Advanced Arithmetic Operations on Integers

This lesson is a continuation of the previous lesson, and it explores some advanced arithmetic operations that we can perform with integers in D.

Arithmetic operations with assignment

All of the operators that take two expressions have assignment counterparts. These operators assign the result back to the expression that is on the left-hand side:

```
import std.stdio;

void main() {
    int number = 10;

    number += 20;  // same as number = number + 20; now 30
    writeln(number);

    number -= 5;   // same as number = number - 5;  now 25
    writeln(number);

    number *= 2;   // same as number = number * 2;  now 50
    writeln(number);

    number /= 3;   // same as number = number / 3;  now 16
    writeln(number);

    number %= 7;   // same as number = number % 7;  now  2
    writeln(number);
    number ^^= 6;  // same as number = number ^^ 6; now 64
    writeln(number);
}
```

Assignment counterparts of arithmetic operators

Negation: -

This operator converts the value of the expression from negative to positive or vice versa:

```
import std.stdio;

void main() {
    int number_1 = 1;
    int number_2 = -2;

    writeln(-number_1);
    writeln(-number_2);
}
```

- operator

It is different from subtraction because it takes only one operand. Since unsigned types cannot have negative values, the result of using this operator with unsigned types can be surprising:

```
import std.stdio;

void main() {
    uint number = 1;
    writeln("negation on uint: ", -number);
}
```

The type of `-number` is *uint* as well, which cannot have negative values:

```
negation: 4294967295
```

Plus sign: +

Plus sign: +

This operator has no effect and exists only for consistency with the negation operator. Positive values stay positive, and negative values stay negative:

```
import std.stdio;

void main() {
    int number_1 = 1;
    int number_2 = -2;

    writeln(+number_1);
    writeln(+number_2);
}
```

+ operator

Post-increment: ++

> **Note:** Unless there is a strong reason, always use the regular increment operator (which is sometimes called the **pre-increment operator**).

Contrary to the regular increment operator, the post-increment operator is written after the expression and still increments the value of the expression by 1. The difference is that the post-increment operation produces the old value of the expression. To see this difference, let’s compare it with the regular increment operator:

```
import std.stdio;

void main() {
    int incremented_regularly = 1;
    writeln(++incremented_regularly);      // prints 2
    writeln(incremented_regularly);        // prints 2

    int post_incremented = 1;

    // Gets incremented, but its old value is used:
    writeln(post_incremented++);           // prints 1
    writeln(post_incremented);             // prints 2
}
```

++ operator

The `writeln(post_incremented++);` statement above is the equivalent of the following code:

```
int old_value = post_incremented;
++post_incremented;
writeln(old_value);
```

Post-decrement: --

> **Note:** Unless there is a strong reason, always use the regular decrement operator (which is sometimes called the **pre-decrement operator**).
> This operator behaves the same way as the post-increment operator except that it decrements the value of its operand.

Operator precedence

The operators we've discussed above have all been used in operations on their own with only one or two expressions. However, it is common to combine these operators to form more complex arithmetic expressions:

```
int value = 77;
int result = (((value + 8) * 3) / (value - 1)) % 5;
```

As with logical operators, arithmetic operators also obey operator precedence rules. For example, the * operator has precedence over the + operator. For that reason, when parentheses are not used (e.g. in the `value + 8 * 3` expression), the * operator is evaluated before the + operator. As a result, that expression becomes the equivalent of `value + 24`, which is quite different from `(value + 8) * 3`.
Using parentheses is useful both for ensuring correct results and for communicating the intent of the code to programmers who may work on the code in the future.

Using the concepts covered so far in this chapter, solve the coding challenge given in the next lesson.

given in the next lesson.