Associative Arrays

This lesson explains how associative arrays can be used in D and the different methods that can be used with associative arrays.

WE'LL COVER THE FOLLOWING

Associative arrays

Associative array is a feature that is found in most modern high-level languages. They are very fast data structures that work like mini databases and are used in many programs.

We have seen in the arrays lesson that plain arrays are containers that store their elements side-by-side and provide access to them by index. An array that stores the names of the days of the week can be defined like this:

```
string[] dayNames =
[ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sun
day" ];
```

> Note: `string` datatype will be covered later in this chapter.

The name of a specific day can be accessed by its index in that array:

```
writeln(dayNames[1]); // prints "Tuesday"
```

How are associative arrays different from plain arrays?

- The fact that plain arrays provide access to their values through index numbers can be described as an association of indexes with values. In other words, arrays map indexes to values. Plain arrays can use only integers as indexes.
  Associative arrays allow indexing not only using integers but also using any other type. They map the values of one type to the values of another type. The values of the type that associative arrays map from are called **keys** rather than indexes. *Associative arrays store their elements as key-value pairs*.

- Associative arrays are implemented in D using a *hash table*. Hash tables are among the fastest data structures for storing and accessing elements. Other than in rare pathological cases, the time it takes to store or access an element is independent of the number of elements that are in the associative array.
  The high performance of hash tables comes at the expense of storing the elements in an unordered way. Also, unlike arrays, the elements of hash tables are not stored side-by-side.

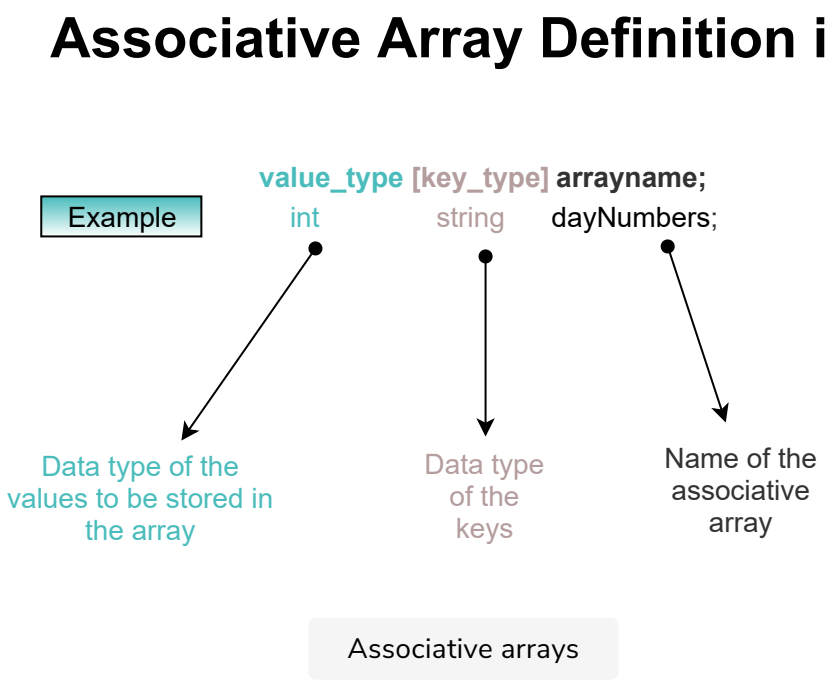
- For plain arrays, index values are not stored at all. Because array elements are stored contiguously in memory, index values are implicitly the relative positions of elements from the beginning of the array.

  On the other hand, associative arrays do store both the keys and the values of elements. Although this difference makes associative arrays use more memory, it also allows them to use *sparse* key values. For example, when there are just two elements to store for keys 0 and 999, an associative array stores just two elements, not 1000 as a plain array has to.

Definition

The syntax of associative arrays is similar to the array syntax. The difference is that it is the type of key that is specified within the square brackets, not the

length of the array. For example, an associative array that maps *day names* of type `string` to *day numbers* of type `int` can be defined like this:



Associative arrays

The `dayNumbers` variable above is an associative array that can be used as a table that provides a mapping from day names to day numbers. In other words, it can be used as the opposite of the `dayNames` array at the beginning of this lesson. We will use the `dayNumbers` associative array in the examples below.

The keys of associative arrays can be of any type, including user-defined struct and class types.

The length of associative arrays cannot be specified when defined. They grow automatically as key-value pairs are added.

> **Note:** An associative array that is defined without any element is null, not empty. This distinction has an important consequence when passing associative arrays to functions. We will cover these concepts in later chapters.

Adding key-value pairs

Using the assignment operator is sufficient to build the association between a key and a value:

```
// associates value 0 with key "Monday"
dayNumbers["Monday"] = 0;

// associates value 1 with key "Tuesday"
dayNumbers["Tuesday"] = 1;
```

The table grows automatically with each association. For example, `dayNumbers` would have two key-value pairs after the operations above. This can be demonstrated by displaying the entire table:

```
writeln(dayNumbers);
```

The output indicates that the values 0 and 1 correspond to keys “Monday” and “Tuesday,” respectively:

```
["Monday":0, "Tuesday":1]
```

There can be only one value per key. For that reason, when we assign a new key-value pair and the key already exists, the table does not grow; instead, the value of the existing key changes:

```
dayNumbers["Tuesday"] = 222;
writeln(dayNumbers);
```

The output:

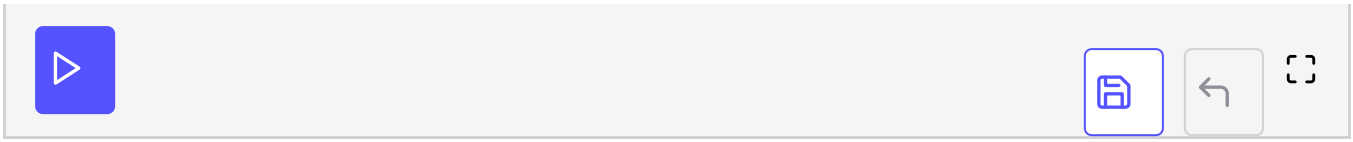
```
["Monday":0, "Tuesday":222]
```

Initialization

Sometimes some of the mappings between the keys and the values are already known at the time of the definition of the associative array. Associative arrays are initialized similar to regular arrays, using a colon to separate each key from its respective value:

```
import std.stdio;

void main(){
// key : value
    int[string] dayNumbers =[ "Monday" : 0, "Tuesday" : 1, "Wednesday" : 2,
                              "Thursday" : 3, "Friday" : 4, "Saturday" : 5,
                              "Sunday" : 6 ];

    writeln(dayNumbers);
```

```
}
```



Associative array initialization

Removing key-value pairs

Key-value pairs can be removed by using `.remove()`:

```
dayNumbers.remove("Tuesday");
writeln(dayNumbers["Tuesday"]); // ← run-time ERROR
```

The first line above removes the key-value pair "Tuesday:1." Since that key is not in the container anymore, the second line would cause an exception to be thrown and the program to be terminated if the exception is not caught. `.clear` removes all elements:

```
dayNumbers.clear; // The associative array becomes empty
```

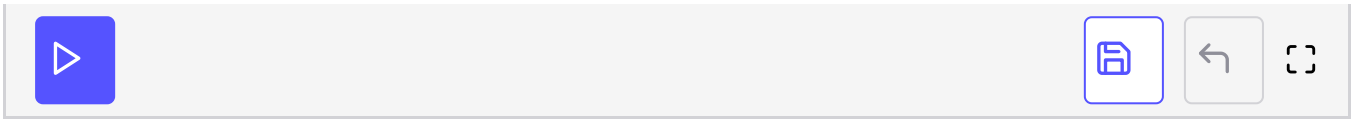
Determining the presence of a key

The `in` operator determines whether a given key exists in the associative array:

```
int[string] colorCodes = [ /* ... */ ];

if ("purple" in colorCodes) {
// key "purple" exists in the table

} else {
// key "purple" does not exist in the table
}
```

Sometimes it makes sense to use a default value if a key does not exist in the associative array. For example, the special value of -1 can be used as the code for colors that are not in `colorCodes`. `.get()` is useful in such cases: it returns the value associated with the specified key if that key exists. Otherwise, it returns the default value. The default value is specified as the second parameter of `.get()`:

```
import std.stdio;

void main(){
    int[string] colorCodes = [ "blue" : 10, "green" : 20 ];
    writeln(colorCodes.get("purple", -1));
}
```



Using .get() with associative arrays

Since the array does not contain a value for the key “purple,” `.get()` returns -1.

Properties

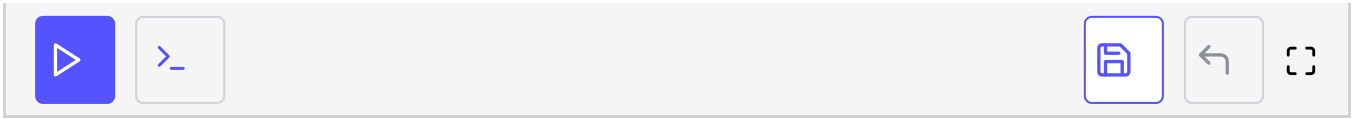
- `.length` returns the number of key-value pairs.
- `.keys` returns a copy of all keys as a dynamic array.
- `.byKey` provides access to the keys without copying them; we will see how `.byKey` is used in `foreach` loops in a later chapter.
- `.values` returns a copy of all values as a dynamic array.
- `.byValue` provides access to the values without copying them.
- `.byKeyValue` provides access to the key-value pairs without copying them.
- `.rehash` may make the lookups in an array more efficient in some cases, such as after inserting a large number of key-value pairs.
- `.sizeof` is the size of the array reference (it has nothing to do with the number of key-value pairs in the table and is the same value for all associative arrays).
- `.get` returns the value if it exists, or the default value otherwise.
- `.remove` removes the specified key and its value from the array.
- `.clear` removes all elements.

Example

Here is a program that prints the Turkish names of colors that are specified in

English:

```
import std.stdio;
import std.string;

void main() {
    string[string] colors = [ "black" : "siyah",
                              "white" : "beyaz",
                              "red"   : "kırmızı",
                              "green" : "yeşil",
                              "blue"  : "mavi" ];

    writefln("I know the Turkish names of these %s colors: %s",
             colors.length, colors.keys);

    write("Please ask me one: ");
    string inEnglish = toLower(strip(readln()));

    if (inEnglish in colors) {
        writefln("\"%s\" is \"%s\" in Turkish.",
                 inEnglish, colors[inEnglish]);

    } else {
        writeln("I don't know that one.");
    }
}
```



Program to tell Turkish name of a color

`strip()` is an inbuilt function that removes leading and trailing characters from the string.

In the next lesson, you will find a coding challenge related to arrays.