Floating Point Types

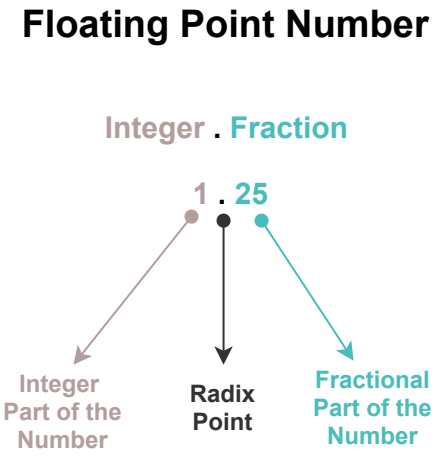
This lesson digs into the types of floating points and their properties.

WE'LL COVER THE FOLLOWING

Earlier in this chapter, we have seen that, despite their ease of use, arithmetic operations on integers are prone to programming errors due to overflow and truncation. We have also seen that integer type variables cannot have values with fractional parts, as in 1.25.

Floating point types

Floating point types are designed to support fractional parts. The “point” in their name comes from the *radix point*, which separates the integer part from the fractional part, and “floating” refers to the detail in how these types are implemented. The decimal point floats left and right as the number of digits is not fixed before and after the decimal point.

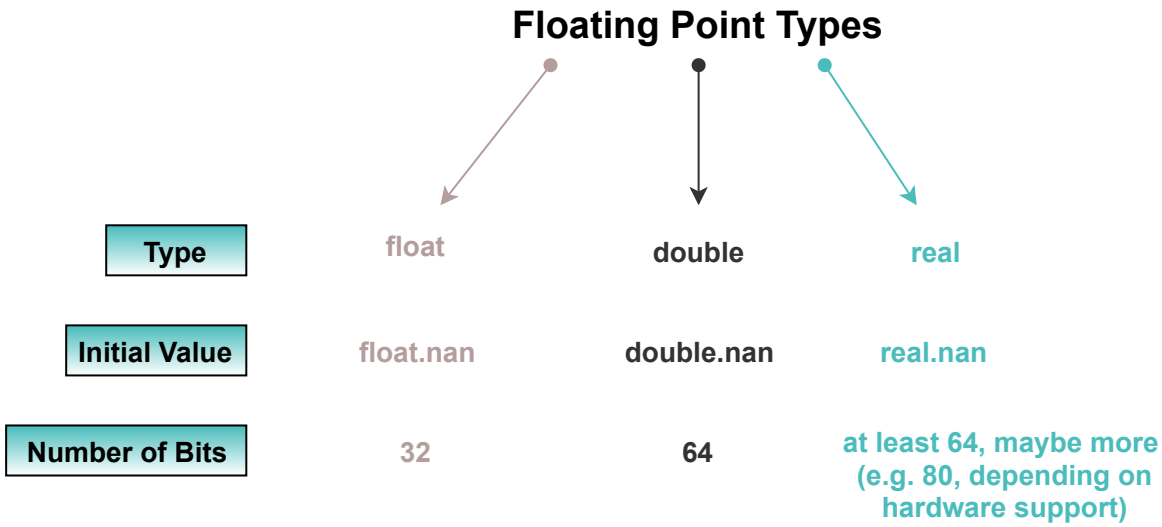


Example of floating point number

Here are some interesting aspects of floating point types:

- Adding 0.001 a thousand times is not the same as adding 1.
- Using the logical operators `==` and `!=` with floating point types produces incorrect results in most cases.
- The initial value of floating point types is `.nan`, not 0. `.nan` may not be used in expressions in any meaningful way. When used in comparison operations, `.nan` is neither less than nor greater than any value.
- There are two overflow values: positive .infinity and negative .infinity.

Although floating point types are more useful than integers in some cases, they have peculiarities that every programmer must know. Compared to integers, they are very good at avoiding truncation because their main purpose is to support fractional values. Like other types, floating point types are also represented using a fixed number of bits and are prone to overflow. However, the range of values that are supported using floating point types is much larger than that of integers. Additionally, instead of being silent in the case of overflow, they get the special values of positive and negative infinity. As a reminder, in D, the floating point types are the following:



Floating point types in D

Properties of floating point type

Floating point types have more properties than other types:

- `.stringof` is the name of the type.

- `.sizeof` is the length of the type in terms of bytes (In order to determine the bit count, this value must be multiplied by 8, the number of bits in a byte.).

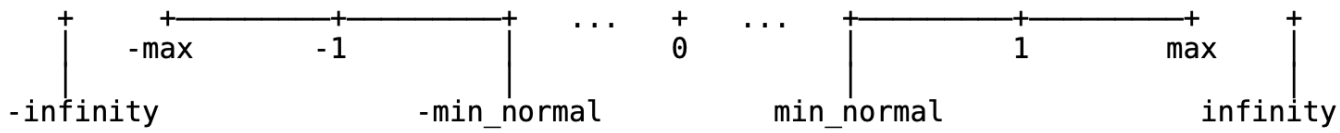
- `.max` is short for "maximum" and is the maximum value that the type can store. There is no separate `.min` property for floating types; the negative of `.max` is the minimum value that the type can have. For example, the minimum value of double is `-double.max`.

- `.min_normal` is the smallest positive value that this type can represent with its normal precision. (Precision is the number of digits that is used when specifying a value, more on it in the next lesson) . The type can represent smaller values than `.min_normal`, but those values cannot be as precise as other values of the type and are generally slower to compute. The condition of a floating point value being between `- .min_normal` and `+ .min_normal` (excluding 0) is called **underflow**.

- `.dig` is short for "digits" and specifies the number of digits that signify the precision of the type.

- `.infinity` is the special value used to denote overflow.

Other properties of floating point types are used less commonly. You can see all of them at properties for floating point types at dlang.org.
The properties of floating point types and their relations can be shown on a number line, like the following:



Properties of floating point types and their relations

Other than the two special infinity values, the line above is to scale: the number of values that can be represented between `min_normal` and **1** is equal to the number of values that can be represented between **1** and *max*. This means that the precision of the fractional parts of the values between `min_normal` and **1** are very high (The same is true for the negative side as well.).

well.):

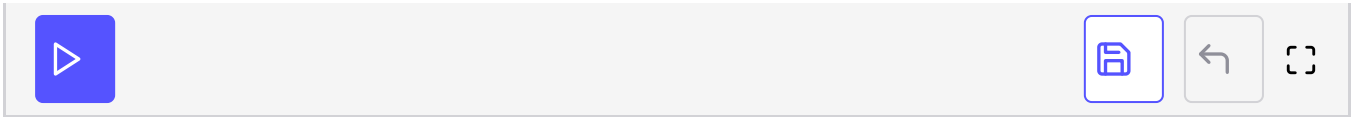
The floating point format is apparent in the output of the following program, that displays the properties of the three floating point types:

```
import std.stdio;

void main() {
    writeln("Type                    : ", float.stringof);
    writeln("Precision               : ", float.dig);
    writeln("Minimum normalized value: ", float.min_normal);
    writeln("Minimum value           : ", -float.max);
    writeln("Maximum value           : ", float.max);
    writeln();

    writeln("Type                    : ", double.stringof);
    writeln("Precision               : ", double.dig);
    writeln("Minimum normalized value: ", double.min_normal);
    writeln("Minimum value           : ", -double.max);
    writeln("Maximum value           : ", double.max);
    writeln();

    writeln("Type                    : ", real.stringof);
    writeln("Precision               : ", real.dig);
    writeln("Minimum normalized value: ", real.min_normal);
    writeln("Minimum value           : ", -real.max);
    writeln("Maximum value           : ", real.max);
}
```



Properties of floating points

`.nan`

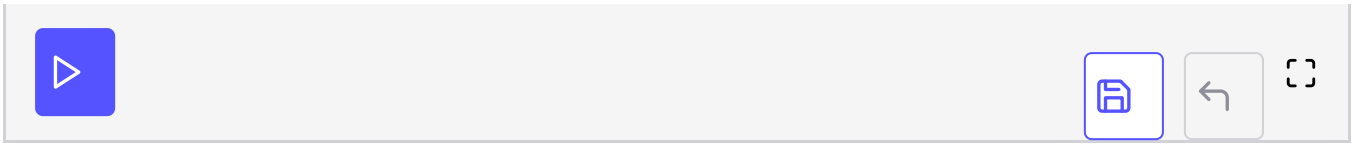
We have already seen that this is the default value of floating point variables. `.nan` may appear as a result of meaningless floating point expressions as well. For example, the floating point expressions in the following program all produce `double.nan`:

```
import std.stdio;

void main() {
    double zero = 0;
    double infinity = double.infinity;

    writeln("Any expression with nan: ", double.nan + 1);
    writeln("zero / zero            : ", zero / zero);
    writeln("zero * infinity        : ", zero * infinity);
    writeln("infinity / infinity    : ", infinity / infinity);
    writeln("infinity - infinity    : ", infinity - infinity);
```

```
}
```



```
.nan
```

`.nan` is not just useful because it indicates an uninitialized value, but also because it is propagated through computations, making error detection easier.

Specifying floating point values

Floating point values can be built from integer values without a decimal point, like 123, or created directly with a decimal point, like 123.0.

Floating point values can also be specified with the special floating point syntax, as in 1.23e+4. The *e+* part of that syntax can be read as *"times 10 to the power of."* According to that reading, the previous value is "1.23 times 10 to the power of 4," which is the same as "1.23 times 10^4" which in turn is the same as 1.23x10000 being equal to 12300.

If the value after *e* is negative, as in 5.67e-3, then it is read as *"divided by 10 to the power of"*. Accordingly, this example is "5.67 divided by 10^3", which in turn is the same as 5.67/1000, which is equal to 0.00567.

> **Note:** Although `double` and `real` have more precision than `float`, `writeln` prints all floating point values with 6 digits of precision. Precision is explained in the next lesson.

Observations

As you will remember from the earlier lessons of this chapter, the maximum value of `ulong` has 20 digits: 18,446,744,073,709,551,616. That value looks small when compared to even the smallest floating point type: `float` can have values up to the 1038 range, e.g. 340,282,000,000,000,000,000,000,000,000,000,000. The maximum value of `real` is in the range 104932, which can have a value with more than 4900 digits!

As another observation, let's look at the minimum value that `double` can represent with 15-digit precision:

`0.000...(there are 300 more zeroes here)...0000222507385850720`

Minimum value that double can represent with 15-digit precision

---

In the next lesson, we will see overflow and truncation in floating point types.