Detecting Overflow and Truncation in Integers

This lesson explains how overflow and truncation can be detected and prevented.

WE'LL COVER THE FOLLOWING

Detecting overflow

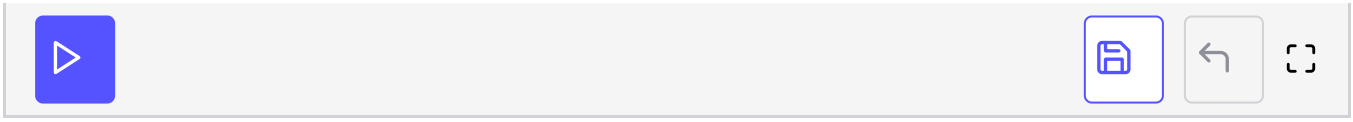
Although detecting overflow uses functions, if conditions and ref parameters, which we have not covered yet, it should be mentioned here that the `core.checkedint` module contains arithmetic functions that detect overflow. Instead of operators like `+` and `-`, this module uses functions: `adds` and `addu` for signed and unsigned addition, `muls` and `mulu` for signed and unsigned multiplication, `subs` and `subu` for signed and unsigned subtraction and `negs` for negation.

Example

Assume that `a` and `b` are two *int* variables, the following code can detect if adding them has caused an overflow:

```
import std.stdio;
import core.checkedint;

void main() {
    // Let's cause overflow for test purposes
    int a = int.max - 1;
    int b = 2;

    // This variable will become 'true' if the addition
```

```
    // This variable will become 'true' if the addition
    // operation inside the 'adds' function overflows:
    bool hasOverflowed = false;
    int result = adds(a, b, hasOverflowed);

    if (hasOverflowed) {
        writeln("Overflow occured");
        // We must not use 'result' because it has overflowed
        // ...

    } else {
        writeln("Overflow did not occur");
        // We can use 'result'
        // ...
    }
}
```



Function to detect overflow

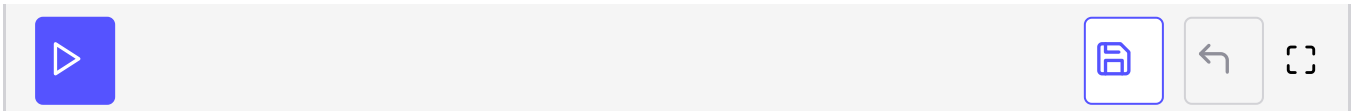
Preventing overflow

If the result of an operation cannot fit in the type of the result, then there is not much that can be done. Sometimes, although the ultimate result would fit in a certain type, the intermediate calculations may overflow and cause incorrect results.

Example

Let's assume that we need to plant one apple tree per 1000 square meters of an area that is 40 by 60 kilometers. How many trees are needed?
When we solve this problem on paper, we see that the result is 40000 times 60000 divided by 1000, which is equal to 2.4 million trees. Let's write a program that executes this calculation:

```
import std.stdio;

void main() {
    int width  = 40000;
    int length = 60000;
    int areaPerTree = 1000;

    int treesNeeded = width * length / areaPerTree;

    writeln("Number of trees needed: ", treesNeeded);
}
```



Overflow example

Not only the output is incorrect, but it is also less than zero! In this case, the intermediate calculation `width * length` overflows and the subsequent calculation of division produces an incorrect result.
One way of avoiding the overflow in this example is to change the order of operations:

```
int treesNeeded = width / areaPerTree * length;
```

The result would now be correct:

```
Number of trees needed: 2400000
```

This method works because all of the steps of the calculation now fit the *int* type.
Please note that this is not a complete solution because this time the intermediate value (`width / areaPerTree`) is prone to truncation, which may affect the result significantly in certain other calculations. Another solution might be to use a floating point type instead of an integer type e.g, float, double, or real.

Preventing truncation

Changing the order of operations may be a solution to truncation as well.

Example

Truncation can be seen by dividing and multiplying a value with the same number. We would expect the result of 10/9*9 to be 10, but it comes out as 9:

```
import std.stdio;

void main() {
    int value1 = 10;
    int value2 = 9;

    writeln(value1 / value2 * value1);
}
```

Truncation example

The result is correct when truncation is avoided by changing the order of operations:

```
writeln(value1 / value2 * value1);
```

The result, in this case, will be:

```
10
```

This, too, is not a complete solution: This time, the intermediate calculation (`value1 / value2`) could be prone to overflow. Using a floating point type may be another solution to truncation in certain calculations.

In the next lesson, we will start looking at floating point types in D language.