Should a Parameter be const or immutable?

This lesson explains when a parameter should be const and when it should be immutable.

Our discussion so far shows that because they are more flexible, `const` parameters should be preferred over `immutable` parameters. This is not always true.
`const` erases the information about whether the original variable was mutable or immutable. This information is hidden even from the compiler.

A consequence of this fact is that `const` parameters cannot be passed as arguments to functions that take `immutable` parameters. For example, `foo()` below cannot pass its `const` parameter to `bar()`:

```
/* NOTE: This program is expected to fail compilation. */

void main() {
    /* The original variable is immutable */
    immutable int[] slice = [ 10, 20, 30, 40 ];
    foo(slice);
}

/* A function that takes its parameter as const, in order to
 * be more useful. */
void foo(const int[] slice) {
    bar(slice);    // ← compilation ERROR
}

/* A function that takes its parameter as immutable, for a
 * plausible reason. */
void bar(immutable int[] slice) {
    // ...
}
```

Compilation error while passing const parameter

`bar()` requires that the parameter be `immutable`. However, it is not known (in general) whether the original variable that `foo()`'s const parameter was `immutable` or not.

`immutable` or not.

> **Note:** It is clear in the code above that the original variable in `main()` is `immutable`. However, the compiler compiles functions individually, without considering the place that the function is called from. To the compiler, the slice parameter of `foo()` may refer to a mutable variable or an immutable one.

A solution would be to call `bar()` with an immutable copy of the parameter:

```
/* NOTE: This program is expected to fail compilation. */

void main() {
    /* The original variable is immutable */
    immutable int[] slice = [ 10, 20, 30, 40 ];
    foo(slice);
}

/* A function that takes its parameter as const, in order to
 * be more useful. */
void foo(const int[] slice) {
    bar(slice.idup);
}

/* A function that takes its parameter as immutable, for a
 * plausible reason. */
void bar(immutable int[] slice) {
    // ...
}
```

Compilation error while passing const parameter

Although that is a sensible solution, it does incur the cost of copying the slice and its contents, which would be wasteful in the case where a original variable was `immutable` to begin with.

After this analysis, it should be clear that always declaring parameters as `const` is not the best approach in every situation. After all, if `foo()`'s parameter had been defined as `immutable`, there would be no need to copy it before calling `bar()`:

```
void foo(immutable int[] slice) {  // This time immutable
    bar(slice);      // Copying is not needed anymore
```

```
   bar(slice);    // Copying is not needed anymore
}
```

```
/* NOTE: This program is expected to fail compilation. */

void main() {
    /* The original variable is immutable */
    immutable int[] slice = [ 10, 20, 30, 40 ];
    foo(slice);
}

/* A function that takes its parameter as const, in order to
 * be more useful. */
void foo(immutable int[] slice) {  // This time immutable
   bar(slice);    // Copying is not needed anymore
}

/* A function that takes its parameter as immutable, for a
 * plausible reason. */
void bar(immutable int[] slice) {
    // ...
}
```

Compilation error while passing const parameter

Although the code compiles, defining the parameter as `immutable` has a similar cost: this time an `immutable` copy of the original variable is needed when calling `foo()`, if that variable was not `immutable` to begin with:

```
foo(mutableSlice.idup);
```

Templates can help. Although it is not expected from you to fully understand the following function at this point in the course, we will present it as a solution to this problem. The following function template `foo()` can be called both with mutable and immutable variables. The parameter would be copied only if the original variable was mutable; no copying would take place if it was immutable:

```
import std.conv;
// ...
/* Because it is a template, foo() can be called with both mutable * and i
mmutable variables. */
void foo(T)(T[] slice) {
/* 'to()' does not make a copy if the original variable is
* already immutable. */
  bar(to!(immutable T[])(slice));
  bar(to!(immutable T[])(slice));
}
```

In the next lesson, we will see the immutability of the slice versus the elements and how to use immutability in general.