Basic Arithmetic Operations on Integers

This lesson digs into the basic arithmetic operations on integers.

WE'LL COVER THE FOLLOWING

- Increment: ++
- Decrement: --
- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Remainder (modulus): %
- Power: ^^

We will take advantage of the `.min` and `.max` properties below, which we have seen in the fundamental types lesson. These properties provide the minimum and maximum values that an integer type can have.

Increment: ++

This operator uses a single operand (usually a variable or an expression) and is written before the name of that variable. It increments the value of that variable by 1:

```
import std.stdio;

void main() {
    int number = 10;
    ++number;
    writeln("New value: ", number);
}
```

Use of ++ operator

As the program's output shows, the value of `number` has been updated.

```
New value: 11
```

The increment operator is the equivalent of using the `+=` (add-and-assign) operator with a value of 1:

```
number += 1; // same as ++number
```

If the result of the increment operation is greater than the maximum value of that type, the result *overflows* and becomes the minimum value. We can see this effect by incrementing a variable that initially has the value `int.max`:

```
import std.stdio;

void main() {
    writeln("minimum int value   : ", int.min);
    writeln("maximum int value   : ", int.max);

    int number = int.max;
    writeln("before the increment: ", number);
    ++number;
    writeln("after the increment : ", number);
}
```

Overflow example

As you can see, the value becomes `int.min` after the increment:

```
minimum int value: -2147483648
maximum int value: 2147483647
before the increment: 2147483647
after the increment: -2147483648
```

This is a very important observation because the value changes from the maximum to the minimum as a result of incrementing and without any warning! This effect is called **overflow**. We will see similar effects with other operations.

Decrement: --

This operator is similar to the increment operator; the difference is that the value is decreased by 1:

```
--number; // the value decreases by 1
```

The decrement operation is the equivalent of using the `-=` (subtract-and-assign) operator with the value of 1:

```
number -= 1; // same as --number
```

Similar to the `++` operator, if the value is the minimum value to begin with, it becomes the maximum value after decrement. This effect is called *overflow* as well.

Addition: +

This operator adds the value of two variables or expressions:

```
import std.stdio;

void main() {
    int first = 12;
    int second = 100;

    writeln("Adding first and second: ", first + second);
    writeln("Adding a constant expression: ", 1000 + second);
}
```

Use of + operator

If the sum of two expressions is greater than the maximum value of that type, it overflows and takes a value that is less than both the expressions:

```
import std.stdio;

void main() {
    // 3 billion each
    uint first = 3000000000;
    uint second = 3000000000;

    writeln("Maximum value of uint: ", uint.max);
    writeln("                First: ", first);
    writeln("               Second: ", second);
    writeln("                  Sum: ", first + second);

    writeln("OVERFLOW! The result is not 6 billion!");
}
```

Overflow example

Subtraction: -

This operator is used with two expressions and gives the difference between the two:

```
import std.stdio;

void main() {
    int first = 10;
    int second = 20;

    writeln(first - second);
    writeln(second - first);
}
```

Use of - operator

It is again surprising if the actual result is less than zero and is stored in an unsigned type. Let’s rewrite the program using the `uint` type:

```
import std.stdio;

void main() {
    uint first = 10;
    uint second = 20;

    writeln("PROBLEM! uint cannot have negative values:");
    writeln(first - second);
    writeln(second - first);
}
```

Error using uint

It is a good guideline to use signed types to represent concepts that may be subtracted.

Multiplication: *

This operator multiplies the values of two expressions. The result is again subject to overflow:

```
import std.stdio;

void main() {
    uint number_1 = 6;
    uint number_2 = 7;

    writeln(number_1 * number_2);
}
```

Use of * operator

Division: /

This operator (/) divides the first expression by the second expression. Since integer types cannot have fractional values, the fractional part of the value is discarded. This effect is called **truncation**. As a result, the following program prints 3, not 3.5:

```
import std.stdio;

void main() {
    writeln(7 / 2);
}
```

Use of / operator

For calculations where fractional parts matter, floating point types must be used instead of integers. We will see floating point types later in this chapter.

Remainder (modulus): %

Remainder (modulus): %

This operator (%) divides the first expression by the second expression and outputs the remainder of the operation:

```
import std.stdio;

void main() {
    writeln(10 % 6);
}
```

Use of % operator

A common application of this operator is to determine whether a value is odd or even. Since the remainder of dividing an even number by 2 is always 0, comparing the result against 0 is sufficient to make that distinction.

Power: ^^

This operator raises the first expression to the power of the second expression. For example, raising 3 to the power of 4 is multiplying 3 by itself 4 times:

```
import std.stdio;

void main() {
    writeln(3 ^^ 4);
}
```

Use of ^^ operator

So far we have covered all the basic arithmetic operations that can be performed on integers in D. In the next lesson, we will look at a few advanced arithmetic operations that can be performed in D.