Value Types vs Reference Types

In this lesson we will see how D's arrays and slices behave with value type and reference type. Furthermore, we will perform an experiment to apply == operator to different types.

WE'LL COVER THE FOLLOWING

Fixed-length arrays and slices

D’s arrays and slices show different behavior when it comes to value type versus reference type.
As we have already seen above, *slices are reference types*. On the other hand, *fixed-length arrays are value types*. They own their elements, behaving as individual values:

```
import std.stdio;

void main() {
    int[3] array1 = [ 10, 20, 30 ];
    auto array2 = array1; // array2's elements are different from array1's

    array2[0] = 11;

    // First array is not affected:
    assert(array1[0] == 10);
    writeln(array1[0]);
}
```

Effect of = operation on arrays

`array1` is a fixed-length array because its length is specified when it has been

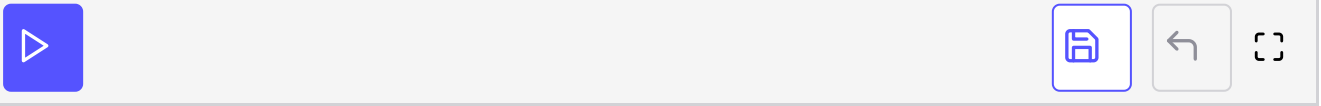
defined. Since `auto` makes the compiler infer the type of `array2`, it is a fixed-length array as well. The values of `array2`'s elements are copied from the values of the elements of `array1`. Each array has its own elements. Modifying an element through one does not affect the other.

Experiment

The following program is an experiment of applying the `==` operator to different types. It applies the operator to both variables and addresses of variables of a certain type. The program produces the following output:

```
import std.stdio;
import std.array;

int moduleVariable = 9;

class MyClass {
    int member;
}

void printHeader() {
    immutable dchar[] header =
        "                      Type of variable" ~
        "                       a == b  &a == &b";

    writeln();
    writeln(header);
    writeln(replicate("=", header.length));
}

void printInfo(const dchar[] label,
               bool valueEquality,
               bool addressEquality) {
    writefln("%55s%9s%9s",
             label, valueEquality, addressEquality);
}

void main() {
    printHeader();

    int number1 = 12;
    int number2 = 12;
    printInfo("variables with equal values (value type)",
              number1 == number2,
              &number1 == &number2);

    int number3 = 3;
    printInfo("variables with different values (value type)",
              number1 == number3,
              &number1 == &number3);

    int[] slice = [ 4 ];
    foreach (i, ref element; slice) {
        printInfo("foreach with 'ref' variable",
                  element == slice[i],
```

```
                &element == &slice[i]);
    }


    foreach (i, element; slice) {
        printInfo("foreach without 'ref' variable",
                  element == slice[i],
                  &element == &slice[i]);
    }

    outParameter(moduleVariable);
    refParameter(moduleVariable);
    inParameter(moduleVariable);

    int[] longSlice = [ 5, 6, 7 ];
    int[] slice1 = longSlice;
    int[] slice2 = slice1;
    printInfo("slices providing access to same elements",
              slice1 == slice2,
              &slice1 == &slice2);

    int[] slice3 = slice1[0 .. $ - 1];
    printInfo("slices providing access to different elements",
              slice1 == slice3,
              &slice1 == &slice3);

    auto variable1 = new MyClass;
    auto variable2 = variable1;
    printInfo(
        "MyClass variables to same object (reference type)",
        variable1 == variable2,
        &variable1 == &variable2);

    auto variable3 = new MyClass;
    printInfo(
        "MyClass variables to different objects (reference type)",
        variable1 == variable3,
        &variable1 == &variable3);
}

void outParameter(out int parameter) {
    printInfo("function with 'out' parameter",
              parameter == moduleVariable,
              &parameter == &moduleVariable);
}

void refParameter(ref int parameter) {
    printInfo("function with 'ref' parameter",
              parameter == moduleVariable,
              &parameter == &moduleVariable);
}

void inParameter(in int parameter) {
    printInfo("function with 'in' parameter",
              parameter == moduleVariable,
              &parameter == &moduleVariable);
}
```



Applying == operator to different types

Applying == operator to different types

Let's look at some of the important points in this code:

- The program makes use of a module variable when comparing different types of function parameters. Module variables are defined at module level outside of all of the functions. They are globally accessible to all of the code in the module.
- The `replicate()` function of the `std.array` module takes an array (the `"="` string above) and repeats it the specified number of times.

Summary

- Variables of value types have their own values and addresses.
- Reference variables do not have their own values or addresses. They are aliases of existing variables.
- Reference type variables have their own addresses but the values that they refer to do not belong to them.
- With reference types, the assignment does not change the value, it changes the value being accessed.
- The variables of reference types may be `null`.

In the next lesson, you will find a quiz on the concepts of value types and reference types.