Parameter Qualifiers: in, out, const and immutable

This lesson first acquaints you with the term parameter qualifier and then explains in, out, const and immutable parameter qualifiers.

WE'LL COVER THE FOLLOWING

Parameter qualifiers

Parameters are passed to functions according to the general rules described below:

- Value types are copied, after which the original variable and the copy are independent.
- Reference types are copied as well, but both the original reference and the parameter provide access to the same variable.

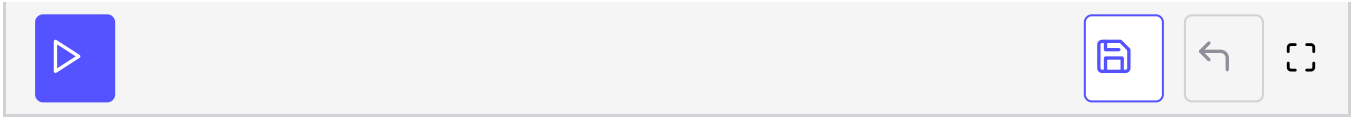
Those are the default rules that are applied when parameter definitions have no qualifiers. The following qualifiers change the way parameters are passed and what operations are allowed on them.

`in`

We have seen that functions can produce values and can have side effects. The `in` keyword specifies that the parameter is going be used only as input:

```
import std.stdio;
```

```
double weightedTotal(in double currentTotal,
                     in double weight,
                     in double addend) {

    return currentTotal + (weight * addend);
}

void main() {
    writeln(weightedTotal(1.23, 4.56, 7.89));
}
```



Use of in keyword

Like `const`, `in` parameters cannot be modified:

```
void foo(in int value) {
    value = 1; // ← compilation ERROR
}
```

`out`

We know that functions return what they produce as their return values. The fact that there is only one return value is sometimes limiting as some functions may need to produce more than one result.

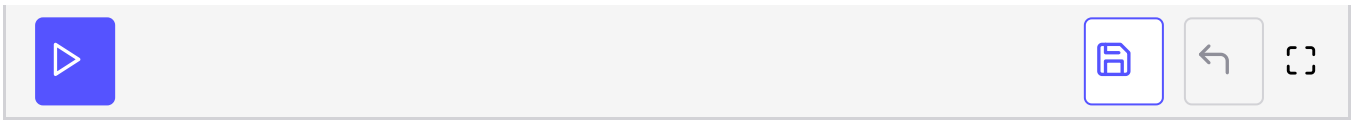
The `out` keyword makes it possible for functions to return results through their parameters. When `out` parameters are modified within the function, those modifications affect the original variable that has been passed to the function. In a sense, the assigned value goes out of the function through the `out` parameter.

> **Note:** It is possible to return more than one result by defining the return type as a tuple or a `struct`.

Let's have a look at a function that divides two numbers and produces both the quotient and the remainder. The `return` value is used for the quotient and the remainder is returned through the `out` parameter:

```
import std.stdio;

int divide(int dividend, int divisor, out int remainder) {
```

```
    remainder = dividend % divisor;
    return dividend / divisor;
}


void main() {
    int remainder;
    int result = divide(7, 3, remainder);

    writeln("result: ", result, ", remainder: ", remainder);
}
```

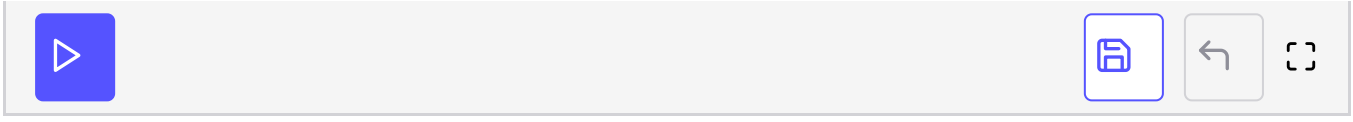


Use of out keyword

Modifying the `remainder` parameter of the function modifies the `remainder` variable in `main()` (their names need not be the same).

Regardless of their values at the time of the function's call, `out` parameters are first assigned the `.init` value of their types automatically inside the function:

```
import std.stdio;

void foo(out int parameter) {
    writeln("After entering the function      : ", parameter);
}

void main() {
    int variable = 100;

    writeln("Before calling the function      : ", variable);
    foo(variable);
    writeln("After returning from the function: ", variable);
}
```



Working with the out parameter

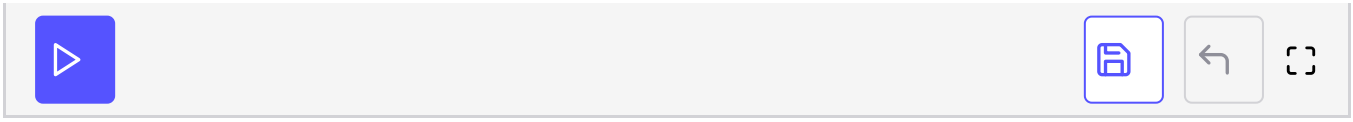
As this demonstrates, `out` parameters cannot pass values into functions; they are strictly for passing values out of functions.
We will see in later chapters that returning tuple or struct types are better alternatives to `out` parameters.

`const`

As we saw earlier, `const` guarantees that the parameter will not be modified

As we saw earlier, `const` guarantees that the parameter will not be modified inside the function. It is helpful for the programmers to know that certain variables will not be changed by a function. `const` also makes functions more useful by allowing `const`, `immutable` and mutable variables to be passed through that parameter:

```
import std.stdio;

dchar lastLetter(const dchar[] str) {
    return str[$ - 1];
}

void main() {
    writeln(lastLetter("constant"));
}
```

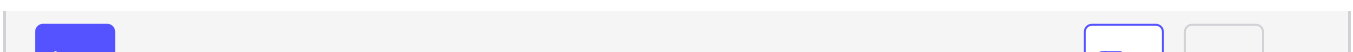


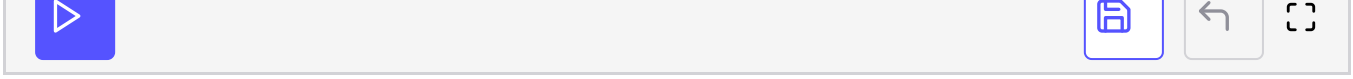
Use of const keyword

`immutable`

As we saw earlier, `immutable` makes functions require that certain variables must be `immutable` types. Because of such a requirement, the following function can only be called with strings with `immutable` elements (e.g. string literals):

```
import std.stdio;

dchar[] mix(immutable dchar[] first,
            immutable dchar[] second) {
    dchar[] result;
    int i;

    for (i = 0; (i < first.length) && (i < second.length); ++i) {
        result ~= first[i];
        result ~= second[i];
    }

    result ~= first[i..$];
    result ~= second[i..$];

    return result;
}

void main() {
    writeln(mix("HELLO", "world"));
}
```





Use of immutable keyword

Since it enforces a constraint on the parameter, `immutable` parameters should be used only when immutability is required. Otherwise, in general, `const` is more useful because it accepts `immutable`, `const` and mutable variables.

In the next lesson, we will see `ref`, `auto ref` and `inout` parameter qualifiers.