Floating Point Types Usage and Limitations

In this lesson, you will learn how to choose an appropriate type of the floating point and some limitations of these types.

WE'LL COVER THE FOLLOWING

Which floating type to use

Unless there is a specific reason to choose otherwise, you can choose `double` for floating point values. `float` has lower precision than `double` and `real`, but it takes less number of bits and can be useful when we have limited memory. On the other hand, since the precision of `real` is higher than `double` on some hardware, it would be preferable for high precision calculations.

Limitations

Let's discuss a few limitations that we have with the floating point types.

Cannot represent all values

It is impossible to represent some values in our daily life. In the decimal system that we use daily the digits before the decimal point represent ones, tens, hundreds, etc., and the digits after the decimal point represent tenths, hundredths, thousandths, etc. If a value is created from a combination of

these values, it can be represented exactly. This is not true for recurring decimal values.

Example #

As the value 0.23 consists of 2 tenths and 3 hundredths, it is represented exactly. On the other hand, the value 1/3 cannot be exactly represented in the decimal system because the number of digits is always insufficient, no matter how many are specified: 0.33333...

For such cases, the floating point types cannot represent the value beyond a certain precision because they have a limited number of bits.

Difference with the binary system

The difference between the decimal system and the binary system that computers use is that the digits before the decimal point are ones, twos, fours, etc., and the digits after the decimal point are halves, quarters, eighths, etc. Only the values that are exact combinations of those digits can be represented accurately.

A value that cannot be represented exactly in the binary system used by computers is 0.1, as in 10 cents. Although this value can be represented exactly in the decimal system, its binary representation never ends and continuously repeats four digits: 0.0001100110011...

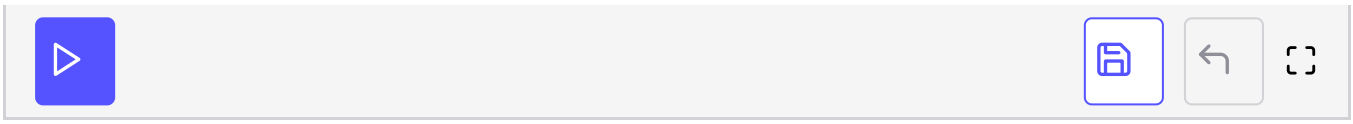
> **Note:** Observe that the value is written in a binary system, not decimal.

It is always inaccurate at some level depending on the precision of the floating point type that is used.

The following program demonstrates this problem. The value of a variable is being incremented by 0.001 a thousand times in a loop. Surprisingly, the result is not 1:

```
import std.stdio;

void main() {
    float result = 0;

    // Adding 0.001 for a thousand times:
    int counter = 1;
    while (counter <= 10) {
        result += 0.1;
        ++counter;
```

```
    }

    if (result == 1) {

        writeln("As expected: ", result);

    } else {
        writeln("DIFFERENT: ", result);
    }
}
```



Limited values problem demonstration

Because 0.001 cannot be represented exactly, that inaccuracy affects the result at every iteration:

```
DIFFERENT: 0.999991
```

> **Note:** The variable `counter` above is a loop counter. We will see loops in a later chapter.

Unorderedness

The same comparison operators that we have covered with integers are used with floating point types as well. However, since the special value `.nan` represents invalid floating point values, comparing `.nan` to other values is not meaningful. For example, it does not make sense to ask whether `.nan` or 1 is greater.
For that reason, floating point values introduce another comparison concept: *unorderedness*. Being **unordered** means that at least one of the values is `.nan`.

The table below lists all the floating point comparison operators. All of them are binary operators (meaning that they take two operands) and are used in *left* == *right*. The columns that contain `false` and `true` are the results of the comparison operations.

The last column indicates whether the operation is meaningful if one of the operands is `.nan`.

Example

Even though the result of the expression `1.2 < real.nan` is *false*, that result is meaningless because one of the operands is `real.nan`. The result of the reverse comparison `real.nan < 1.2` would produce *false* as well. The abbreviation **lhs** stands for left-hand side, indicating the expression on the left-hand side of each operator.

Operator	Meaning	If lhs is greater	If lhs is less	If both are equal	If at least one is .nan	Meaningful with .nan
==	is equal to	false	false	true	false	yes
!=	is not equal to	true	true	false	true	yes
>	is greater than	true	false	false	false	no
>=	is greater than or equal to	true	false	true	false	no
<	is less than	false	true	false	false	no
<=	is less than or equal to	false	true	true	false	no

Unorderedness

Although meaningful to use with `.nan`, the `==` operator always produces false when used with a `.nan` value. This is the case even when both values are `.nan`:

```
import std.stdio;

void main() {
    if (double.nan == double.nan) {
        writeln("equal");

    } else {
        writeln("not equal");
    }
}
```

.nan value comparison

Although one would expect `double.nan` to be equal to itself, the result of the comparison is false.

`IsNaN()` for `.nan` equality comparison

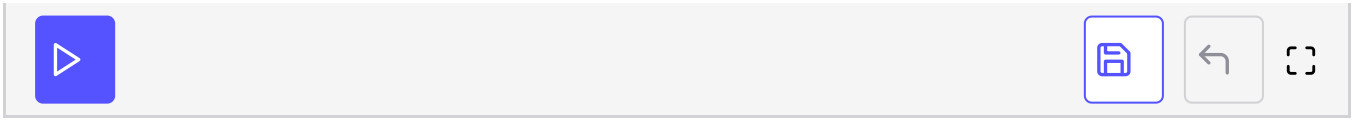
As we have seen above, it is not possible to use the `==` operator to determine whether the value of a floating point variable is `.nan`:

```
if (variable == double.nan) { // ← WRONG
    // ...
```

```
}
```

The `isNaN()` function from the `std.math` module is for determining whether a value is `.nan`:

```
import std.math;
import std.stdio;

void main() {

    double variable;

    writeln(isNaN(variable));
}
```



Similarly, to determine whether a value is not `.nan`, one must use `!isNaN()` because the `!=` operator would always produce true.

In the next lesson, there is a coding challenge related to floating point types.