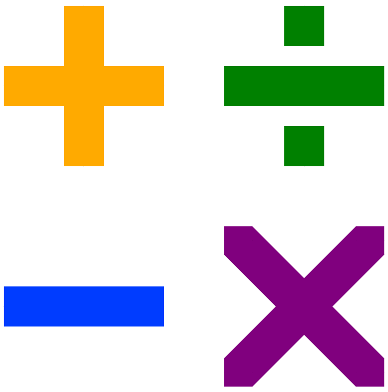
Integers

This lesson explains fundamental arithmetic operations in general and provides background knowledge related to the integer data type.

WE'LL COVER THE FOLLOWING

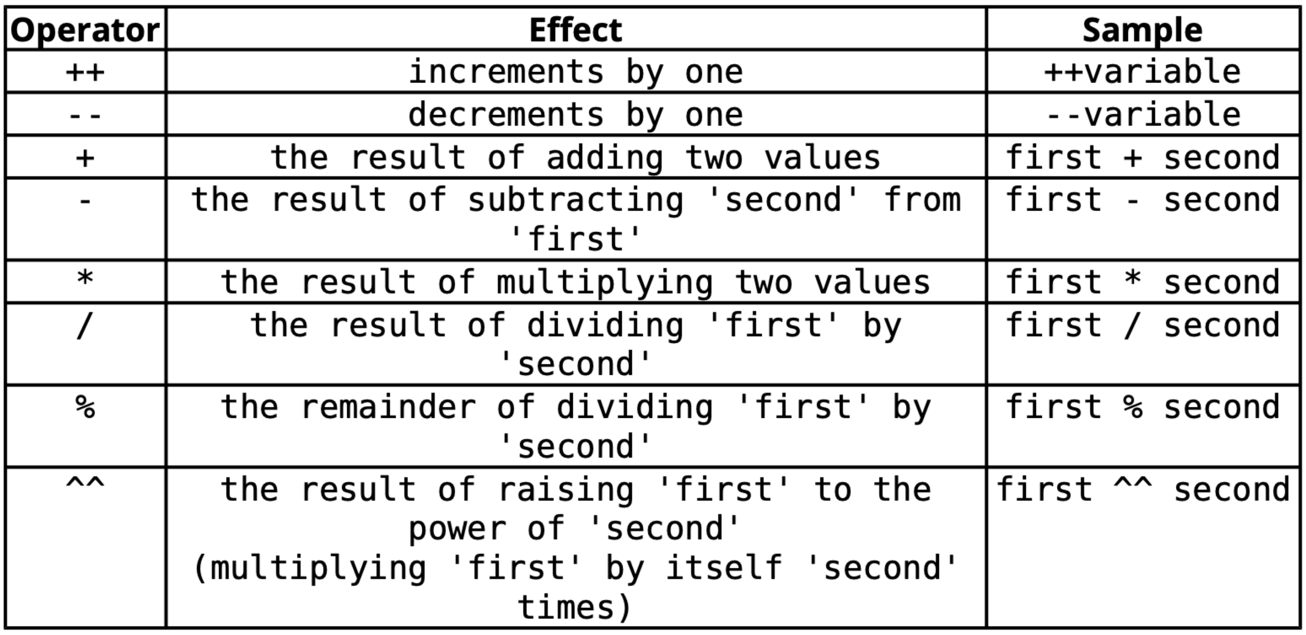
Arithmetic operations

The basic arithmetic operations are addition, multiplication, division and subtraction. Arithmetic operations allow us to write much more useful programs.



Arithmetic operations

Before going further, let's have a look at the summary of arithmetic operations:



Operator	Effect	Sample
++	increments by one	++variable
--	decrements by one	--variable
+	the result of adding two values	first + second
-	the result of subtracting 'second' from 'first'	first - second
*	the result of multiplying two values	first * second
/	the result of dividing 'first' by 'second'	first / second
%	the remainder of dividing 'first' by 'second'	first % second
^^	the result of raising 'first' to the power of 'second' (multiplying 'first' by itself 'second' times)	first ^^ second

Summary of arithmetic operations/operators

Most of those operators have counterparts that have an `=` sign attached: `+=`, `-=`, `*=`, `/=`, `%=`, and `^^=`. These operators assign the result of the operation to the variable on the left-hand side.

```
variable += 10;
```

This expression adds `10` to the value of the `variable` and assigns the result to `variable`. After the operation, the value of the `variable` would be incremented by `10`. It is the equivalent of the following expression:

```
variable = variable + 10;
```

Although arithmetic operations are a part of our daily lives and are actually simple, there are very important concepts that a programmer must be aware of in order to produce correct programs: the *bit length* of a type, *overflow* (wrap) and *truncation*. These concepts are summarized below.

Overflow: Any given type can hold only a range of values depending on the number of bytes used to store it in the memory. If the value to be stored is too big for the variable, we say that the variable **overflows**.
For example, a variable of type ubyte can have values only in the range of 0 to 255. When it is assigned the value 260, the variable overflows, wraps around and its value becomes 4.

> **Note:** Unlike some other languages like C and C++, overflow for signed types is legal in D. It has the same wrap around behavior as exhibited by unsigned types.

Similarly, a variable cannot hold a value that is less than the minimum value of its type.

Truncation: Integer types cannot have values with fractional parts. For example, the value of the *integer* expression 3/2 is 1, not 1.5.

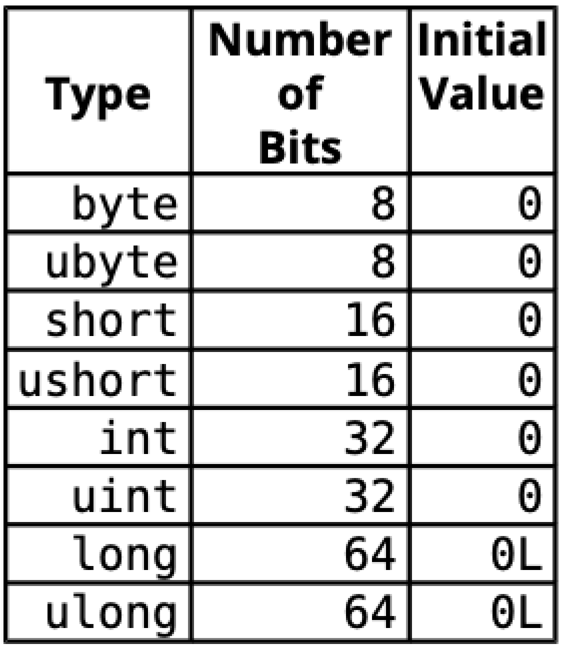
We encounter arithmetic operations daily without many surprises: if a bagel is $1, two bagels are $2; if four sandwiches are $15, one sandwich is $3.75, etc. Unfortunately, things are not as simple with arithmetic operations in computers. If we don't understand how values are stored in a computer, we may be surprised to see that a company's debt is reduced to $1.7 billion when it borrows $3 billion more on top of its existing debt of $3 billion! Or when a box of ice cream serves 4 kids, an arithmetic operation may claim that 2 boxes would be sufficient for 11 kids!

How are integers stored in computers?

Let's first have a look at the types of integers we have in D language.

Integer types

Integer types are the types that can only have whole values like -2, 0, 10, etc. These types cannot have fractional parts, as in 2.5. The integer types that we saw in the fundamental types lesson are:



Type	Number of Bits	Initial Value
byte	8	0
ubyte	8	0
short	16	0
ushort	16	0
int	32	0
uint	32	0
long	64	0L
ulong	64	0L

Integer types

The ‘u’ at the beginning of the type names stands for “unsigned” and indicates that such types cannot have values less than zero.

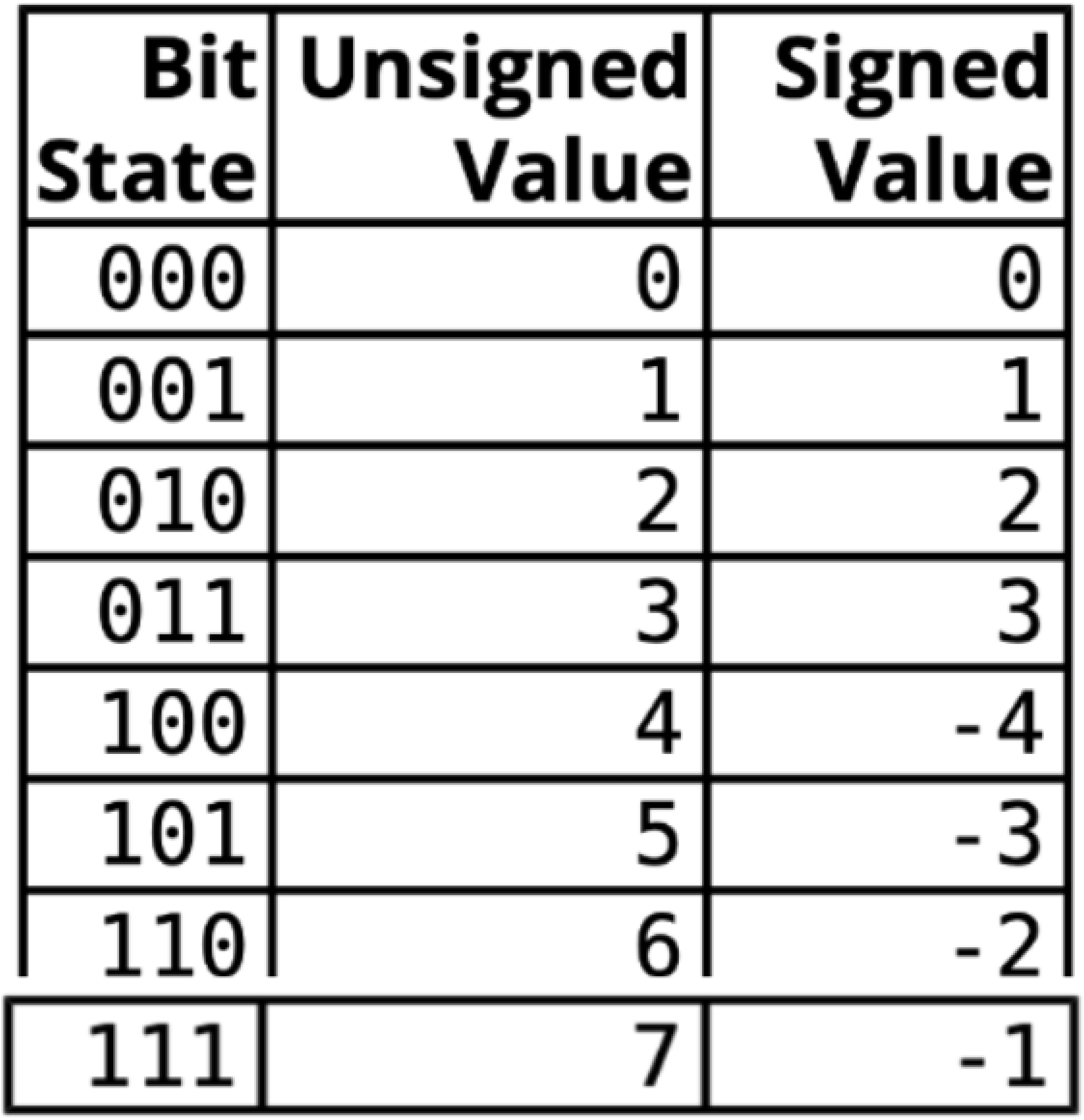
Number of bits of a type

In today’s computer systems, the smallest unit of information is called a **bit**. At the physical level, a bit is represented by electrical signals around certain points in the circuitry of a computer. A bit can be in one of two states that correspond to different voltages in the area that defines that particular bit. These two states are arbitrarily defined to have the values 0 and 1. As a result, a bit can have one of these two values.

As there aren’t many concepts that can be represented by just two states, a bit is not a very useful type. It can only be useful for concepts with two states like heads or tails or whether a light switch is on or off.

If we consider two bits together, the total amount of information that can be represented multiplies. Based on each bit having a value of 0 or 1 individually, there are a total of 4 possible states. Assuming that the left and right digits represent the first and second bit respectively, these states are 00, 01, 10, and 11. Let’s add one more bit to see this effect better; three bits can be in 8 different states: 000, 001, 010, 011, 100, 101, 110, 111. As seen, each added bit doubles the total number of states that can be represented.

The values corresponding to the eight states are defined by convention. The following table shows the signed and unsigned values for the representation of 3 bits:



Bit State	Unsigned Value	Signed Value
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

Values for the signed and unsigned representations of 3 bits

We can construct the following table by adding more bits:

Bits	Number of Distinct Values	D Type	Minimum Value	Maximum Value
1	2			
2	4			
3	8			
4	16			
5	32			
6	64			
7	128			
8	256	byte	-128	127
		ubyte	0	255
...	...			
16	65536	short	-32768	32767
		ushort	0	65535
...	...			
32	4294967296	int	-2147483648	2147483647
		uint	0	4294967295
...	...			
64	18446744073709551616	long	-9223372036854775808	9223372036854775807
		ulong	0	18446744073709551615
...	...			

Integer values having n bits

The table above has many rows skipped, and it indicates the signed and unsigned versions of the D types that have the same number of bits on the same row (e.g., int and uint are both on the 32-bit row).

Choosing a type

If D had a 3-bit data type, it could only have 8 distinct values. It could only represent concepts such as the value of a die or the weekday's number. On the other hand, although *uint* is a very large type, it is insufficient to represent the concept of an identification number for each living person, as its maximum value is less than the world population of 7 billion. *long* and *ulong* would be more than enough to represent many concepts.
As a general rule, you can use `int` for integer values.

Overflow

The fact that types can have a limited range of values may cause unexpected results. For example, adding two `uint` variables with values of 3 billion each should produce 6 billion because that sum is greater than the maximum value that a `uint` variable can hold (about 4 billion). Therefore, the value of the variable overflows. Without any warning, only the difference of 6 and 4 billion gets stored. (A little more accurately, 6 minus 4.3 billion.)

Truncation

Since integers cannot have values with fractional parts, they lose the part after the decimal point. For example, assuming that a box of ice cream serves 4 kids, 11 kids would actually need 2.75 boxes, the fractional part of this value cannot be stored in an integer type, so it is truncated and the value becomes 2.

Limited techniques to help reduce the risk of overflow and truncation will be shown later in the chapter.

In the next lesson, we will see programs for applying arithmetic operations on integers