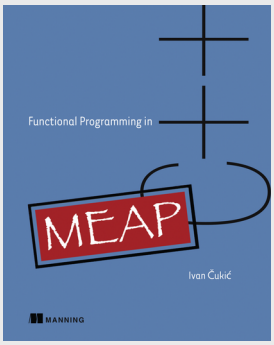

Functions taking functions

By Ivan Čukić

This article has been excerpted from Functional Programming in C++.
Save 37% with code **fcccukic**.

The main feature of all functional programming languages is that functions can be treated like ordinary values. They can be saved into variables, put into collections and structures, passed to other functions as arguments, and also returned from other functions as results.

Functions that take other functions as arguments, or that return new functions are called higher-order functions. Higher-order functions is probably the most important concept in functional programming. As you might know, programs can be made more concise and efficient by describing what the program should do on a higher level, with the help of standard algorithms, instead of implementing everything by hand. Higher-order functions are indispensable for that. They allow us to define abstract behaviours and more complex control structures than those provided by the C++ programming language.

Let's illustrate this with an example. Imagine that we have a group of people, and that we need a list of the names of all females in that group (figure 1).

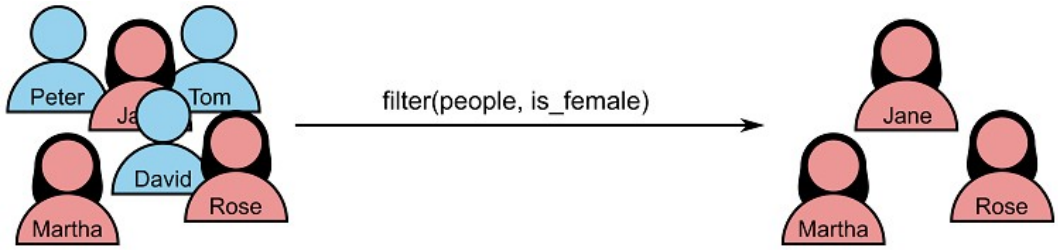

Figure 1. Filtering a collection of people based on the `is_female` predicate.
It should return us a new collection containing only females.

The first higher-level construct that we could use here is collection filtering. Generally speaking, filtering is a simple algorithm that checks whether an item in the original collection satisfies a condition, and if it does, it gets put into the result. The filtering algorithm can't know in advance the different predicates users will filter their collections with. The filtering could be done on a specific attribute (we're filtering on a specific value of the gender), on multiple attributes at the same time (for example, if we wanted only females with black hair), or on a more complex condition (getting all females that've recently bought a new car). This construct needs to provide a way for the user to specify it. In our case, it needs to allow us to provide a predicate that takes a person, and returns whether it's a female or not. Because filtering allows us to pass a predicate function, it is, by definition, a higher-order function.

Sidebar **Notation for specifying the function type**

When we want to denote an arbitrary collection containing items of type T, we'll write collection<T>, or C<T>. When we want to say that an argument for a function is another function, we'll write its type in the list of arguments. Because the filter construct takes a collection and a predicate function (T -> bool) as arguments, and it returns a collection of filtered items, we'll write its type like this:

```
filter: (collection<T>, (T -> bool)) -> collection<T>
```

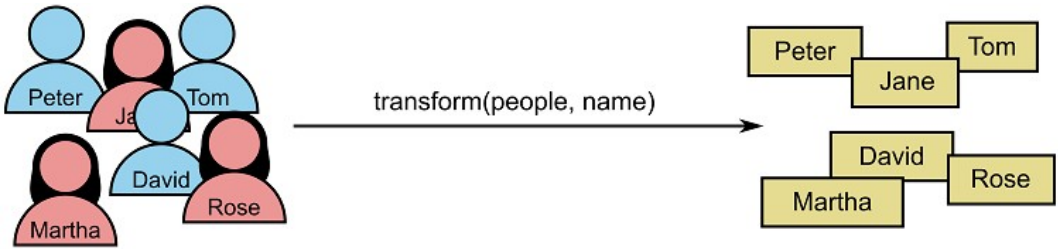

Figure 2. We have a collection of people. The `transform` algorithm should call the transformation function for each of them, and collect the results. In this case, we're passing it a function that returns the name of a person. This means that the `transform` algorithm will collect the names of every person into a new collection.

After the filtering is finished, we're left with the task of getting the names. We need a construct that takes a group of people, and returns their names. Similar to filtering, this construct can't know in advance what information we want to collect. We might want to get a value of a specific attribute (the name, in this example), or to combine multiple attributes (if we wanted to fetch and concatenate both the name and the surname) or to do something more complex (getting a list of children for each person). Again, the construct needs to allow the user to specify the function that takes an item from the collection, does something with it, and returns a value which will be put into the resulting collection. It should be noted that the output collection doesn't need to contain items of the same type as the input collection (unlike filtering). As mentioned earlier, this construct is called map or transform, and its type is:

```
transform: (collection<In>, (In -> Out)) -> collection<Out>
```

When we compose these two constructs (figure 3), by passing the result of one as the input for the other, we get the solution to our original problem—we get the names of all females in the given group of people.

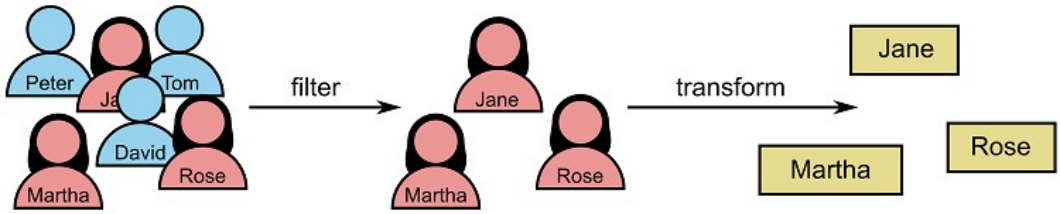

Figure 3. Now that we have one function that filters a collection of people to contain only females, and a function that extracts the names of every person in a specified collection, we can compose these two and we'll have a function that gets the names of all females in a group of people.

Both `filter` and `transform` are common programming patterns that many programmers keep implementing and re-implementing in many projects they work on. Small differences like different filtering predicates (for example,

filtering people based on gender or on age) require writing the same code all over again. Higher-order functions allow us to factor out these differences, and implement the general higher-level concept common to them all. This significantly improves the code re-use and coverage.

For more neat things you can do with functional programming in C++, download the free first chapter and see this Slideshare presentation for more info. Use code **fcccukic** to save 37% off of Functional Programming in C++, all formats.