Parameter Qualifiers: ref, auto ref and inout

In this lesson, we will discuss three more parameter qualifiers: ref, auto ref and inout

WE'LL COVER THE FOLLOWING

`ref`

This keyword allows passing a variable by reference even though it would normally be passed as a copy (i.e. by value).

For the `reduceEnergy()` function that we saw earlier, to modify the original variable, it must take its parameter as `ref`:

```
import std.stdio;

void reduceEnergy(ref double e) {
    e /= 4;
}

void main() {
    double energy = 100;

    reduceEnergy(energy);
    writeln("New energy: ", energy);
}
```

Use of ref keyword

This time, the modification that is made to the parameter changes the original variable in `main()`.

As can be seen, `ref` parameters can be used both as input and output. `ref`

As can be seen, `ref` parameters can be used both as input and output. `ref`

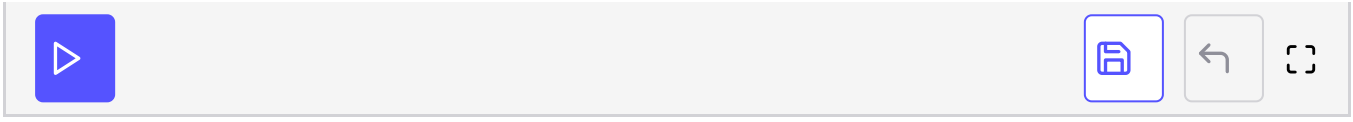
parameters can also be thought of as aliases of the original variables. The function parameter `e` above is an alias of the variable `energy` in `main()`.

Similar to `out` parameters, `ref` parameters allow functions to have side effects as well. In fact, `reduceEnergy()` does not return a value; it only causes a side effect through its single parameter.

The programming style called functional programming favors return values over side effects, so much so that some functional programming languages do not allow side effects at all. This is because functions that produce results purely through their return values are easier to understand, implement, and maintain.

The same function can be written in a functional programming style by returning the result, instead of causing a side effect. The parts of the program that changed are highlighted:

```
import std.stdio;

double reducedEnergy(double energy) {
    return energy / 4;
}

void main() {
    double energy = 100;

    energy = reducedEnergy(energy);
    writeln("New energy: ", energy);
}
```



Functional programming style

Note the change in the name of the function as well. Now, it is a noun as opposed to a verb.

`auto ref`

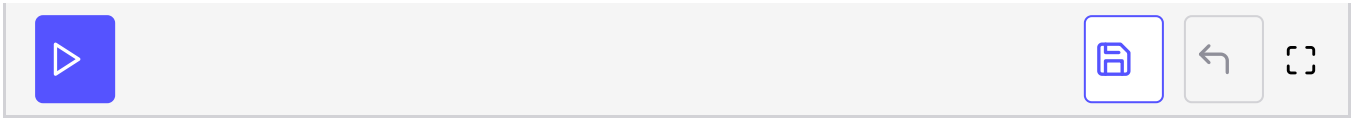
This qualifier can only be used with templates. As we will see later in this chapter, an `auto ref` parameter takes `lvalues` by reference and `rvalues` by copy.

`inout`

Despite its name consisting of *in* and *out*, this keyword does not mean input and output; we have already seen that input and output are achieved by the `ref` keyword.

`inout` carries the mutability of the parameter on to the return type. If the parameter is `const`, `immutable` or mutable; then the return value is also `const`, `immutable` or mutable; respectively.
To see how `inout` helps in programs, let's look at a function that returns a slice to the inner elements of its parameter:

```
import std.stdio;

int[] inner(int[] slice) {
    if (slice.length) {
        --slice.length;                // trim from the end

        if (slice.length) {
            slice = slice[1 .. $];     // trim from the beginning
        }
    }

    return slice;
}

void main() {
    int[] numbers = [ 5, 6, 7, 8, 9 ];
    writeln(inner(numbers));
}
```



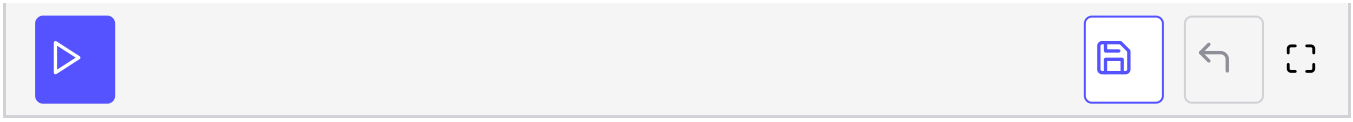
Use of inout keyword

According to what we have established so far, in order for the function to be more useful, its parameter should be `const(int)[]` because the elements are not being modified inside the function.

> **Note:** There is no harm in modifying the parameter slice itself, as it is a copy of the original variable.

However, defining the function that way would cause a compilation error:

```
import std.stdio;


int[] inner(const(int)[] slice) {
    if (slice.length) {
        --slice.length;               // trim from the end

        if (slice.length) {
            slice = slice[1 .. $];    // trim from the beginning
        }
    }

    return slice;
}

void main() {
    int[] numbers = [ 5, 6, 7, 8, 9 ];
    writeln(inner(numbers));
}
```



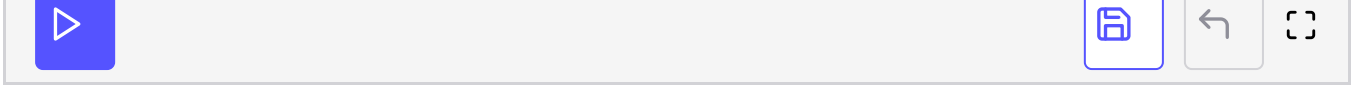
Use of inout keyword

The compilation error indicates that a slice of `const(int)` cannot be returned as a slice of mutable `int`:

```
Error: cannot implicitly convert expression (slice) of type const(int)[] t
o int[]
```

One may think that specifying the return type as `const(int)[]` would be the solution:

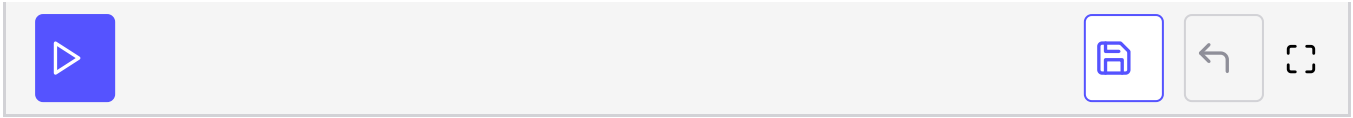
```
import std.stdio;

const(int)[] inner(const(int)[] slice) {
    if (slice.length) {
        --slice.length;               // trim from the end

        if (slice.length) {
            slice = slice[1 .. $];    // trim from the beginning
        }
    }

    return slice;
}

void main() {
    int[] numbers = [ 5, 6, 7, 8, 9 ];
    writeln(inner(numbers));
}
```



Use of inout keyword

Although the code now compiles, it brings a limitation: even when the function is called with a slice of mutable elements, this time the returned slice ends up consisting of `const` elements. To see how limiting this is, let's look at the following code, which tries to modify the inner elements of a slice:

```
import std.stdio;

const(int)[] inner(const(int)[] slice) {
    if (slice.length) {
        --slice.length;                // trim from the end

        if (slice.length) {
            slice = slice[1 .. $];     // trim from the beginning
        }
    }

    return slice;
}

void main() {
    int[] numbers = [ 5, 6, 7, 8, 9 ];
    int[] middle = inner(numbers); // ← compilation ERROR
    middle[] *= 10;
}
```



Use of inout keyword

The returned slice of type `const(int)[]` cannot be assigned to a slice of type `int[]`, resulting in an error:

```
Error: cannot implicitly convert expression (inner(numbers)) of type const
(int)[] to int[]
```

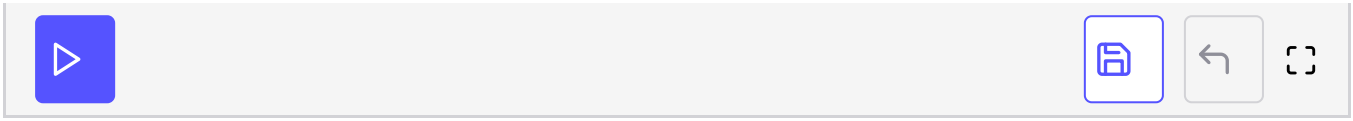
However, since we started with a slice of mutable elements, this limitation is artificial and unfortunate. `inout` solves this mutability problem between parameters and return values. It is specified on both the parameter and the return type and carries the mutability of the former to the latter:

```
inout(int)[] inner(inout(int)[] slice) {
    // ...
```

```
    return slice;
}
```

With that change, the same function can now be called with `const`, `immutable`, and mutable slices:

```
import std.stdio;

inout(int)[] inner(inout(int)[] slice) {
    if (slice.length) {
        --slice.length;                 // trim from the end

        if (slice.length) {
            slice = slice[1 .. $];      // trim from the beginning
        }
    }

    return slice;
}

void main() {
    int[] numbers = [ 5, 6, 7, 8, 9 ];
    // The return type is a slice of mutable elements
    int[] middle = inner(numbers);
    middle[] *= 10;
    writeln(middle);

    immutable int[] numbers2 = [ 10, 11, 12 ];
    // The return type is a slice of immutable elements
    immutable int[] middle2 = inner(numbers2);
     writeln(middle2);

    const int[] numbers3 = [ 13, 14, 15, 16 ];
    // The return type is a slice of const elements
    const int[] middle3 = inner(numbers3);
    writeln(middle3);
}
```



Use of inout keyword

In the next lesson, we will discuss three more parameter qualifiers which are: `lazy`, `scope` and `shared`.