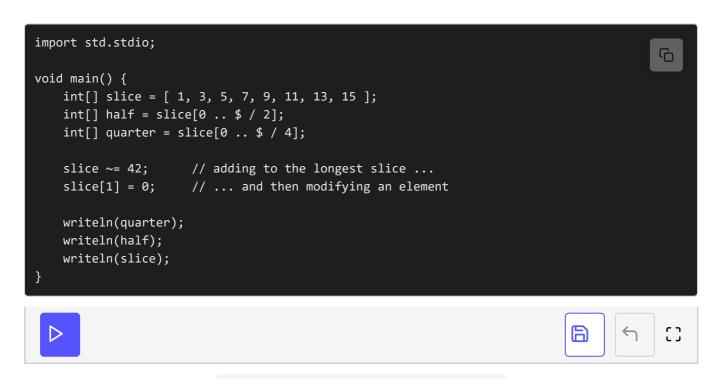
void main(){
   int[] a = [ 1, 11, 111 ];
```

Sharing continues even after reducing the length

The sharing of elements is still in effect.

Using capacity to determine whether sharing will be terminated

There are cases when slices continue sharing elements even after an element is added to one of them. This happens when the element is added to the longest slice and there is room at the end of it:



Addition of element to the longest slice

The capacity property of slices determines whether the sharing will be terminated if an element is added to a particular slice. capacity is actually a function, but this distinction does not have any significance in this discussion. The value of capacity has the following meanings:

- When its value is **0**, it means that this is not the *longest original slice*. In this case, adding a *new* element would definitely relocate the elements of the slice and the sharing would terminate.
- When its value is non-zero, it means that this is the *longest original slice*. In this case, capacity denotes the *total* number of elements that this slice can hold without the need to be copied. The number of new elements that can be added can be calculated by *subtracting the actual length of the slice from the capacity value*(slice.capacity slice.length). If the length of the slice equals its capacity, then it indicates that there is no room for more elements. In this case, the slice will be copied to a new location if one more element is added.

Accordingly, a program that needs to determine whether the sharing will terminate should use logic similar to the following:

```
if (slice.capacity == 0) {
    /* Its elements would be relocated if one
    more element is added to this slice. */
    // ...
} else {
    /* This slice may have room for new elements
    before needing to be relocated. Let's
    calculate how many: */
    auto howManyNewElements = slice.capacity - slice.length;
    // ...
}
```

An interesting corner case is when there is more than one slice comprising all elements. In such a case all slices report to have capacity:

```
import std.stdio;

void main() {
    // Three slices to all elements
    int[] s0 = [ 1, 2, 3, 4 ];
    int[] s1 = s0;
    int[] s2 = s0;

    writeln(s0.capacity);
    writeln(s1.capacity);
    writeln(s2.capacity);
}
```







Slices capacity

However, as soon as an element is added to one of the slices, the **capacity** of the others drop to **0**:

```
import std.stdio;

void main() {
    // Three slices to all elements
    int[] s0 = [ 1, 2, 3, 4 ];
    int[] s1 = s0;
    int[] s2 = s0;

    s1 ~= 42; // \(in s1\) becomes the longest

    writeln(s0.capacity);
    writeln(s1.capacity);
    writeln(s2.capacity);
}

Change in capacity
```

Since the slice with the added element is now the longest, it is the only one with capacity greater than 0:

```
0
7 ← now only s1 has capacity
0
```

Reserving room for elements

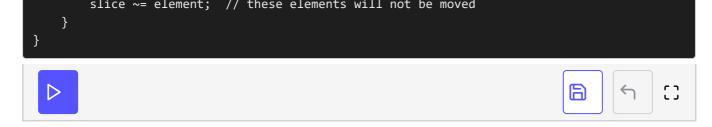
Both *copying elements* and *allocating new memory* to increase capacity have some cost. For that reason, appending an element can be an expensive operation. When the number of elements to append is known beforehand, it is possible to reserve capacity for the elements using the .reserve function:

```
import std.stdio;

void main() {
   int[] slice;

   slice.reserve(24);
   writeln(slice.capacity);

foreach (element; 0 .. 17) {
```



Reserving room for elements

The elements of slice would be moved only after there are more than 31 elements.

In the next lesson, we will see operations on all elements.