Value Types

This lesson introduces the concept of value types and discusses the & operator in more detail.

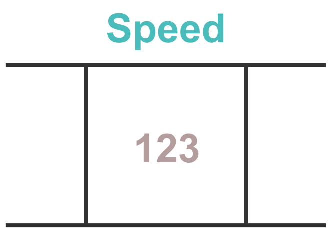
WE'LL COVER THE FOLLOWING

- The use of assert checks below
- Value identity
- Address-of operator `&`

Value types are easy to describe: variables of value types carry values. For example, all of the integer and floating point types are values types. Although not immediately obvious, fixed-length arrays are value types as well.
For example, a variable of type `int` has an integer value:

```
int speed = 123;
```

The number of bytes that the variable `speed` occupies is the size of an `int`. If we visualize the memory as a ribbon going from left to right, we can imagine the variable living on some part of it:

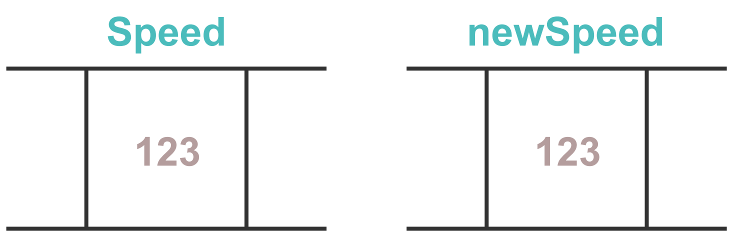


Copy of value type variable

When variables of value types are copied, they get their own values:

```
int newSpeed = speed;
```

The new variable would have a place and a value of its own:

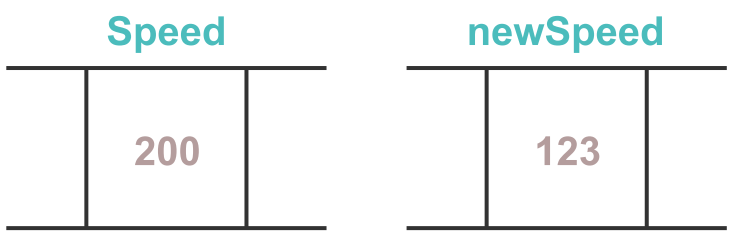


int variable

Naturally, modifications that are made to these variables are independent:

```
speed = 200;
```

The value of the other variable does not change:



No effect on the newSpeed variable

The use of assert checks below

The following examples contain assert checks to indicate that their conditions are true. In other words, these are not checks in the normal sense, rather my way of showing the reader that "this is true".
For example, the check `assert(speed == newSpeed)` below means "speed is equal to newSpeed".

Value identity

As the memory representations above indicate, there are two types of equality that concern variables:

- **Value equality:** The `==` operator that appears in many examples throughout the course compares variables by their values. When two variables are said to be equal in that sense, their values are equal.

- **Value identity:** in the sense of owning separate values, `speed` and `newSpeed` have separate identities. Even when their values are equal, they are different variables.

are different variables.

```
import std.stdio;

void main() {
    int speed = 123;
    int newSpeed = speed;
    assert(speed == newSpeed);
    speed = 200;
    assert(speed != newSpeed);
}
```

Address-of operator `&`

We have been using the `&` operator so far with `readf()`. The `&` operator tells `readf()` where to put the input data i.e., the address of the input in memory.

> **Note:** As we have seen in the Reading from the standard input chapter, `readf()` can be used without explicit pointers as well.

The addresses of variables can be used for other purposes as well. The following code simply prints the addresses of two variables:

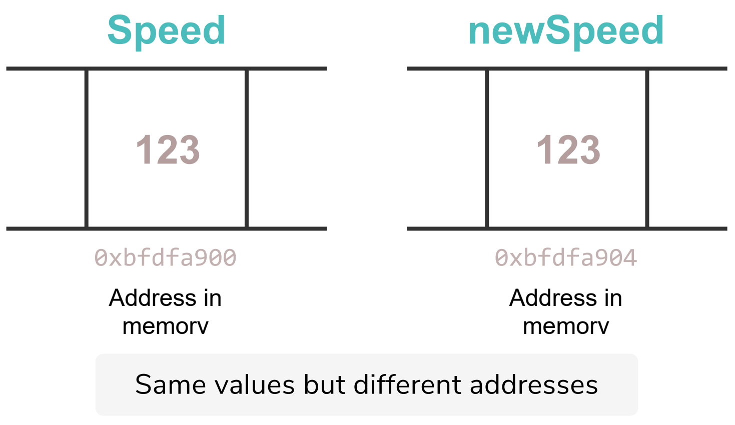
```
import std.stdio;

void main(){
    int speed = 123;
    int newSpeed = speed;

    writeln("speed : ", speed, " address: ", &speed);
    writeln("newSpeed: ", newSpeed, " address: ", &newSpeed);
}
```

& operator

`speed` and `newSpeed` have the same value but their addresses are different.



> **Note:** It is normal for the addresses to have different values every time the program is run. Variables live at parts of memory that happen to be available during that particular execution of the program.

Addresses are normally printed in hexadecimal format. Additionally, if the value of two addresses is 4 bytes apart, it indicates that these two corresponding variables are placed next to each other in memory.

> **Note:** The value of hexadecimal C is 12, so the difference between 8 and 12 is 4.

In the next lesson, we will see the reference variables.