

Functional Programming in

MEAP

Ivan Čukić



MEAP Edition
Manning Early Access Program
Functional Programming in C++
Version 11

Copyright 2017 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/functional-programming-in-cplusplus>

Licensed to Alexey Fadeev <alexey.s.fadeev@gmail.com>

Welcome

Thank you for purchasing the MEAP for *Functional Programming in C++*. I am very excited that this book is being developed during the C++ renaissance (as this period is named by Herb Sutter), in which we get significant improvements to the C++ programming language every few years. It also comes at a time when many C++ programmers are becoming interested in functional programming.

The book is mainly aimed at professional C++ programmers who are interested in functional programming, though it can be a useful source for *functional* programmers who want to get to know C++ better. While many functional programming books tend to focus mainly on the theoretical aspect (after all, functional programming is always a hot topic in academia), we are going to take the pragmatic approach and keep that at a minimum. The focus of this book is the practical side of functional programming and what tools C++ provides us to develop software in the functional style.

There are a few things I think are worth mentioning about the style of writing that you should be aware of before reading this book:

- Many C++ projects are tied to older compilers that do not support all the latest and greatest features of C++11/14/17. I have spent quite some time in the first chapters of the book to differentiate between features in the different versions of C++ standard for this reason. It would be much easier to focus only on C++17, but then a lot of people would not be able to use this book in their day-to-day projects.
- C++ is a complex language, and while most people tend to use it just fine without knowing the nitty-gritty details, I do think these details are important. C++ is the "*speed king*" in the world of programming languages, but for our programs to be fast we need to be able to understand how things work internally.
- Some things in the book follow a style not often practiced by C++ developers for easier readability. One example of this is that all types we define will have an `_t` suffix even if it is not commonly used outside of the standard library and POSIX.

I encourage you to post any questions or comments you might have about the content of the book in the [Author Online forum](#). Your feedback will be instrumental in making this book as good as it can be, so all constructive criticism you can provide is highly appreciated.

— Ivan Čukić

brief contents

- 1 Introduction to functional programming*
- 2 Getting started with functional programming*
- 3 Function objects*
- 4 Creating new functions from the old ones*
- 5 Purity – avoiding mutable state*
- 6 Lazy evaluation*
- 7 Ranges*
- 8 Functional data structures*
- 9 Algebraic data types and pattern matching*
- 10 Monads*
- 11 Template meta-programming*
- 12 Functional design for concurrent systems*
- 13 Testing and debugging*

Introduction to functional programming



This chapter covers:

- What functional programming is
- How to think about the intent instead of exact algorithm steps
- What pure functions are
- Some benefits of functional programming
- How has C++ evolved into a functional programming language

As programmers, we are required to learn more than a few different programming languages during our lifetime, and we usually end up focussing on two or three that we are most comfortable with. It is common to hear somebody saying that learning a new programming language is easy — that the differences between different languages are mainly in the syntax, and that most languages provide roughly the same features. If we know C++, it should be easy to learn Java or C#, and vice-versa.

While this claim does have some merit, we usually end up trying to simulate the style of programming we used in the previous language in the one that we are learning. When I first saw a functional programming language at my university, my first reaction was to learn how to use its features to simulate the `for` and `while` loops and `if-then-else` branching. And this was the approach most of us took just to be able to pass the exam and never look back.

There is a saying that if the only tool you have is a hammer, it is tempting to treat

every problem as if it was a nail. This also goes the other way round: if you have a nail, whatever tool you are given you will want to use that tool as a hammer. Many programmers who check out some functional programming language decide that it isn't worth it because they don't see the benefits — they try to use the new tool in the same way they used the old one.

While this book isn't meant to teach a new programming language, it *is* meant to teach an alternative way of using a language (C++), a way that is different enough that it will often *feel* like it is a new language. With this new style of programming, we can write more concise programs and write code that is safer, easier to read and reason about, and, dare I say, more beautiful than the code we usually write in C++.

1.1 What is functional programming?

Functional programming is an old programming paradigm that was born in the academia during 1950s; it stayed tied to that environment for a long time. While it was always a hot topic for scientific researchers, it was never that popular in the *real world*. Instead, the imperative languages (first the procedural ones, later object-oriented) became ubiquitous.

While it was often predicted that one day the functional programming languages would rule the world, it hasn't happened yet. We still don't have the famous functional languages like Haskell and Lisp on the *top 10* lists of the most popular programming languages. Those lists are still reserved for the traditionally imperative ones like C, Java, and C++. Like most predictions, this one needs to be open to interpretation to be considered fulfilled. Instead of the functional programming languages becoming the most popular, something else has been happening — the most popular programming languages have started introducing features inspired by functional programming languages.

So, what *is* functional programming? This question is difficult to answer because there isn't a widely accepted definition. There is a saying that if you ask two functional programmers what functional programming is, you will get (at least) three different answers. People tend to define what functional programming is through different related concepts like pure functions, lazy evaluation, pattern matching, and such. And usually, they tend to list exactly the features that their favorite language has.

In order not to alienate anyone, we'll start with an overly mathematical definition from the functional programming Usenet group. And later, during the course of this book, we'll cover the different concepts related to functional programming. I'll leave it up to you to pick your favorites that you consider essential for a language to be called functional.

Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these language are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style.

-- FAQ for comp.lang.functional

Broadly speaking, it is a style of programming where the main program building blocks are functions as opposed to objects and procedures. A program written in the functional style doesn't specify the commands that should be performed in order to achieve the result, but rather define what the result is.

Consider a small example of calculating a sum of a list of numbers. In the imperative world, we would implement this by iterating over the list, and adding the numbers to the accumulator variable. That is, we would explain the step-by-step process of how to sum a list of numbers. Contrary to that, in the functional style, we would only need to define what a sum of a list of numbers is, and the computer would know what to do when it is required to calculate it. One way we could do this is to simply say that the sum of a list of numbers equals the first element of the list added to the sum of the rest of the list, and that the sum is zero in the case the list is empty. We have just defined what the sum is without explaining how to calculate it.

This difference is the origin of the terms *imperative* and *declarative* programming. Imperative means that we are commanding the computer how it should do something by explicitly stating each step it needs to take in order to calculate the result. On the other hand, declarative means that we are just stating what should be done, and the programming language has the task of figuring out how to do it. We have defined what a sum of a list of numbers is, and the language needs to use that definition to calculate the sum of a given list of numbers.

1.1.1 Relationship with object-oriented programming

It isn't possible to say which one is better — the most popular imperative paradigm—the object-oriented programming or the most commonly used declarative one—the functional programming paradigm. Both have their own advantages and weaknesses.

The object-oriented paradigm is based on creating abstractions for the data. It allows the programmer to hide the inner representation inside an object, and only provide a view of it to the rest of the world via the object's API.

The functional programming style creates abstractions on the functions themselves. This allows creating more complex control structures than the underlying language provides. When C++11 introduced the range-based for loop (sometimes

called *foreach*), it needed to be implemented in every C++ compiler out there (and there are quite a few of those). By using functional programming techniques, this was possible to do without changing the compiler. In fact, many third-party libraries implemented their own versions of the range-based for loop over the years. When we start using the functional programming idioms, we will be able to create new language constructs like the range-based for loop, and quite a few more advanced ones. These will be useful even when writing programs in the imperative style.

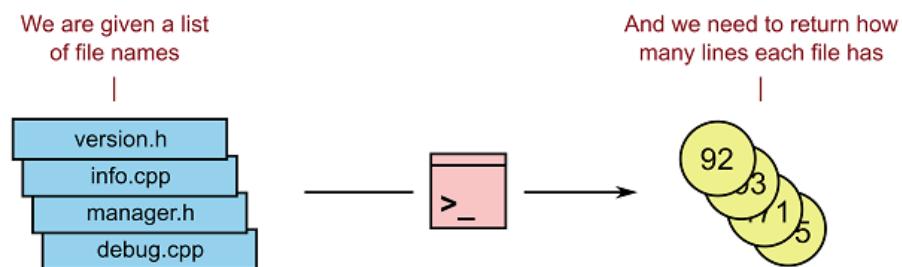
In some cases, one paradigm is more suitable than the other, and vice-versa. In many cases, the combination of the two will hit the sweet spot. This is evident from many old and new programming languages becoming multi-paradigm instead of sticking to their primary paradigm.

1.1.2 A concrete example of the difference between the imperative and declarative styles of programming

To demonstrate the difference between these two styles of programming, let's start with a simple program implemented in the imperative style, and convert it to its functional equivalent.

One of the ways often used to measure the complexity of software is counting the lines of code (LOC) that software has. While it is debatable whether this is a good metric, it is a perfect way for us to demonstrate the differences between imperative and functional programming styles. Imagine we want to write a function that takes a list of files, and calculates the number of lines each of them has (Figure 1.1). To keep this example as simple as possible, we will just count the number of newline characters in the file — we will assume that the last line in the file also ends with a newline character.

Figure 1.1. The program input is a list of files and it needs to return the number of newlines in each of them as the output



When thinking imperatively, we analyze the problem in the following way:

- We need to open each file;
- We need to define a counter to store the number of lines;

- We should read the file one character at a time and increase the counter every time the newline character (\n) occurs;
- When we reach the end of a file, we store the number of lines we have calculated.

Listing 1.1. Calculating the number of lines the imperative way – reading files character by character and counting the number of newlines

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    std::vector<int> results;
    char c = 0;

    for (const auto& file : files) {
        int line_count = 0;

        std::ifstream in(file);

        while (in.get(c)) {
            if (c == '\n') {
                line_count++;
            }
        }

        results.push_back(line_count);
    }

    return results;
}
```

We end up with two nested loops and a few variables to keep the current state of the process (Listing 1.1). While the example is quite simple, it has a few places where the programmer might make an error — be it an uninitialized (or badly initialized) variable, an improperly updated state, or a wrong loop condition. The compiler will report some of these mistakes as warnings, but the mistakes that get through are usually hard to find because our brains are hard-wired to ignore them, just like spelling errors. We should try to write our code in a way that minimizes the possibility of making mistakes like these.

More C++-savvy readers might have noticed that we could have used the standard algorithm `std::count` instead of counting the number of newlines manually. C++ provides convenient abstractions like stream iterators that allow us to treat the I/O streams similarly to ordinary collections like lists and vectors, so we might as well use them (Listing 1.2).

Listing 1.2. Using std::count to count the number of newline characters (example:count-lines-stdcount/main.cpp)

```
int count_lines(const std::string& filename)
{
    std::ifstream in(filename);
    in.unsetf(std::ios_base::skipws);

    return std::count(
        std::istream_iterator<char>(in),           1
        std::istream_iterator<char>(),               1
        '\n');                                       1
}

std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    std::vector<int> results;

    for (const auto& file : files) {
        results.push_back(count_lines(file));  2
    }

    return results;
}
```

- ① Counting newlines from the current position in the stream, until the end of the file
- ② Saving the result

We got a solution in which we are no longer concerned about how exactly the counting is implemented; we are just declaring that we want to count the number of newlines that appear in the given input stream. This is always the main idea when writing programs in the functional style — use abstractions which allow us to define the **intent** instead of specifying **how** to do something — and is the aim of most techniques that we will cover in this book. This is the reason why functional programming goes hand-in-hand with generic programming (especially in C++) — because both allow the programmer to think on a higher level of abstraction compared to the down-to-earth view of the imperative programming style.

Note

One thing that always amused me is that most developers would say that C++ is an object-oriented language.

The reason for it being amusing is that barely any parts of the standard library of the C++ programming language (commonly referred to as the **STL**—Standard Template Library) use inheritance-based polymorphism which is at the heart of the object-oriented programming paradigm.

The **STL** was created by Alexander Stepanov, a vocal critic of OOP. He wanted to create a generic programming library, and he did so by leveraging the C++ templates system combined with a few functional programming techniques.

This is one of the reasons why I'm going to rely a lot on STL in this book — even if it isn't a *proper FP library*, it models a lot of FP concepts which makes it a great starting point to entering the world of functional programming.

The benefit of this solution is that we have fewer state variables to worry about, and we have started to express the higher-level intent of our program instead of specifying the exact steps it needs to take in order to find the result. We no longer care how the counting is implemented. The only task of the `count_lines` function is to convert its input (the file name) to the type that the `std::count` can understand (a pair of stream iterators).

Let's take this even further. Let's try to define the whole algorithm in the functional style — *what* should be done, instead of *how* it should be done. We are left with a range-based for loop that just applies a function to all elements in a collection and collects the results. This is a common pattern, and it is to be expected that the programming language has support for it in its standard library. In C++, this is what `std::transform` is for (in other languages, this is usually called `map` or `fmap`). The implementation of the same logic with the `std::transform` algorithm is shown in the Listing 1.3 . The `std::transform` traverses the items in the `files` collection one by one, transforms them using the function `count_lines`, and stores the resulting values in the `results` vector.

Listing 1.3. Mapping files to line counts using std::transform (example:count-lines-transform/main.cpp)

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    std::vector<int> results(files.size());

    std::transform(files.cbegin(), files.cend(), ❶
                  results.begin(), ❷
                  count_lines); ❸

    return results;
}
```

- ❶ Specifying which items to transform
- ❷ The transformation function
- ❸ Where to store the results

This code is no longer specifying the algorithm steps that need to be taken, but rather how the input should be transformed in order to get the desired output. It can be argued that by removing the state variables, and by relying on the standard library implementation of the counting algorithm instead of rolling out our own, we have made the code less prone to errors.

The problem is that it has too much boilerplate code to be considered more readable than the original example. The above function only has three important words:

- `transform` (what are we doing),
- `files` (the input),
- and `count_lines` (transformation function);

everything else is just noise.

The function would be much more readable if we were able just to write the important bits and skip everything else. In chapter 7, we'll see that this is achievable with the help of the ranges library. Here, we are just going to show what this function would look like when implemented with ranges and range transformations. Ranges use the `|` (pipe) operator to denote pushing a collection through a transformation:

Listing 1.4. Transformation using ranges

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    return files | transform(count_lines);
}
```

This code snippet does the same as the previous one, but the meaning is more obvious. We take the input list, pass it through the transformation, and return the result.

Note

Notation for specifying the function type

C++ does not have a single type to represent a function (we will see all the different things that C++ considers to be function-like in chapter 3). In order to be able to specify just the argument types and the return type of a function, without specifying exactly what type it will have in C++, we will need to introduce a new language-independent notation.

We will write `f: (arg1_t, arg2_t, ..., argn_t) → result_t`. It will mean that `f` accepts `n` arguments, where `arg1_t` is the type of the first argument, `arg2_t` of the second, etc.; and that `f` returns a value of type `result_t`. If the function takes only one argument, we will omit the parentheses around the argument type. It is also worth noting that we will avoid using `const`-references in this notation for simplicity.

For example, if we say that the function `repeat` has a type of `(char, int) → string`, it means that it takes two arguments — one character and one integer, and returns a string. In C++, it would be written like this (the second version is available since C++11):

```
std::string repeat(char c, int count);
auto repeat(char c, int count) -> std::string;
```

This form also increases the maintainability of the code. You might have noticed that our `count_lines` function has a design flaw. If somebody were to look just at

its name and type (`count_lines: std::string → int`), the function takes a const-reference to string], it wouldn't be clear that the string represents a file name. It would be normal to expect that the function counts the number of lines in the passed string instead. If we wanted to fix this issue, we could separate this function into two — `open_file: std::string → std::ifstream` which takes the file name and returns the file stream, and `count_lines: std::ifstream → int` which counts the number of lines in the given stream. With this change, it would be obvious what the functions do from their names and involved types. Changing the range-based `count_lines_in_files` function would involve just one additional transformation:

Listing 1.5. Transformation using ranges, modified

```
std::vector<int>
count_lines_in_files(const std::vector<std::string> &files)
{
    return files | transform(open_file)
                  | transform(count_lines);
}
```

This solution is much less verbose than the imperative one we started with, and much more obvious. We start with a collection of file names, it doesn't even matter which collection type are we using, and we perform two transformations on each element in that collection. First, we take the file name and create a stream from it, and then go through that stream to count the newline characters. This is exactly what the code above says — without any excess syntax, and without any boilerplate.

1.2 Pure functions

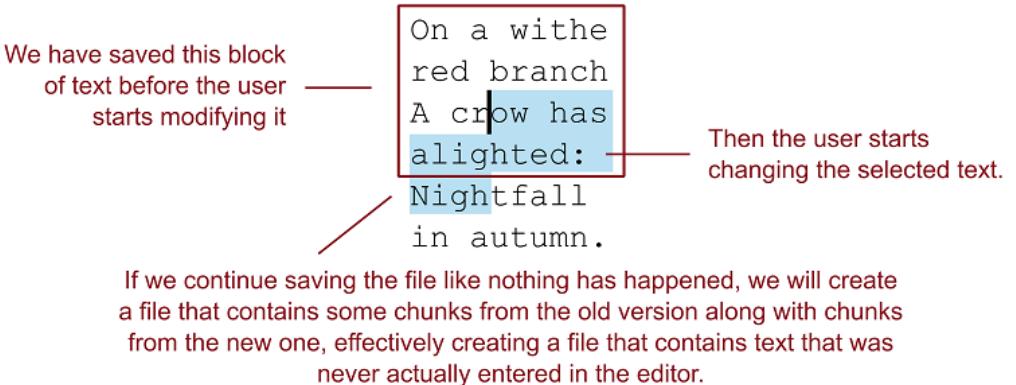
One of the greatest source of software bugs is the program state. It's difficult to keep track of all possible states a program can be in. The object-oriented programming paradigm gives us the option to group parts of the state into objects, thus making it easier to manage. But it doesn't reduce the number of possible states significantly.

Say we are making a text editor, and we are keeping the text that the user has written in a variable. The user presses the save button and continues typing. The program saves the text by writing one character at a time to the storage (this is a bit oversimplified, but bear with me). What happens when the user changes a part of the text while the program is saving it? Is the program going to save the text as it was when the user clicked the save button, is it going to save the current version, or could something else happen?

The problem is that all three cases are possible — and the answer will depend on the progress of the saving operation and on what part of the text the user is

changing. In the case presented in the Figure 1.2 , the program will save text that was never actually in the editor. Some parts of the saved file will come from the text as it was before the change has occurred, while some parts will be from the text after it has been changed. We will have saved parts of two different states at the same time.

Figure 1.2. If we allow the user to modify the text while we are saving it, we will open ourselves to saving incomplete or invalid data thus creating a corrupted file



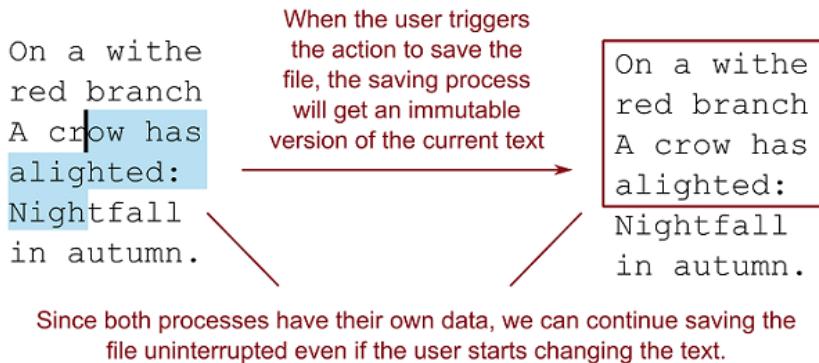
This issue wouldn't exist if the saving function had its own immutable copy of the data that it should write. This is the biggest problem of mutable state — it creates dependencies between the parts of the program that don't need to have anything in common. In this example, we have two clearly separate user actions — saving the typed text, and typing the text. These should be able to be performed independently of one another. By having multiple actions that might be executed at the same time and that share a mutable state, we create a dependency between them, and open ourselves to issues like the above one.

OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.

-- Michael Feathers, author of *Working with Legacy Code*

Even local mutable variables can be considered bad for the same reason. They create dependencies between different parts of the function, making it difficult to factor out parts of it into a separate function.

Figure 1.3. If we create a copy (either a full copy, or by using some structure that can remember multiple versions of data at the same time), we will be able to decouple the processes of saving the file and changing the text in the text editor



One of the most powerful ideas of functional programming are the pure functions. That is, the functions that only use (but don't modify) the arguments passed to them in order to calculate the result. If a pure function is called multiple times with the same arguments, it must return the same result every time and leave no trace it was ever invoked (no side-effects). This all implies that pure functions are unable to alter the state of the program.

This is great, because it allows us not to think about the program state at all. But, unfortunately, it also implies that pure functions cannot read from the standard input, write to the standard output, create nor delete files, insert rows into a database, and so on. If we wanted to be overly dedicated to immutability, we would even have to forbid them also to change the processor registers, memory, or anything else on the hardware level.

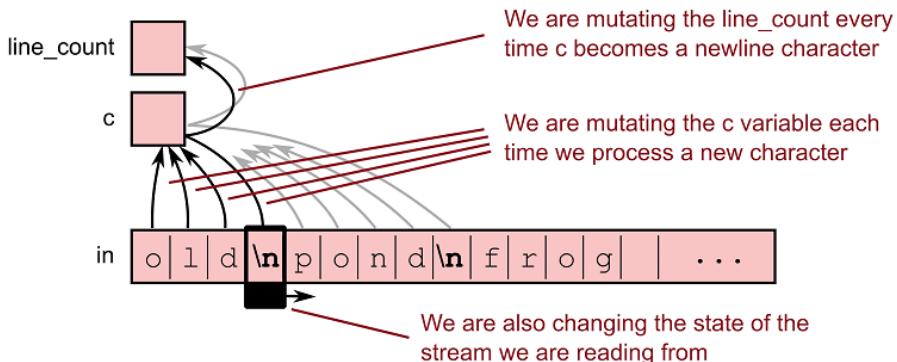
This makes this definition of pure functions unusable. The CPU executes instructions one by one, and it needs to track which instruction needs to be executed next. This means that we cannot execute anything on the computer without mutating at least the internal state of the CPU. Apart from that, we couldn't write useful programs if we couldn't communicate with the user or another software system.

Because of this, we are going to relax our requirements a bit and refine our definition — a pure function will be any function that doesn't have observable (at a higher level) side-effects. This means that the function caller shouldn't be able to see any trace that the function was executed, apart from getting the result of the call. We will also not limit ourselves to only use and write pure functions, but we will try to limit the number of the non-pure ones.

1.2.1 Avoiding mutable state

We started talking about the functional programming style by considering an imperative implementation of the algorithm which counts newlines in a collection of files. The function that counts newlines should always return the same array of integers when invoked over the same list of files (provided that the files weren't changed by an external entity). This means that it could be implemented as a pure function.

Figure 1.4. We have a couple of independent variables that we need to modify while counting the number of newlines in a single file. Some changes depend on each other, while some of them don't



When we look at the initial implementation of this function, we can see quite a few statements that are impure:

```

for (const auto &file: files) {
    int line_count = 0;

    std::ifstream in(file);

    while (in.get(c)) {
        if (c == '\n') {
            line_count++;
        }
    }

    results.push_back(line_count);
}

```

Calling `.get` on an input stream changes the stream and the value stored in the variable `c`, we are changing the `results` array by appending new values to it, and we also modify the `line_count` by incrementing it (Figure 1.4 shows the state changes for processing a single file). So, this function is definitely not implemented in a pure way.

But this isn't the only question that we need to ask ourselves. The other important question is whether its impurities are observable from the outside. All mutable variables in this function are local, not even shared between possible concurrent invocations of the function, and aren't visible to the caller, nor to any of the external entities. This means that the users of this function can consider it to be pure, even if the actual implementation isn't. This benefits the callers because they can rely on us not changing their state, but we still have to manage our own. And while doing so, we must ensure that we aren't changing anything that doesn't belong to us. Naturally, it would be better if we also limited our state and tried to make the function implementation as pure as possible. If we make sure that we are using only pure functions in our implementation, we won't need to think about whether we are leaking any state changes since we aren't mutating anything at all.

The second solution (Listing 1.2) separates the actual counting into a function named `count_lines`. This function is also pure-looking from the outside, even if it internally declares an input stream and modifies it. Unfortunately, due to the API of `std::ifstream`, this is the best we can get:

```
int count_lines(const std::string& filename)
{
    std::ifstream in(filename);

    return std::count(
        std::istreambuf_iterator<char>(in),
        std::istreambuf_iterator<char>(),
        '\n');
}
```

On the other hand, we haven't improved the `count_lines_in_files` function in any significant manner in this step. We have just moved some of its impurities to a different place, but still kept the two mutable variables. Unlike `count_lines`, this function doesn't need IO, and is implemented only in terms of the `count_lines` function, which we (as a caller) can consider to be pure. There is no reason why it would contain any impure parts at all. In our last version of the code that uses the range-notation, we managed to do just that. We removed all state, mutable or not, and defined the whole function in terms of just other function calls on the input:

```
std::vector<int>
count_lines_in_files(const std::vector<std::string>& files)
{
    return files | transform(count_lines);
```

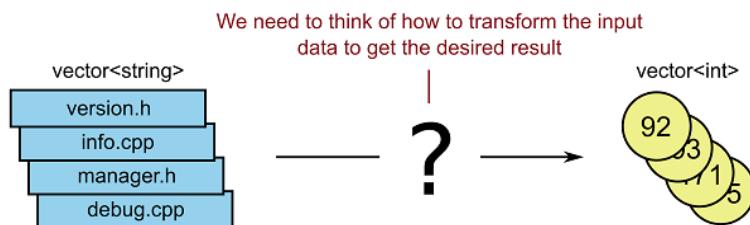
This solution is a perfect example of what functional programming style looks like. It is short, concise, and it is quite obvious what it does. What's more, it is obvious that it doesn't do anything else — it has no visible side-effects, it just gives the

desired output for the given input.

1.3 Thinking functionally

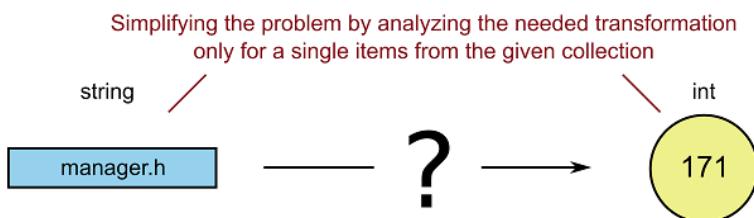
It would be inefficient and counter-productive to write the code in imperative style first, and then change it bit by bit until it becomes functional. Instead, we should start thinking about problems differently. Instead of thinking of the algorithm steps, we should think about what our input is, what the output is, and which transformations we should perform in order to map one to the other.

Figure 1.5. When thinking functionally, we need to think about the transformations we need to apply to the given input, so that we get the desired output as the result



In our example (Figure 1.5), we have been given a list of file names, and for each file we need to calculate the number of lines in it. The first thing we need to notice is that we can simplify this problem by considering only a single file at a time. We do have a list of file names, but we can process each of them independently of the rest. If we can find a way to solve this problem for a single file, we will easily solve the original problem as well (Figure 1.6).

Figure 1.6. The first step is to notice that we are performing the same transformation on each element in a collection – this allows us to look at the simpler problem of transforming just a single item instead of a collection of items

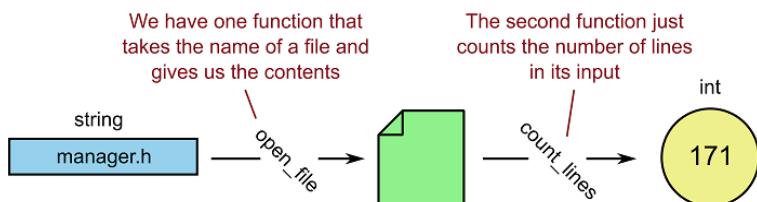


Now, our main problem is to define a function that takes a file name and calculates the number of lines in the file represented by that file name. From this definition, we can see that we are given one thing (the file name) while we actually need something else (the file contents, so that we can count the newline characters in it). Therefore, we will need a function that can give us the contents of a file when we provide it with a file name. Now, whether the contents should be returned as a

string, a file stream, or something else, it is up to the programmer to decide. It just needs to be able to provide us with one character at a time, so that we can pass it to the function that counts the newlines.

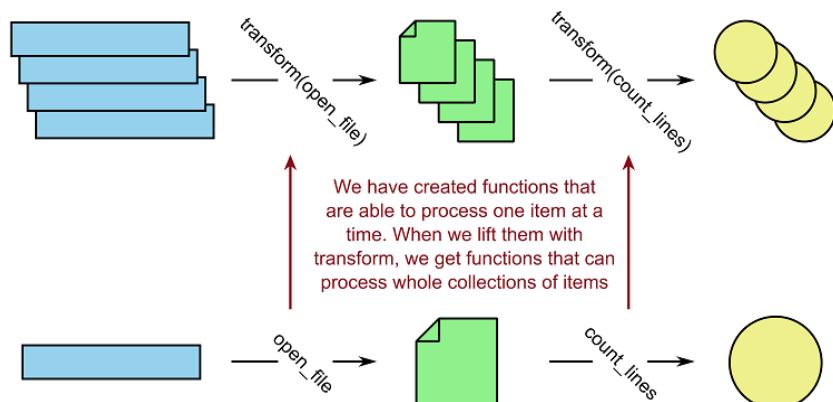
When we have the function that gives us the contents of a file (`string → ifstream`), we are able to call the function that counts the lines on its result (`ifstream → int`). Composing these two functions by passing the `ifstream` created by the first one as the input to the second gives us the function that we wanted (Figure 1.7).

Figure 1.7. Decomposing a bigger problem of counting the number of lines in a file whose name we have into two smaller ones — opening a file given its name, and counting the number of lines in a given file



With this, we have solved our problem. We just need to *lift* these two functions up to be able to work not only on a single value, but to work on a collection of values. This is conceptually what `std::transform` does (just with a more complicated API) — it takes a function that can be applied to a single value, and creates a transformation that can work on a whole collection of values (Figure 1.8). For the time being, you can think of lifting as a generic way to convert functions that operate on simple values of some type to functions that work on more complex data structures containing values of said type. We will cover *lifting* in more detail in chapter 4 (section "Function lifting revisited").

Figure 1.8. By using the transform, we are creating functions that are able to process collections of items from functions that can process only one item at a time

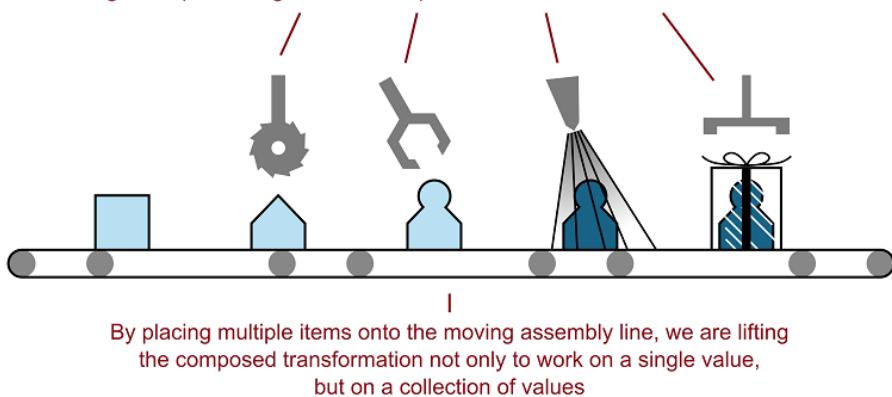


With this simple example, we have seen the functional approach to splitting bigger programming problems into smaller independent tasks that are easily composed.

One useful analogy for thinking about function composition and lifting is the moving assembly line. At the beginning, there is the raw material that the final product will be composed of. It goes through different machines that transform it. And in the end, we get the final product. With the assembly line, we are thinking about the transformations that the product is going through instead of the steps that the machine needs to perform.

Figure 1.9. Thinking about function composition and lifting might be easier if compared to a moving assembly line. We have different transformations that are made to work on single items. By lifting these transformations to work on collections of items, and composing them so that the result of one transformation gets passed on to the next one, we get an assembly line which applies a series of transformations to as many items as we want.

We have different transformations that we need to apply one by one to the given input. This gives us a composition of all these transformation functions.



In our case, the raw material is the input that we receive, and the machines are the different functions that we apply to it. Each function is highly specialized to do one simple task without concerning itself with the rest of the assembly line. It just requires a valid input, it doesn't care where it comes from. The input items are placed on the moving assembly line one by one (or we could have a few assembly lines which would allow us to process a few items in parallel). Each item is transformed, and we get a collection of transformed items in the end.

1.4 Benefits of functional programming

Different concepts of functional programming give us different benefits. We will cover them in due course, but we can start here with a few main benefits that most of these concepts aim to bring.

The most obvious thing that most people notice when starting to implement

programs in the functional style is that the code becomes much shorter. Some projects even have official code annotations like "*could have been one line in Haskell*". This comes from the fact that the tools that the functional programming gives us are simple but highly expressive, and that most things can be implemented on a higher level without bothering with gritty details.

This characteristic, combined with purity, has brought the functional programming style into the spotlight in the recent years. Purity improves the correctness of the code, while expressiveness allow us to write less code in which we might make mistakes.

1.4.1 Code brevity and readability

One of the things that functional programmers claim is that it is easier to understand programs written in the functional style. This is subjective, and people who are used to writing and reading imperative code may disagree with that statement. Objectively, it can be said that programs written in the functional style tend to be shorter and more concise. It was also apparent in our example. We started with twenty lines of code, and ended up with a single line for the `count_lines_in_files` function and around five lines for the `count_lines`, which mainly consisted of boilerplate code imposed on us by C++ and the standard library (or STL). This was only possible to achieve because we allowed ourselves to use higher-level abstractions that the functional programming parts of STL give us.

One unfortunate truth is that many C++ programmers tend to stay away from using higher-level abstractions like STL algorithms. They state various reasons for this — from being able to write more performant code manually, to avoiding writing the code that their colleagues might not easily understand. While these reasons are valid sometimes, they aren't in the majority of cases. Not availing yourself of more advanced features of the programming language you are using just lowers the power and the expressiveness of said language and makes your code more complex and more difficult to maintain.

In 1987, Edsger Dijkstra published a paper "Go To Statement Considered Harmful", in which he advocated abandoning the GOTO statement that was overused in that period in favor of *structured programming* and using higher-level constructs like routines, loops, and if-then-else branching.

"The unbridled use of the go to statement has as an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. ... The go to statement as it stands is just too primitive, it is too much an invitation to make a mess of one's program."

-- Edsger Dijkstra

In many cases, loops and branching are also overly primitive. And just like with the GOTO, loops and branching can make it harder to write and understand programs, and can often be replaced by even more higher-level functional programming constructs. We often write the same code in multiple places without even noticing that it is the same because it works with different types, or has some differences in behavior that could easily be factored out.

By using existing abstractions provided by STL or a third-party library, and by creating our own, we make our code safer and shorter. But, we also make it easier to expose bugs in those abstractions, since the same code ends up being used in multiple places.

1.4.2 Concurrency and synchronization

The main problem when developing concurrent systems is the shared mutable state. It requires the programmer to pay extra attention to ensure that the different components don't interfere with one another.

Parallelizing programs written with pure functions is trivial, since those functions don't mutate anything. There's no need for explicit synchronization with atomics or mutexes. This means that we can run the code written for a single-threaded system on multiple threads with almost no changes. We cover this in more depth in chapter 12.

Consider the following code snippet in which we want to sum the square roots of values in the `xs` vector:

```
std::vector<double> xs = {1.0, 2.0, ...};
auto result = sum(xs | transform(sqrt));
```

If the `sqrt` implementation is pure (there's no reason for it not to be), the implementation of the `sum` algorithm might automatically split the input into chunks and calculate partial sums for those chunks on separate threads. When all threads finish, it would just need to collect the results and sum them.

Unfortunately, C++ doesn't (yet) have a notion of a pure function, so parallelization can't be performed automatically. Instead, you would need to explicitly call the parallel version of the `sum` algorithm. The `sum` function might even be able to detect the number of CPU cores at runtime, and use this information when deciding on the number of chunks the `xs` vector should be split into. If we wrote the above code with a `for` loop, it wouldn't be as easy to parallelize it. We would need to think about ensuring that variables aren't changed by different threads at the same time, and that we are creating exactly the number of threads optimal for the system our program is running on, instead of leaving all that to the library that provides us with the summing algorithm.

Note **Compiler optimization**

C++ compilers can sometimes perform automatic vectorization or other optimizations when they recognize that the loop bodies are pure. This optimization also affects the code that uses standard algorithms, since the standard algorithms are usually internally implemented with loops.

1.4.3 Continuous optimization

There is another big benefit of using higher-level programming abstractions from STL or other trusted libraries — our program will improve over time even if we don't change a single line. Every improvement of the programming language, the compiler implementation, and the implementation of the used library will improve our program as well. While this is true for both the functional and non-functional higher-level abstractions, the functional programming concepts significantly increase the amount of code that we can cover with these.

This seems like a no-brainer, but many programmers prefer writing low-level *performance critical* code manually, sometimes even in assembly language. This approach sometimes has benefits, but most of the time it just optimizes the code for a specific target platform, and makes it borderline impossible for the compiler to optimize it for another.

Let's consider our `sum` function. We might optimize it for a system that uses instruction pre-fetching by making the inner loop take two (or more) items in every iteration, instead of summing the numbers one by one. This lowers the number of jumps in the code, so the CPU will pre-fetch the correct instructions more often. This will obviously improve the performance for the target platform. But what happens if we run the same program on a different platform? For some platforms, it might be more optimal to have the original loop, while for others it might be more optimal to have even more items summed with every iteration of the loop. And some systems might even provide us with a CPU instruction that does exactly what our function needs.

By manually optimizing code in this way, we miss the mark on all platforms but one. If we use the higher-level abstractions, we are relying on other people to write optimized code. In the case of most STL implementations, it can be expected that they are providing specific optimizations for the platforms and compilers they are targeting.

1.5 Evolution of C++ as a functional programming language

C++ was born as an extension of the C programming language to allow writing object-oriented code (initially called "C with classes"). But even after its first standardised version (C++98), it was difficult to call it just object-oriented. Namely, since the introduction of templates into the language, and the creation of the standard template library (STL), which only sparsely uses inheritance and virtual member functions, C++ became a proper multi-paradigm language.

Considering the design and implementation of STL, it can even be argued that C++ isn't primarily an object-oriented language, but a generic programming language. Generic programming is based on the idea that we can write code that uses some general concepts, and then we can apply it to any structure that fits those concepts. For example, the standard library provides the `vector` template that we can use over different types like ints, strings, or user types that satisfy some preconditions. The compiler will then generate optimized code for each of the specified types. This is usually called static or compile-time polymorphism, as opposed to dynamic or run-time polymorphism that inheritance and virtual member functions provide.

For functional programming in C++, the importance of templates isn't (mainly) in the creation of container classes like `vector`, but in the fact that it allowed creation of STL algorithms — a set of common algorithm patterns such as sorting, counting and alike. Most of these algorithms allow us to pass custom functions to customise their behaviour without resorting to function pointers and `void*`. This way we can change the sorting order, define which items should be included when counting, and similar.

The fact that we can pass functions as arguments to another function, and that we can have functions that return new functions (or more precisely, things that **look** like functions which we will cover in chapter 3) basically made even the first standardised version of C++ a functional programming language.

C++11, C++14 and C++17 have brought quite a few features that make writing programs in the functional style much easier. The additional features are mostly syntactic sugar, but nevertheless, important syntactic sugar in the form of the `auto` keyword and lambdas (which we will cover in chapter 3). They also brought significant improvements to the set of standard algorithms. The next revision of the standard is planned for 2020, and is planned to bring even more FP-inspired features like ranges, concepts and coroutines, which are currently released as Technical Specifications.

Note

ISO C++ Standard Evolution

The C++ programming language is an ISO standard. This means that every new version goes through a rigorous process before getting released. The core language and the standard library are developed by a committee, which means that each new feature is discussed thoroughly and voted on before it becomes a part of the final proposal for the new standard version.

In the end, when all changes are incorporated into the definition of the standard, it needs to pass another vote — the final vote that happens for any new ISO standard.

Starting in 2012, the committee has separated the work into separate sub-groups. Each group works on a specific language feature and these can be delivered when they are ready (when the group deems them ready) as Technical Specifications (or TS for short) that are separate from the main Standard can later be incorporated into the Standard.

The compiler vendors aren't required to implement TSs, but they usually do. The purpose of TSs

is for the developers to test the features, to uncover kinks and bugs before they are included into the main Standard.

You can find more information at isocpp.org/std/status.

While most of the concepts that we will cover during the course of this book can be used with older C++ versions, we will mostly focus on C++14 and C++17.

1.6 What you will learn in this book

This book is mainly aimed at experienced developers who use C++ every day and who want to add more powerful tools to their toolbox. To get the most out of this book, you should be familiar with the basic C++ features such as the C++ type system, references, const-ness, templates, operator overloading, and so on. You don't need to be familiar with the features introduced in C++14/17 as those are covered in more detail in the book, as these features aren't yet as widely used and it is likely that many readers won't be familiar with them.

We will start with the basic concepts like higher-order functions which allow us to increase the expressiveness of the language thus making our main programs shorter, and how to design software without mutable state to avoid the problems of explicit synchronization in concurrent software systems. After this, we will switch to the second gear and cover more advanced topics such as ranges — the truly composable alternative to the standard library algorithms, and algebraic data types — which we will use to reduce the number of states our program can be in. In the end, we will cover one of the most talked about idioms in FP — the infamous *monad* and how we can use various monads to implement complex highly-composable systems.

By the time you finish reading this book, you will be able to design and implement safer concurrent systems that can scale horizontally without much effort; to implement the program state in a way that minimizes or even removes the possibility for the program to ever be in an invalid state due to an error or a bug; to think about software as a data flow and to use *the next big C++ thing* — ranges — to define this data flow; and so on. With these skills, you will be able to write terser and less error-prone code even when working on object-oriented software systems. And if you take the full dive into the functional style, it will allow you to design software systems in a cleaner and more composable way, as we will see in the chapter 13 when we implement a simple web service.

1.7 Summary

- The main philosophy of functional programming is that we shouldn't concern ourselves with the way something should work, but rather with what it should do.
- Both approaches, functional programming and object-oriented programming, have a lot to offer — we should know when to use one, when the other, and when to combine them.

- C++ is a multi-paradigm programming language that allows us to write programs in various styles — procedural, object-oriented, and functional, and to combine those styles with generic programming.
- Functional programming goes hand-in-hand with generic programming, especially in C++. They both inspire the programmer not to think at the hardware level, but to move higher.
- Function lifting allows us to create functions that operate on collections of values from functions that operate only on single values. Function composition allows us to pass a value through a chain of transformations — each transformation passing on its result to the next one.
- Avoiding mutable state improves the correctness of the code and removes the need for mutexes in multithreaded code.
- When thinking functionally, we tend to think about the input data and the transformations we need to perform in order to get the desired output.

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch01

Getting started with functional programming

This chapter covers:

- Higher-order functions
- Using higher-order functions from STL
- Problems with composability of STL algorithms
- Recursion and tail-call optimization
- The power of the folding algorithm

The previous chapter has shown us a few neat examples of how we can improve our code by using some simpler functional programming techniques. We focused on the benefits like greater code conciseness, correctness and efficiency, but we haven't covered what special magic do functional programming language posses which allow writing code in that manner.

The truth is once you look behind the curtain, there is no magic. Just some very simple concepts with big consequences. The first simple yet far-reaching concept is the ability we saw earlier to pass one function as an argument to an algorithm from the standard library. The algorithms in the STL are applicable to a multitude of different problems mainly because they allow us to customize their behaviour in this manner.

2.1 Functions taking functions?

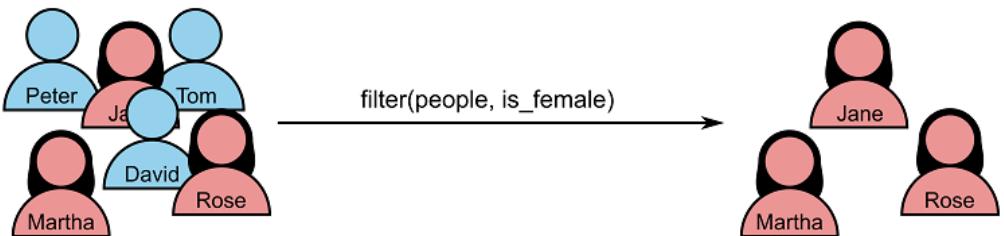
The main feature of all functional programming languages is that functions can be treated like ordinary values. They can be stored into variables, put into collections

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*. They are probably the most important concept in functional programming. In the previous chapter, we have seen how we could make our programs more concise and efficient by describing what the program should do on a higher level with the help of some 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 to write out the names of all females in that group (Figure 2.1).

Figure 2.1. 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 use to filter their collections on. The filtering could be done on a specific attribute (we are filtering on a specific value of the gender), on multiple attributes at the same time (for example, if we wanted to get all females with black hair), or on a more complex condition (getting all females that have recently bought a new car). So, 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 is a female or not. Since filtering allows us to pass a predicate function to it it is, by definition, a higher-order function.

Note

Notation for specifying the function type, continued

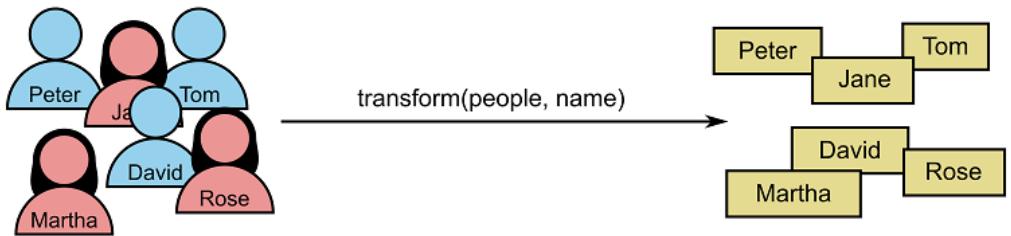
When we want to denote an arbitrary collection containing items of some type T, we will write `collection<T>`, or just `C<T>`. When we want to say that an argument for a function is another function, we will just write its type in the list of arguments.

Since the `filter` construct takes a collection and a predicate function ($T \rightarrow \text{bool}$) as

arguments, and it returns a collection of filtered items, we will write its type like this:

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

Figure 2.2. 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 are passing it a function which returns a name of a person. This means that the transform algorithm will collect the names of every person into a new collection.

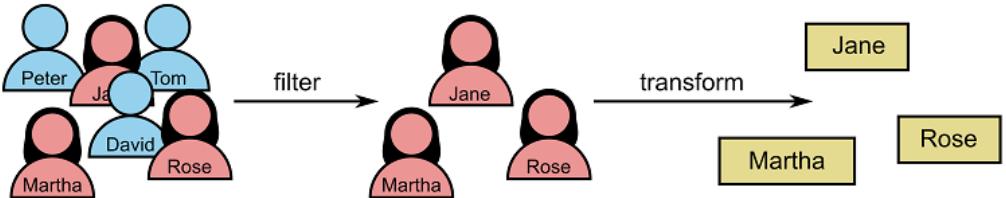


After the filtering is finished, we are left with the task of getting the names. We need a construct that takes a group of people, and returns their names. Similar to filtering, this construct can't know in advance what information we want to collect from the original items. We might want to get a value 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 2.3), 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 2.3. Now that we have one function that filters a collection of people to contain only females, and a function that extracts names of every person in a specified collection, we can compose these two and we will get 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 that is common to all of them. This also significantly improves the code reusability and coverage.

2.2 Examples from the STL, algorithm library

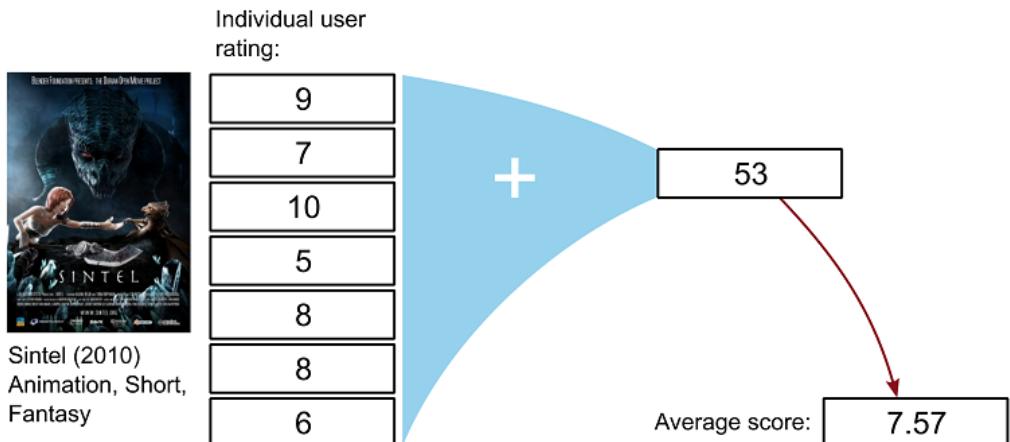
The standard library contains many higher-order functions disguised under the name of *algorithms*. It provides efficient implementations of many common programming patterns. Using the standard algorithms instead of writing everything on a lower-level with loops, branching and recursion, allows us to implement the program logic with less code and fewer bugs.

We aren't going to cover all higher-order functions that exist in the standard library, but we will see a few more interesting ones to pique your interest.

2.2.1 Calculating averages

Imagine we have a list of scores that the website visitors have given to a particular film, and we want to calculate the average. The imperative way of implementing this is to loop over all the scores, sum them one by one, and divide that sum with the total number of scores (Listing 2.1).

Figure 2.4. Calculating the average score for a movie from a list of individual user ratings. We first sum all the individual ratings, then divide the sum by the number of users.



Listing 2.1. Calculating the average score, imperatively

```
double average_score(const std::vector<int>& scores)
{
    int sum = 0;                      ①
    for (int score : scores) {        ②
        sum += score;                ②
    }
    return sum / (double)scores.size(); ③
}
```

- ① Initial value for summing
- ② Summing all the scores
- ③ Calculating the average score

While this approach works, it is unnecessarily verbose. There are a few places where we might make a mistake like using a wrong type while iterating over the collection, making a typo in the loop which would change the code semantics while still allowing it to compile, etc. It also defines an inherently serial implementation for summing, whereas summing can be easily parallelized and executed on multiple cores, or even on some specialized hardware.

The standard library provides us with a higher-order function that can sum all the items in a collection — the `std::accumulate` algorithm. It takes the collection (as a pair of iterators) and the initial value for summing, and it returns us the sum of the initial value and all items in the collection. We just need to divide it by the total number of scores (Listing 2.2) like in the previous example. This implementation doesn't specify how the summing should be performed, it states only what should

be done —the implementation is as general as the problem presentation in the Figure 2.4.

Listing 2.2. Caclulating the average score, functionally (example:average-score/main.cpp)

```
double average_score(const std::vector<int>& scores)
{
    return std::accumulate(
        scores.cbegin(), scores.cend(),
        0
    ) / (double)scores.size();
}
```

- ① Initial value for summing
- ② Summing all the scores in the collection
- ③ Calculating the average score

While the `std::accumulate` algorithm is also a serial implementation of summation, it would be trivial to replace it with a parallel version. On the other hand, making a parallelized version of our initial implementation would be non-trivial.

Parallel versions of standard algorithms

Since C++17, many algorithms in the standard library now allow you to specify that their execution should be parallelized.

The algorithms that can be parallelized accept the execution policy as an additional argument. If we want to have the algorithm execution to be parallelized, we just need to pass it the `std::execution::par` policy.¹

The `std::accumulate` algorithm is a bit special. It guarantees that the items in a collection will be accumulated sequentially from the first, to the last one, which makes it impossible to parallelize without changing its behaviour. If we want to sum all elements, but to have it parallelized, we need to use the `std::reduce` algorithm:

```
double average_score(const std::vector<int>& scores)
{
    return std::reduce(
        std::execution::par,
        scores.cbegin(), scores.cend(),
        0
    ) / (double) scores.length();
}
```

If you have an older compiler and STL implementation that doesn't fully support C++17, you can find the `reduce` algorithm in the `std::experimental::parallel` namespace.

¹ For more information on execution policies, check out en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t

But, something is missing here. We said that the `std::accumulate` algorithm is a higher-order function, yet we didn't pass it another function as an argument, nor did it return us a new function.

By default, the `std::accumulate` algorithm sums the items, but it allows us to provide it with a custom function if we want to change that behaviour. If we needed to calculate the product of all the scores for some reason, it could be done by simply passing `std::multiplies` as the last argument of `std::accumulate` and setting the initial value to 1 (Listing 2.3). We will cover the function objects like `std::multiplies` in detail in the next chapter. At this point, it is only important to note that it takes two values of a certain type, and returns their product.

Listing 2.3. Calculating the product of all scores

```
double scores_product(const std::vector<int>& scores)
{
    return std::accumulate(
        scores.cbegin(), scores.cend(),          2
        1,                                         2
        std::multiplies<int>()                  3
    );
}
```

- ① Initial value for calculating the product
- ② Calculating the product of all the scores
- ③ Multiplying the scores instead of summing

Being able to sum a collection of integers with a single function call has its benefits, but it isn't really that impressive if we limit ourselves just to calculating sums or products. On the other hand, we will see that the ability to replace addition and multiplication with something more interesting is.

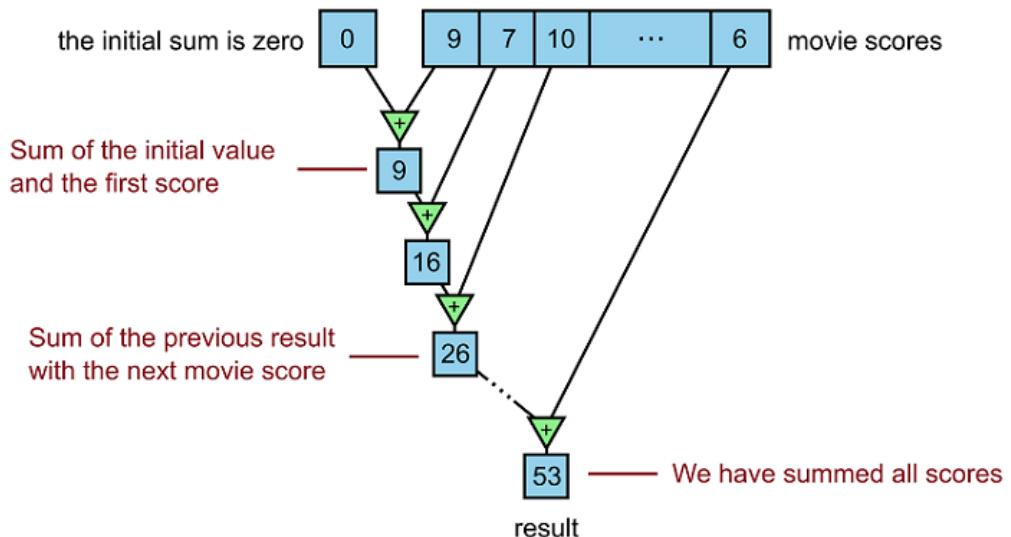
2.2.2 Folding

We often need to process a collection of items, one item at a time, in order to calculate something. The result might be as simple as a sum or a product of all items in the collection (as in our previous examples), or a number of all items that have a specific value, or that satisfy a predefined predicate. The result might also be something more complex like a new collection that contains only a part of the original one (filtering) or a new collection with the original items reordered (for sorting or partitioning).

The `std::accumulate` algorithm is an implementation of a general concept called *folding* (or *reduction*). Fold is a higher-order function that abstracts the process of iterating over recursive structures such as vectors, lists, trees, etc. that allows us to gradually build the result we need. In our previous example, we used folding to sum a collection of movie ratings. The `std::accumulate` algorithm first

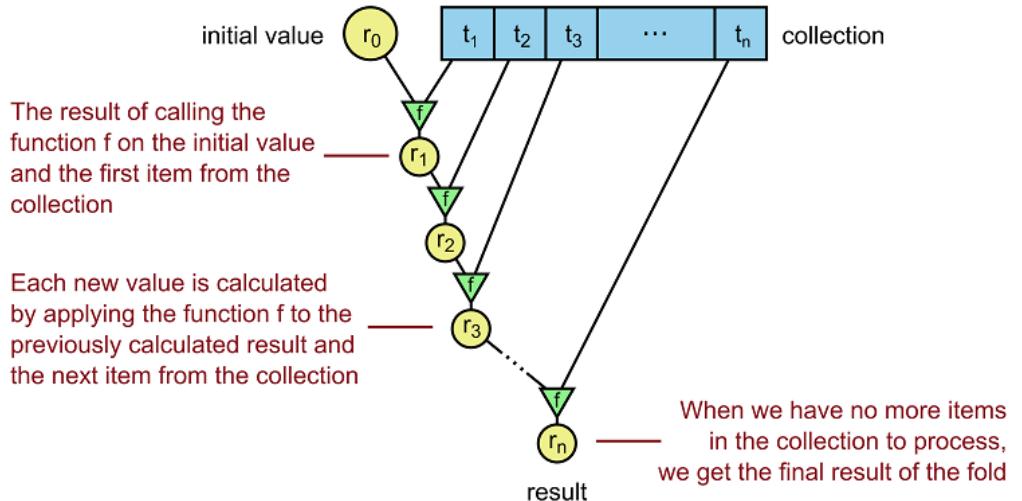
sums the initial value passed to it and the first item in the collection. That result is then summed with the next item in the collection, and so on. This is repeated until we reach the end of the collection (Figure 2.5).

Figure 2.5. Fold calculates the sum of all movie scores in a collection by adding the first score to the initial value used for summation (zero), then it adds the second score to the previously calculated sum, and so on.



In general, folding doesn't require that the arguments and the result of the binary function passed to it have the same type. Folding takes a collection that contains items of type τ , an initial value of type R (which doesn't need to be the same as τ) and a function $f: (R, \tau) \rightarrow R$. It calls the specified function on the initial value and the first item in the collection. The result is then passed to the function f along with the next item in the collection. This is repeated until we process all items from the collection. The algorithm returns the value of type R — the value that the last invocation of the function f returned (Figure 2.6).

Figure 2.6. In general, fold takes the first element in a collection and applies the specified binary function f to it and the initial value for folding (r_0), then it takes the second element from the collection, and calls f on it and the previously calculated result. This is repeated until we get to the last item in the collection.



In our previous example, the initial value was θ (or 1 in the case of multiplication), and the binary function f was addition (or multiplication).

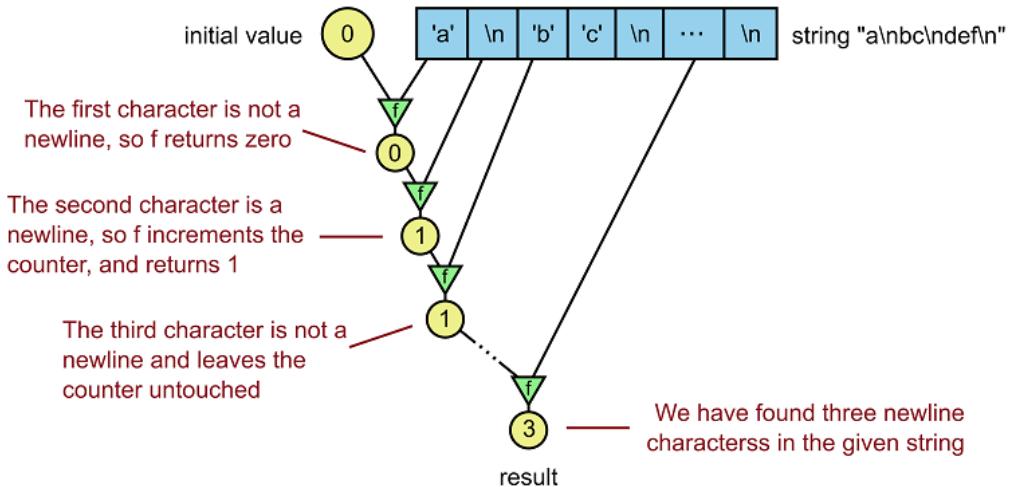
As a first example of when a function f might return a different type to the type of the items in the collection that we are folding over, we will implement a simple function that counts the number of lines in a string by counting the number of occurrences of the newline character in it by using `std::accumulate`. As a reminder, we have used `std::count` for this in the first chapter.

From the requirements of the problem, we can deduce the type of the binary function f that we will pass to the `std::accumulate` algorithm. The `std::string` is a collection of characters, so the type T will be a `char`, and since the resulting number of newline characters is an integer, the type R will be an `int`. This means that the type of the binary function f will be `(int, char) → int`.

We also know that the first time f is called, its first argument will be zero since we are just starting to count, and when it is invoked the last time it should return the final count as the result. Moreover, since f doesn't know whether it is currently processing the first, the last, or any other random character in the string, it can't use the character position in its implementation. It only knows the **value** of the current character, and the result that the previous call to f returned. This basically forces the meaning of *the number of newlines in the previously processed part of the string* to its first argument. From this point on, the implementation is straightforward — if the current character isn't a newline, return the same value it

was passed to us as the previous count, otherwise increment that value and return it ([Listing 2.4](#)). A concrete example for this evaluation is presented in the Figure 2.7 .

Figure 2.7. We can count the number of newline characters that appear in a given string by folding a string with a simple function that increments the count each time it gets a newline.



Listing 2.4. Implementing newline character counting

using std::accumulate (example:count-lines-using-accumulate/main.cpp)

```
int f(int previous_count, char c)
{
    return (c != '\n') ? previous_count
                       : previous_count + 1; ①
}

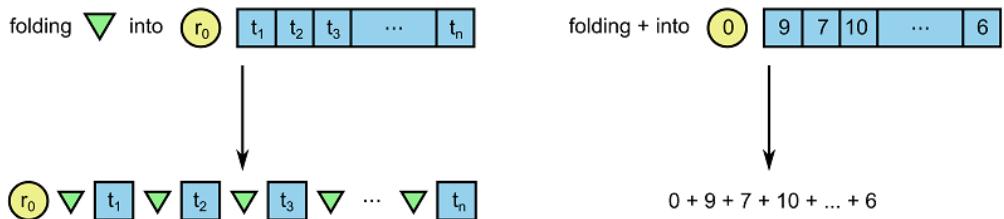
int count_lines(const std::string& s)
{
    return std::accumulate(
        s.cbegin(), s.cend(),
        0, ②
        f ③
    );
}
```

- ① Increasing the count if the current character is a newline
- ② Folding the whole string
- ③ Starting the count with zero

An alternative way to look at folding is to alter our perception of the binary function *f* passed to it. If we consider it to be a normal left-associative binary operator like '+' that we can write in the infix notation, then folding is equivalent

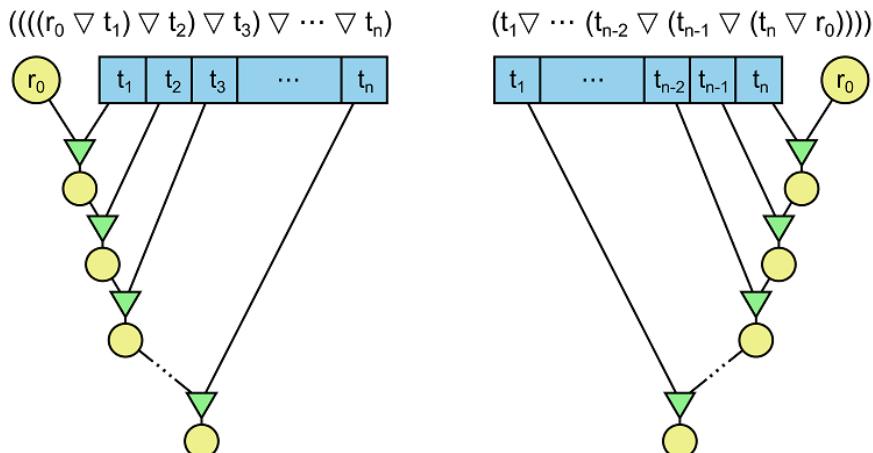
to putting this operator between every element in the collection. In the example of summing the movie scores, it would be equivalent to writing all the scores one after the other, and putting the '+' between them (Figure 2.8). This will be evaluated from left to right, just like the fold performed by `std::accumulate`.

Figure 2.8. Folding on a left-associative operator is the same as writing all the items in the collection one after another, and putting the operator between them.



This type of folding — where we start processing the items from the first item is called a *left fold*. There is also a *right fold* where the processing starts with the last element in a collection, and moves towards the beginning. Right fold corresponds to the evaluation of the infix notation seen in the Figure 2.8 when the operator is right-associative (Figure 2.9). C++ doesn't provide a separate algorithm for right fold, but we could pass reverse iterators (`crbegin` and `crend`) to `std::accumulate` to achieve the same effect.

Figure 2.9. The main difference between left and right folding is that left-fold starts from the first element and moves towards the end of the collection, while the right-fold starts at the end, and moves towards the beginning.



While folding seems really simple at first — summing a bunch of numbers in a collection — when we accept the idea that the addition can be replaced with

arbitrary binary operation that can even produce a result of a different type to the items in the collection, it becomes a real power-tool for implementing all different algorithms. We will see another concrete example later in this chapter.

Note**Associativity**

An operator ∇ is left-associative if $a \nabla b \nabla c = (a \nabla b) \nabla c$, and right-associative if $a \nabla b \nabla c = a \nabla (b \nabla c)$.

Mathematically, addition and multiplication are both left- and right-associative — it doesn't matter whether we start multiplying arguments from the left, or from the right side. But in C++, they are specified as left-associative so that the language can guarantee the order of evaluation for the expression.

Mind that this still doesn't mean that C++ guarantees the order of evaluation of the arguments themselves in most cases, it just guarantees in which order the operator will be applied to them. This means that in the expression $a * b * c$ (where a , b and c are nested expressions) we still don't know which one will be evaluated first, but we know exactly in which order the multiplication will be performed — the product of a and b will be multiplied by c .²

Most binary operators in C++ are left-associative (arithmetic, logical, comparison), while the assignment operators are right-associative (direct and compound assignments).

2.2.3 String trimming

There are many cases where we are given a string, and we need to strip the whitespace from its start and end. For example, we might want to read a line of text from a file, and write it centered on the screen. If we kept the leading and trailing spaces, it would appear misaligned.

The standard library doesn't have the functions to remove the whitespace from the beginning and the end of a string. But it does provide algorithms that we can use in order to implement something like this.

The algorithm we will need is `std::find_if`. It searches for the first item in the collection that satisfies the specified predicate. In our case, we are going to search for the first character in the string that isn't whitespace. The algorithm returns the iterator pointing to the first element in the string that satisfies the predicate. It is sufficient to remove all elements from the beginning up until that element, and we have stripped the whitespace from the start of the string.

```
std::string trim_left(std::string s)
{
    s.erase(s.begin(),
            std::find_if(s.begin(), s.end(), is_not_space));
    return s;
}
```

1

² C++17 does define the order of argument evaluation in some cases (see [wg21.link/P0145R3](#)) but it leaves the order of argument evaluation for operators like summing and multiplication unspecified.

As usual, the algorithm takes a pair of iterators to define the collection. If we pass it the reverse iterators, it will search from the end towards the beginning of the string. With this, we can trim the whitespace from the right.

```
std::string trim_right(std::string s)
{
    s.erase(std::find_if(s.rbegin(), s.rend(), is_not_space).base(),
            s.end());
    return s;
}
```

Note

Passing by value

In this case, it is better to pass the string by value instead of passing it as a const-reference, since we are modifying it, and returning the modified version.

Otherwise, we would need to create a local copy of the string inside the function. Which might be slower if the user calls the function with a temporary value (r-value) which would be copied instead of just being moved into our function. If the user doesn't pass a temporary, it will be copied in both cases, so in that case both approaches are the same.

The only downside of passing by value in this case is that if the copy-constructor for the string throws an exception, the backtrace might be confusing to the user of the function.

For more information about copying and moving, see www.cprogramming.com/c11/rvalue-references-and-move-semantics-in-c11.html

Composing these two functions gives us the full `trim` function:

```
std::string trim(const std::string& s)
{
    return trim_left(trim_right(s));
}
```

This example showed us one of the possible uses for the `std::find_if` higher-order function. We used it to find the first non-white character searching from the start of the string, and the first one from the end of the string. We have also seen how a composition of two smaller functions (`trim_left` and `trim_right`) gives us a function that solves our original problem.

2.2.4 Partitioning collections based on a predicate

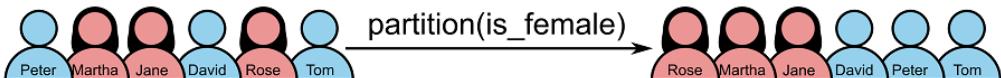
Before we go to something more involved, imagine we have a collection of people like in the example from the beginning of the chapter, but our task is to move all females to the start of the collection.

For this, we have the `std::partition` algorithm, and its variant `std::stable_partition`. Both of these algorithms take a collection and a predicate. They reorder the items in the original collection so that the items that satisfy the specified predicate get separated from those that don't. The items that satisfy the predicate are moved to the beginning of the collection, and the items that don't are

moved to the back. The algorithm returns an iterator to the first element in the second group (to the first element that doesn't satisfy the predicate). The returned iterator can be paired with the begin iterator of the original collection to provide a collection of elements that satisfy the predicate, or paired with the end iterator of the original collection to provide a collection of all elements which don't satisfy the predicate. This is true even if any of these collections are empty.

The difference between these two algorithms is that the `std::stable_partition` retains the ordering between the elements from the same group (note that in the Figure 2.10 we have used `std::partition` and *Rose* ended up in front of *Martha* even though she was after her in the original list).

Figure 2.10. Partitioning the collection of items. We are partitioning a group of people based on a predicate that checks whether a person is a female or not. As the result, all females will be moved to the start of the collection.



Listing 2.5. Females first

```
std::partition(
    people.begin(), people.end(),
    is_female
);
```

While this example might seem contrived, imagine you have a list of items in a user interface. The user can select them and drag the selected ones to a specific place.

If that place is at the beginning of the list, then we have an equivalent problem to the previous one, but instead of moving females to the beginning, we are moving the selected items. If we should move the selected items to the end, it is equivalent to moving those that aren't selected to the top.

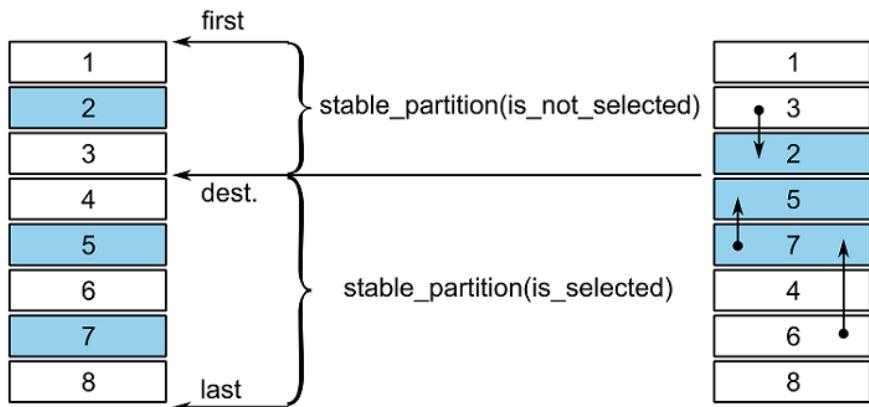
If we need to move the selected items to some place in the middle of the list, we can split the list to the part above the destination point, and the part below. Then, we have one list in which the selected items should go to the bottom, and one list in which they should go to the top.

When the users drag items around, they expect to have the order amongst the selected ones kept intact, and the same goes for those not selected. Moving a few items around shouldn't arbitrarily change the order of the rest of the list. Because of this, we need to use the `std::stable_partition` algorithm instead of `std::partition` even if the latter is more efficient.

Listing 2.6. Moving selected items to a specific point (example:move-selected/main.cpp)

```
std::stable_partition(first, destination, is_not_selected);
std::stable_partition(destination, last, is_selected);
```

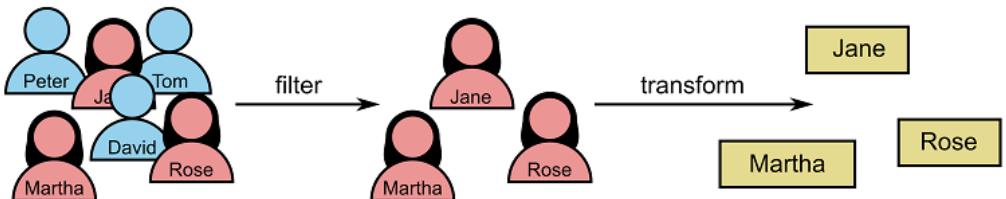
Figure 2.11. Moving selected items to a specified position with the help of std::stable_partition algorithm. We want to move the selected items that are above the destination location downwards, and those that come after the destination location upwards.



2.2.5 Filtering and transformation

Let's try to implement the problem from the beginning of this chapter using STL algorithms. Just a short reminder, the task is to get the names of all females in a given group of people.

Figure 2.12. Getting the names of all females in a group



A person will have the type `person_t`, and we are going to use the `std::vector` as the collection type. In order to make the implementation simpler, let's say we have `is_female`, `is_not_female` and `name` as non-member functions:

```
bool is_female(const person_t& person);
bool is_not_female(const person_t& person);
std::string name(const person_t& person);
```

As we saw earlier, the first thing we need to do is to filter the collection in order to get a vector that contains only females. We have two ways of achieving this using STL algorithms. If we are allowed to change the original collection, we can use the `std::remove_if` algorithm with the erase-remove idiom to remove all persons that aren't females (Listing 2.7).

Listing 2.7. Filtering items by removing undesired ones from the collection (example:filtering-using-remove-if/main.cpp)

```
people.erase(
    std::remove_if(people.begin(), people.end(),
                  is_not_female),
    people.end());
```

- ① "marking" the items for removal
- ② Removing the marked items

The erase-remove idiom

If we want to use the STL algorithms to remove elements that satisfy a predicate, or all elements that have a specific value from a collection, we can do so with the `std::remove_if` and `std::remove` algorithms.

Unfortunately, since these algorithms operate on a range of elements defined by a pair of iterators, and do not know what the underlying collection is, they can not actually remove the elements from it. These two algorithms only move all the elements that do not satisfy the criteria for removal to the beginning of the collection.

Other elements are left in an unspecified state, and the algorithm returns the iterator to the first of them (or to the end of the collection if there are no element that should be removed). We need to pass this iterator to the `.erase` member function of the collection which performs the actual removal.

Alternatively, if we don't want to change the original collection, which should be the right way to go since we aim to be as pure as possible, we can use the `std::copy_if` algorithm to copy all the items that satisfy the predicate we are filtering on to the new collection (Listing 2.8). The algorithm requires us to pass it a pair of iterators that define the input collection, one iterator that points to the destination collection we want to copy the items to, and a predicate that returns whether a specific item should or shouldn't be copied. Since we don't know the number of females in advance (we can try to predict based on statistical data, but that is out of the scope of this example), we need to create an empty vector `females` and use the `std::back_inserter(females)` as the destination iterator.

Listing 2.8. Filtering items by copying them to a new collection (example:filter-and-transform/main.cpp)

```
std::vector<person_t> females; ①
std::copy_if(people.cbegin(), people.cend(), ②
             std::back_inserter(females), ②
             is_female); ②
```

- ① Creating a new collection to store the filtered items in
- ② Copying the items that satisfy the condition to the new collection

The next step is to get the names of the people in the filtered collection. We can use `std::transform` for this. We need to pass it the input collection as a pair of iterators, the transformation function and where the results should be stored. In this case, we know that the number of names is the same as the number of females, so we can immediately create the `names` vector to be of the needed size to store the result and use `names.begin()` as the destination iterator instead of relying on `std::back_inserter`. This way, we will remove the potential memory reallocations needed for vector resizing.

Listing 2.9. Getting the names (example:filter-and-transform/main.cpp)

```
std::vector<std::string> names(females.size()); ①
std::transform(females.cbegin(), females.cend(), ②
              names.begin(), ③
              name); ④
```

- ② What we are transforming
- ③ Where to store the results
- ④ The transformation function

What we did here is to split a larger problem into two smaller ones. This way, instead of creating one highly specialized function that would be of limited use in our program, we created two separate, and much more broadly applicable ones. One that filters a collection of people based on a predicate, and one that retrieves the names of people in a collection. The code reusability is much higher in this case.

2.3 Composability problems of STL algorithms

This solution is valid and will work correctly for any type of the input collection that can be iterated on — from vectors and lists, to sets, hash maps and trees. It also shows the exact intent of the program — to copy all females from the input collection, and then get the names for all of them.

Unfortunately, it isn't as efficient nor as simple as a hand-written loop:

```
std::vector<std::string> names;

for (const auto& person : people) {
    if (is_female(person)) {
        names.push_back(name(person));
    }
}
```

The STL-based implementation makes unnecessary copies of `person` (which might be an expensive operation, or it could even be disabled if the copy constructor is deleted or private) and it creates an additional vector which isn't actually needed. We could try to compensate for these problems by trying to use references or pointers instead of copies, or by creating smart iterators that will skip all persons that aren't females, etc. but the need to do this extra work is a clear indication that the STL has lost in this case — the hand-written loop is just better and requires less effort.

So, what went wrong? If you recall the signatures for `transform` and `filter` that we saw earlier, you can see that they are designed to be composable — that is, to call one on the result of the other.

```
filter    : (collection<T>, (T → bool)) → collection<T>
transform : (collection<T>, (T → T2)) → collection<T2>

transform(filter(people, is_female), name)
```

The `std::copy_if` and `std::transform` algorithms don't fit into these types at all. They do require the same basic elements, but in a significantly different manner. They take the input collection through a pair of input iterators, which makes it impossible to invoke them on a result of a function that returns a collection, without first saving that result to a variable. And they don't return the transformed collection as the result of the algorithm call, but they also require the iterator to the output collection to be passed in as the algorithm argument. Instead, the result of the algorithm is an output iterator to the element in the destination range, one past the last element stored.

```
OutputIt copy_if(InputIt first, InputIt last,
                  OutputIt destination_begin,
                  UnaryPredicate pred);

OutputIt transform(InputIt first, InputIt last,
                  OutputIt destination_begin,
                  UnaryOperation transformation);
```

This effectively kills the ability to compose these two algorithms without creating intermediary variables like we saw earlier.

While this might seem like a design problem that shouldn't have happened, there are practical reasons why these *problems* exist. First, passing a collection as a pair of iterators allows us to invoke the algorithm on parts of a collection instead of on the whole. Passing the output iterator as an argument instead of the algorithm returning the collection allows us to have different collection structures for input and output (for example, if we wanted to collect all the *different* female names, we might pass the iterator to a `std::set` as the output iterator).

Even having the result of the algorithm to be an iterator to the element in the destination range, one past the last element stored has its merits. If we wanted to create an array of all people, but order them in a way to have females first, and then others (like the `std::stable_partition` algorithm, but to create a new collection instead of changing the old one), it would be as simple as calling `std::copy_if` twice, once to copy all the females and once to get everyone else. The first call will return us the iterator to the place where we should start putting non-females (Listing 2.10).

Listing 2.10. Females first

```
std::vector<person_t> separated(people.size());

const auto last = std::copy_if(
    people.cbegin(), people.cend(),
    separated.begin(),
    is_female); ❶

std::copy_if(
    people.cbegin(), people.cend(),
    last, ❷
    is_not_female);
```

- ❶ Returns the location after the last person copied
- ❷ Storing the rest after all females

So, while the standard algorithms do provide us a way to write our code in a more functional manner than doing everything with hand-written loops and branches, they aren't designed to be composed in a way common to other functional programming libraries and languages. We will see a more composable approach that is taken in the ranges library in the chapter 7.

2.4 Writing your own higher order functions

While there are many algorithms already implemented in the standard library, or in some of the freely available third-party libraries like `boost`, we often need to implement something domain specific. Sometimes, we just need a specialized version of a standard algorithm, but sometimes we need to implement something from scratch.

2.4.1 Receiving functions as arguments

In the previous example, we wanted to get the names of all females in our collection. Say we often have the need to get the names of persons in a collection that satisfy some predicate, but we don't want to limit ourselves to use only a predefined predicate like `is_female`. We want to support any predicate that takes a `person_t` which the user can throw at us. The user might want to separate the people based on their age, hair colour, marital status etc.

It would be useful to create a function for this that we can call multiple times. The function would need to accept a vector of people and a predicate function that will be used for filtering, and it will return a vector of strings containing the names.

This example might seem overly specific. It would be much more useful if it could work with any type of collection, and any collected type, and not only vectors of people. Also it would be better if it could extract any kind of data from the person and not only the name. We could make it more generic, but we will keep simple in order to keep the focus on what is important here.

We know that we should pass the vector of people as a const-reference, but what to do with the function? We could use function pointers, but that would be overly limiting. In C++, many things can behave like functions, and we have no universal type that would be able to hold any function-like thing without incurring performance penalties. Instead of trying to guess which type would be the best, we can just make the function type to be the template parameter, and leave it up to the compiler to figure the exact one.

```
template <typename FilterFunction>
std::vector<std::string> names_for(
    const std::vector<person_t>& people,
    FilterFunction filter)
```

This will allow the user to pass in anything that can behave like a function, and we will be able to call it as if it was an ordinary function. We will cover all the different things that C++ considers function-like in the next chapter.

2.4.2 Implementing with loops

We have seen how we could implement the `names_for` function using STL algorithms. While it is advised to use STL algorithms whenever it is possible, it isn't a rule that can't be broken. But if we do break it, we should be aware that we are opening ourselves to the possibility to have more bugs in our code. And we should have a strong reason to do so. In this case, since it would incur unnecessary memory allocations, it might be smarter if we just implemented it as a hand-written loop.

Note **Ranges to the rescue**

The issue of unnecessary memory allocations arises from the composability problems of STL algorithms. This problem with STL algorithms has been known for some time, and there are a few libraries created to fix this.

We will cover a concept called *ranges* in detail in chapter 7. Ranges are currently published as a Technical Specification and are planned for inclusion into C++20. Before ranges become a part of the standard library, they are available as a third-party library which can be used with most C++11-compatible compilers.

Ranges will allow us to have our cake and eat it too, by creating composable smaller functions, without having any performance penalties for doing so.

Listing 2.11. Implementing using a hand-written loop

```
template <typename FilterFunction>
std::vector<std::string> names_for(
    const std::vector<person_t>& people,
    FilterFunction filter)
{
    std::vector<std::string> result;

    for (const person_t& person : people) {
        if (filter(person)) {
            result.push_back(name(person));
        }
    }

    return result;
}
```

Implementing a function that is as simple as this one using a hand-written loop isn't a huge issue. The names of the function and its parameters are sufficient to make it obvious what the loop does. In fact, most STL algorithms are implemented as loops, just like this function.

Now, if STL algorithms are implemented as loops, why is it bad for us to implement everything with loops. Why bother learning STL at all?

There are several reasons. The first one is simplicity. It is time-saving to use the code that somebody else wrote. It also leads to the second big benefit—correctness. If you need to write the same thing over and over again, it isn't unreasonable to expect that one time you will slip up and make a mistake. STL algorithms are thoroughly tested and will work correctly for any given input. This is the same reason why we might allow ourselves to implement often used functions with hand-written loops — they will often be tested.

While many of STL algorithms aren't pure, they are built as higher-order functions, and this allows them to be more generic, and thus applicable to more problems.

And if something is used often, it is less likely to contain previously unseen bugs.

2.4.3 Recursion and tail call optimization

The previous solution is pure from the outside, but its implementation isn't. We are changing the result vector every time we find a new person that matches the criteria. In pure functional programming languages, where the loops don't exist, functions that iterate over collections are usually implemented using recursion. We aren't going to dive too much into recursion, because we aren't going to use it that often, but we need to cover a few important things.

We can process a non-empty vector recursively by first processing its head (the first element), and then by processing its tail (all other elements) which can also be seen as a vector. If the head satisfies the predicate, we will include it in the result. If we get an empty vector, there is nothing to process, so we are also returning an empty vector.

In order to make the code sample shorter, let's pretend we have a function `tail` that takes a vector, and returns us its tail, and that we have the `prepend` function which takes an element and a vector, and returns us copy of the original vector, with the specified element prepended to it.

Listing 2.12. Naive recursive implementation

```
template <typename FilterFunction>
std::vector<std::string> names_for(
    const std::vector<person_t> &people,
    FilterFunction filter)
{
    if (people.empty()) { ①
        return {};
    } else {
        const auto head = people.front();
        const auto processed_tail = names_for( ②
            tail(people),
            filter);

        if (filter(head)) { ③
            return prepend(name(head), processed_tail);
        } else {
            return processed_tail;
        }
    }
}
```

① If the collection is empty, returning an empty result

② Calling the function recursively to process the tail of the collection

③ If the first element satisfies the predicate, include it in the result. Otherwise, skip it.

Now, this implementation is really inefficient. First, there is a reason why the `tail` function for vectors doesn't exist. It would require creating a new vector and copying all the data from the old one into it (apart from the first element). The problem with the `tail` function can be remedied by using a pair of iterators instead of a vector as the input. In that case, getting the tail becomes trivial — we just have to move the iterator that points to the first element.

Listing 2.13. Recursive implementation

```
template <typename FilterFunction, typename Iterator>
std::vector<std::string> names_for(
    Iterator people_begin,
    Iterator people_end,
    FilterFunction filter)
{
    ...
    const auto processed_tail = names_for(
        people_begin + 1,
        people_end,
        filter);
    ...
}
```

The second issue we have in this implementation is that we are prepending items to the vector. We can't do much about that — we also had this in the hand-written loop, albeit with appending which is more efficient than prepending when vectors are concerned.

The last issue, and probably the most important one is that we can get into problems if we call our function with a large collection of items. Every recursive call will take some memory on the stack, and at some point the stack will overflow and our program will crash. Also, even if the collection isn't big enough to induce the stack to overflow, function calls aren't free — simple jumps that a `for` loop gets compiled to are significantly faster.

While the previous issues were easily fixed, this one isn't. Here we need to rely on our compiler to be able to transform the recursion into loops. And in order for the compiler to be able to do so, we need to implement a special form of recursion called *tail recursion*. Tail recursion is a kind of recursion where the recursive call is the last thing in the function, that is, the function mustn't do anything at all after recursing.

In our example, the function isn't tail-recursive because we are getting the result of the recursive call and, in the case where `filter(head)` is true, we are adding an element to it, and only then returning as the result.

Changing the function to be tail-recursive isn't as trivial as the previous changes were. Since the function must return the final result, we must find another way to

collect the intermediary results. We will have to do it through an additional argument that we will pass on from call to call.

Listing 2.14. Tail-recursive implementation (example:filter-and-transform-combined/main.cpp)

```
template <typename FilterFunction, typename Iterator>
std::vector<std::string> names_for_helper(
    Iterator people_begin,
    Iterator people_end,
    FilterFunction filter,
    std::vector<std::string> previously_collected)
{
    if (people_begin == people_end) {
        return previously_collected;
    } else {
        const auto head = *people_begin;

        if (filter(head)) {
            previously_collected.push_back(name(head));
        }

        return names_for_helper(
            people_begin + 1,
            people_end,
            filter,
            std::move(previously_collected));
    }
}
```

Now, we are returning an already calculated value, or we are returning exactly what the inner recursive call has returned to us. The small problem is that we need to call this function with an additional argument. Because of this, we have named it `names_for_helper` and the main function can be trivially implemented by calling the `helper` function and passing it an empty vector for the `previously_collected` parameter.

Listing 2.15. Calling the helper function

```
template <typename FilterFunction, typename Iterator>
std::vector<std::string> names_for(
    Iterator people_begin,
    Iterator people_end,
    FilterFunction filter)
{
    return names_for_helper(people_begin,
                           people_end,
                           filter,
                           {});
```

```
}
```

In this case, the compilers that support the tail-call optimization (TCO) will be able to convert the recursive function to a simple loop and make it as efficient as the code we started with.

Tail-call optimization

The C++ standard makes no guarantees that the tail-call optimization will be performed. However, most modern compilers including GCC, Clang and MSVC, support this.

They are sometimes even able to optimize mutually-recursive calls (when function a calls function b, and b also calls a) and some functions that aren't strictly tail-recursive functions, but are close enough.

Recursion is a powerful mechanism that allows implementing iterative algorithms in the languages that don't have loops. But recursion is still a really low-level construct. We can achieve true inner purity with it, but in many cases it just makes no sense. We said that we want to be pure so that we can make less mistakes while writing the code, since we don't need to think about the mutable state. But, by writing the tail-recursive functions like the above one, we are simply simulating the mutable state and loops in another way. And while it is a nice exercise for the gray cells, it isn't much more than that.

Recursion, like hand-written loops, has its place in the world, but in most cases in C++, it should raise the red flag during the code review. It needs to be checked for correctness, and that it covers all the use-cases without ever overflowing the stack.

2.4.4 Implementing using folds

Since recursion is a low level construct, implementing it manually is often avoided even in pure functional programming languages. Some might say that the higher level constructs are especially popular in FP exactly because recursion is this convoluted.

We have seen what folding is, but we still don't know its roots. We saw that it takes one element at a time, and applies the specified function to the previously accumulated value and the currently processed element, which produces the new accumulated value. If you haven't skipped the previous section, you have probably noticed that this is exactly what our tail-recursive implementation of `names_for_helper` function did. In essence, folding is nothing more than a nicer way to write tail-recursive functions that iterate over a collection — the common parts are abstracted out, and you need to specify only the collection, the starting value, and the essence of the accumulation process without the need to write recursive functions.

If we wanted to implement our `names_for` function with folding (`std::accumulate`),

it would be trivial now that we know how to implement it in a tail-recursive way. We start with an empty vector of strings, we append a new name if the person satisfies the predicate, and that is it.

Listing 2.16. Implementation using fold

```
std::vector<std::string> append_name_if(
    std::vector<std::string> previously_collected,
    const person_t& person)
{
    if (filter(person)) {
        previously_collected.push_back(name(person));
    }
    return previously_collected;
}

...

return std::accumulate(
    people.cbegin(),
    people.cend(),
    std::vector<std::string>{},
    append_name_if);
```

Accumulate makes too many copies

If you run the above function on a large collection, you will see that it is quite slow. The reason for this is that `append_name_if` receives a vector by-value, which induces a copy whenever `std::accumulate` calls it and passes it the value it accumulated so far.

This can be easily fixed by rolling out our own variant of `std::accumulate` which will trigger the C++ move semantics instead of creating copies. You can find the implementation of `moving_accumulate` in the accompanying examples.

Folding is a powerful concept. So far, we have seen that it is expressive enough to implement counting, transformation (or mapping) and filtering (the last example implemented a combination of transform and filter). It can be used to implement more than a few standard algorithms. It would be a nice exercise to implement algorithms

like `std::any_of`, `std::all_of` and `std::find_if` using `std::accumulate` and `check` whether these implementations can be as fast as the original algorithms, and if not investigate why.

2.5 Summary

- By passing specific predicate functions to algorithms like `transform` and `filter`, we are changing their behaviour and making a general algorithm work for our exact problem.

- The `std::accumulate` algorithm lives in the `<numeric>` header, but is applicable to many more areas than just performing simple calculations.
- It implements a concept called folding which isn't limited only to addition and multiplication, but can even be used to implement standard algorithms like `count_if` and `find_if`. It would be a nice exercise to implement an algorithm like insertion sort using `std::accumulate`.
- If we don't want to change the order of selected elements in a list while dragging them around, it is better to use the `std::stable_partition` algorithm instead of the ordinary `std::partition`. In similar cases, when user interface is concerned, `std::stable_sort` is often a preferable choice over `std::sort`.
- While STL algorithms are meant to be composed, they aren't meant to be composed in the same way as algorithms in other functional programming languages. This will be easily remedied once we start using ranges.
- The fact that the most standard algorithms are implemented as loops isn't a reason to use hand-written loops instead of them. In the same way `while`, `if` and others are really implemented using `gotos(jumps)` but we aren't using `goto` statements because we know better.

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch02

3

Functional objects

This chapter covers:

- Ordinary functions, pointers and references to functions
- Different things that can be used as functions in C++
- Creating generic function objects
- What are lambda expressions, and how they relate to normal function objects
- What new things did C++14 bring to lambdas
- Creating prettier function objects with Boost.Phoenix, and by hand
- What is `std::function` and when to use it

We have seen how to create functions that can accept other functions as their arguments in the previous chapter. Now we need to check out the other side of the coin — to see all the things that we can use as functions in C++. This is a bit dry topic, but it is necessary for the deeper understanding of functions in C++ and for achieving that sweet-spot of using higher level abstractions without incurring performance penalties. If we want to be able to use all the power that C++ gives us during our functional programming adventures, we need to know what all the different things that C++ can treat as functions are, and which of them to prefer, and which of them to avoid.

If we use duck-typing (if it walks like a duck, and quacks like a duck, it is a duck), we can say that anything that can be called like a function, is a function object. Namely, if we can write a name of an entity, followed by arguments in parenthesis like `f(arg1, arg2, ..., argn)` that entity is a function object. We will cover all the

different things that C++ considers to be function objects in this chapter.

Note

"Function" versus "function object"

In order to differentiate between ordinary C++ functions, and all the things that can be used as functions, we will call the later *function objects* if we want to make an explicit distinction.

3.1 Functions and function objects

A function is a named group of statements that can be invoked from other parts of the program, or from the function itself in the case of recursive functions. C++ provides several slightly different ways to define a function:

```
// The 'old' C-like syntax
int max(int arg1, int arg2) { ... }

// With a trailing return type
auto max(int arg1, int arg2) -> int { ... }
```

While some people prefer to write functions with the trailing return type even in the cases when it isn't necessary, it isn't a common practice. This syntax is mainly useful when writing template functions where the return type depends on the types of the arguments.

Trailing return type definition

In the above case, it is the same whether the return type is specified before the function name and its arguments, or after.

Specifying the return type after the function name and the arguments is required when we are writing a template function and we need to deduce the return type based on the types of the arguments.

Some people prefer use the trailing return type always because they consider it to be much less important than the function name and its arguments, so that it should come after. While this notion is valid, the traditional approach is still omnipresent.

3.1.1 Automatic return type deduction

Since C++14, it is also allowed to omit the return type and to rely on the compiler to deduce it automatically from the expression in the `return` statement. The type deduction in this case follows the rules of template argument deduction.

In the following example, we have an integer named `answer` and two functions `ask1` and `ask2`. Both functions have the same body—they just return the `answer`. The difference is that they have the return types specified differently. The first one returns a value whose type is automatically deduced, and the second one returns a const-ref to a type that is automatically deduced. The compiler will check the type of the variable we have passed to the `return` statement, which is `int`, and seemingly replace the keyword `auto` with it.

Listing 3.1. Leaving it up to the compiler to deduce the return type

```
int answer = 42;
auto ask1() { return answer; }      ①
const auto& ask2() { return answer; } ②
```

- ① return type is int
 ② return type is const int&

While the template argument deduction (and therefore the return type deduction) rules are more complex than just '*replace the `auto` keyword with a type*', they behave quite intuitively, and we aren't going to cover them here.³

In the case of functions with multiple `return` statements, all of them need to return results of the same type. If the types differ, the compiler will report an error, even if one of them can be implicitly cast to the other. In the following snippet, we have a function that takes a Boolean flag, and depending on the value of that flag returns an `int`, or a `std::string`. The compiler will complain that it first deduced the return type to be an integer, and that the second `return` statement returns a different type.

Listing 3.2. Error when returning different types

```
auto ask(bool flag)
{
    if (flag) return 42;
    else      return std::string("42"); ①
}
```

- ① error: inconsistent deduction for 'auto': 'int' and then 'std::string'

Once the return type is deduced, it can be used in the rest of the function. This allows writing recursive functions with automatic return type deduction.

Listing 3.3. Recursive function with automatic return type deduction

```
auto factorial(int n)
{
    if (n == 0) {
        return 1;          ①
    } else {
        return factorial(n - 1) * n; ②
    }
}
```

³ You can find a detailed explanation of template argument deduction rules at en.cppreference.com/w/cpp/language/template_argument_deduction

- 1 deducing the return type to be `int`
- 2 it is already known that `factorial` returns an `int`, and multiplying two `int`s returns an `int`, so we are OK.

Swapping the `if` and `else` branches would result in an error because the compiler would arrive at the recursive call to `factorial` before the return type of the function was deduced. This means that when writing recursive functions, it is necessary either to specify the return type, or to first write the return statements that don't recurse.

As an alternative to `auto` which uses the template argument type deduction rules for the resulting type, it is also possible to use `decltype(auto)` as the return type specification. In that case, the function return type will be `decltype` of the returned expression.

Listing 3.4. Using decltype(auto) to specify the return type

```
decltype(auto) ask() { return answer; }      1
decltype(auto) ask() { return (answer); }      2
decltype(auto) ask() { return 42 + answer; }    3
```

- 1 returns an `int` — `decltype(answer)`
- 2 returns a reference to `int` — `decltype((answer))`, whereas the `auto` would deduce just `int`
- 3 returns an `int` — `decltype(42 + answer)`

This is useful when writing generic functions that just forward the result of another function without modifying it. In this case, we don't know in advance what function will be passed to us, and we can't know whether we should pass its result back to the caller as a value, or as a reference. If we pass it as a reference, it might return a reference to a temporary value which will produce undefined behaviour. And if we pass it as a value, it might make an unnecessary copy of the result. Copying will have performance penalties, and will sometimes even be semantically wrong — the caller might expect a reference to an existing object.

If we want to perfectly-forward the result, that is, to return the result returned to us without any modifications, we can use the `decltype(auto)` as the specification of the return type.

Listing 3.5. Perfect forwarding of the function result

```
template <typename Object, typename Function>
decltype(auto) call_on_object(Object&& object, Function function)
{
    return function(std::forward<Object>(object));
}
```

In the above code snippet, we have a simple template function that takes an object,

and a function that should be invoked on that object. We are perfectly forwarding the `object` argument to the `function`, and by using the `decltype(auto)` as the return type, we are perfectly forwarding the result of the `function` back to our caller.

PERFECT FORWARDING FOR ARGUMENTS

We sometimes need to write a function that wraps in another function — the only thing it should do is to call the wrapped function with some arguments modified, added, or removed. In that case, we have the problem of how to pass the arguments from the wrapper to the function we need to call.

If we have the same setup as in the Listing 3.5, the user is passing us a function we know nothing about. We don't know how it expects us to pass it the argument.

The first option we have is to make our `call_on_object` function accept the `object` argument by-value, and pass it on to the wrapped function. This will lead to problems if the wrapped function accepts a reference to the object because it needs to change it. The change won't be visible outside of the `call_on_object` function since it will be performed on the local copy of the object that exists only in the `call_on_object` function.

```
template <typename Object, typename Function>
decltype(auto) call_on_object(Object object,
                             Function function)
{
    return function(object);
}
```

The second attempt is to pass the object by-reference. This would make the changes to the `object` visible to the caller of our function. But we will have a problem if `function` doesn't actually change the `object`, but accepts the argument as a `const`-reference. The caller won't be able to invoke `call_on_object` on a constant object, nor on a temporary. So, the idea to accept the object as an ordinary reference still isn't right.

And we can't make that reference `const`, since the `function` might want to change the original `object`.

Some pre-C++11 libraries approached this problem by creating overloads for both `const` and non-`const` references. This isn't practical because the number of overloads grows exponentially with the number of arguments that need to be forwarded.

This is solved in C++11 with the forwarding references (formerly known as *universal references*, as named by Scott Meyers). A forwarding reference is written as a *double reference* on a templated type. In the following code, the `fwd` argument is a forwarding reference to type `T`, while `value` isn't (it is a normal *rvalue reference*).

```
template <typename T>
void f(T&& fwd, int&& value) { ... }
```

The forwarding reference allows us to accept both const and non-const objects, and temporaries. Now, we only need to pass that argument on, and pass it on in the same value-category as we have received it, which is exactly what `std::forward` does.

3.1.2 Function pointers

A function pointer is a variable that stores the address of a function that can later be called through that pointer. They are a low-level construct C++ inherited from the C programming language that allows writing polymorphic code. The (runtime) polymorphism is achieved simply by changing which function the pointer points to, thus changing the behaviour when that function pointer is called.

Function pointers (and references) are also function objects, since they can be called like ordinary functions. Additionally, all types that can be implicitly converted to a function pointer are also function objects, as can be seen in the Listing 3.6, but this should be avoided. We should prefer proper function objects (which we will see in a few moments) since they are more powerful and easier to deal with. Objects that can be converted to function pointers can be useful when we need to interface with a C library that takes a function pointer and we need to pass it something more complex than a simple function.

Listing 3.6. Calling function pointers, function references and objects convertible to function pointers

```
int ask() { return 42; }

typedef decltype(ask)* function_ptr;

class convertible_to_function_ptr {
public:
    operator function_ptr() const
    {
        return ask;
    }
};

int main(int argc, char* argv[])
{
    auto ask_ptr = &ask;
    std::cout << ask_ptr() << '\n';           ①

    auto& ask_ref = ask;
    std::cout << ask_ref() << '\n';           ②

    convertible_to_function_ptr ask_wrapper;  ③
```

```
    std::cout << ask_wrapper() << '\n';
}
```

③

- ① A pointer to the function
- ② A reference to the function
- ③ An object that can be implicitly converted to a function pointer
- ④ The casting operator can only return a pointer to a function. While it can return different functions depending on some condition, it can't pass any data to them (without resorting to dirty tricks).

The Listing 3.6 demonstrates that we can create a function pointer (`ask_ptr`) that points to an ordinary function, a function reference (`ask_ref`) that references the same function, and that we can call them as if they were functions themselves — with the usual function call syntax. It also demonstrates that we can create an object that is convertible to a function pointer and call that object as if it was a normal function without any complications.

3.1.3 *Call operator overloading*

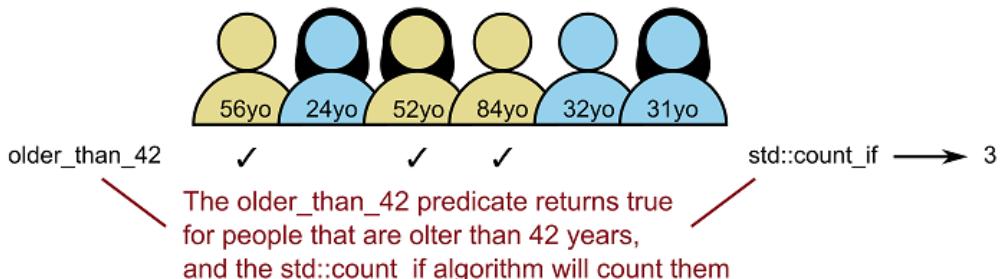
Instead of creating types that can be implicitly converted to function pointers, C++ allows a much nicer way to create new types that behave like functions — by creating classes and overloading their call operators. Unlike other operators, the call operator can have an arbitrary number of arguments, and the arguments can have any type, so we can create a function object of any signature we like. The syntax for overriding the call operator is as simple as defining a member function — just with a special name of `operator()` — we specify the return type, and all the arguments that the function needs.

Listing 3.7. Overloading the call operator

```
class function_object {
public:
    return_type operator()(arguments)
    {
        ...
    }
};
```

These function objects have one advantage compared to the ordinary functions — each instance can have their own state, be it mutable or not. The state is used to customize the behaviour of the function, without the caller having to specify it.

Figure 3.1. We can use an ordinary function that checks whether a person is more than 42 years old when we want to count how many people are above that age limit. But with this approach, we have the ability to check the age against a single predefined value.



Say we have a list of people like in the examples from the previous chapter. Each person has a name, age and a few things that we don't care about at the moment. Now, we want to allow the user to count the number of people that are older than some specified age. If the age limit was fixed, we could create an ordinary function that checks whether the person is older than that predefined value.

```
bool older_than_42(const person_t& person)
{
    return person.age > 42;
}

std::count_if(persons.cbegin(), persons.cend(),
              older_than_42);
```

This solution isn't scalable because we would need to create separate functions for all different age limits that we need, or to employ some error-prone hack like having the age limit we are currently testing against saved in a global variable.

Instead, it would be much wiser to create a proper function object that keeps the age limit as its inner state. This will allow us to write the implementation of the predicate only once, and instantiate it several times for different age limits.

```
class older_than {
public:
    older_than(int limit)
        : m_limit(limit)
    {}

    bool operator()(const person_t& person) const
    {
        return person.age() > m_limit;
    }

private:
```

```
    int m_limit;
};
```

We can now define multiple variables of this type and use them as functions:

```
older_than older_than_42(42);
older_than older_than_14(14);

if (older_than_42(person)) {
    std::cout << person.name() << " is more than 42 years old\n";
} else if (older_than_14(person)) {
    std::cout << person.name() << " is more than 14 years old\n";
} else {
    std::cout << person.name() << " is 14 years old, or younger\n";
}
```

Note **Function objects terminology**

Classes with the overloaded call operator are often called *functors* in various places. This term is highly problematic because it is already used for something quite different in the category theory. While we could ignore this fact, the category theory is important for functional programming (and programming in general), so we will keep calling them function objects.

The `std::count_if` algorithm doesn't care what we have passed to it as a predicate function, as long as it can be invoked like a normal function — and by overloading the call operator, we did just that.

```
std::count_if(persons.cbegin(), persons.cend(),
              older_than(42));
std::count_if(persons.cbegin(), persons.cend(),
              older_than(16));
```

We are here relying on the fact that template functions like `std::count_if` don't require arguments of specific types to be passed to them. They require only that the argument has all the features needed by the template function implementation. In this case, `std::count_if` requires the first two arguments to behave like forward iterators, and the third argument to behave like a function. Some people call this a weakly-typed part of C++, but that is a misnomer — this is still strongly typed, it is just that templates are using duck-typing. There are no runtime checks whether the type is a function object or not, all the needed checks are performed at compile-time.

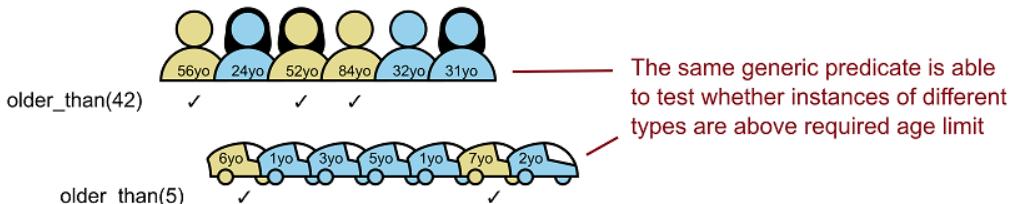
3.1.4 Creating generic function objects

In the previous example, we have created a function object that checks whether a person is older than some predefined age limit. This solved the problem of not needing to create different functions for different age limits, but it is still overly restricting — it accepts only persons as input.

There are quite a few types that could have the age information in them — from

concrete things like cars and pets, to more abstract ones like software projects. If we wanted to count the number of cars that are older than 5 years, we couldn't use the above function object because it is defined only for people.

Figure 3.2. It is useful not only to allow specifying different age limits when creating the predicate, but also to support different types that have the age information



Again, instead of writing a different function object for each type, we want to be able to write the function object once, and to be able to use it for any type that has the age information.

We could solve this in an object-oriented way by creating a super-class with a virtual `.age()` member function, but that would have runtime performance penalties, and it would force all the classes that want to support our `older_than` function object to inherit from that super-class. This ruins encapsulation, so we aren't even going to bother ourselves with this approach.

The first valid approach we could take is turn the `older_than` class into a class template. It would be templated on the type of the object whose age we want to check.

```
template <typename T>
class older_than {
public:
    older_than(int limit)
        : m_limit(limit)
    {}

    bool operator()(const T& object) const
    {
        return object.age() > m_limit;
    }

private:
    int m_limit;
};
```

Now we can use it for any type that has the `.age()` getter:

```
std::count_if(persons.cbegin(), persons.cend(),
    older_than<person_t>(42));
```

```
std::count_if(cars.cbegin(), cars.cend(),
              older_than<car_t>(5));
std::count_if(projects.cbegin(), projects.cend(),
              older_than<project_t>(2));
```

Unfortunately, this approach forces us to specify the type of the object for which we want to check its age. While this is sometimes useful, in most cases it is just tedious. It might also lead to problems if the specified type doesn't exactly match the type that will be passed to the call operator.

Instead of creating the class template, we could have just made the call operator a templated member function. This way, we won't need to specify the type when we instantiate the `older_than` function object, and the compiler will automatically deduce the type of the argument when invoking the call operator.

Listing 3.8. Creating a function object with generic call operator (example:older-than-generic/main.cpp)

```
class older_than {
public:
    older_than(int limit)
        : m_limit(limit)
    {}

    template <typename T>
    bool operator()(T&& object) const
    {
        return std::forward<T>(object).age() > m_limit; ①
    }

private:
    int m_limit;
};
```

- ① We are forwarding the `object` because it can have separate overloads on the `age` member function for lvalue and rvalue instances. This way, the correct overload will be called.

We can now use the `older_than` function object without explicitly stating the type of the object we will be calling it on. We can even use the same object instance for different types (if we want to check all objects against the same age limit).

Listing 3.9. Using a function object with generic call operator

```
older_than predicate(5);

std::count_if(persons.cbegin(), persons.cend(),
              predicate);
std::count_if(cars.cbegin(), cars.cend(),
```

```
predicate);
std::count_if(projects.cbegin(), projects.cend(),
    predicate);
```

We have a single instance of the `older_than` type that checks whether the age of the object passed to it is greater than five. The same instance implements this check for any type that has the `.age()` member function which returns an integer, or something comparable to an integer. We have created a proper generic function object.

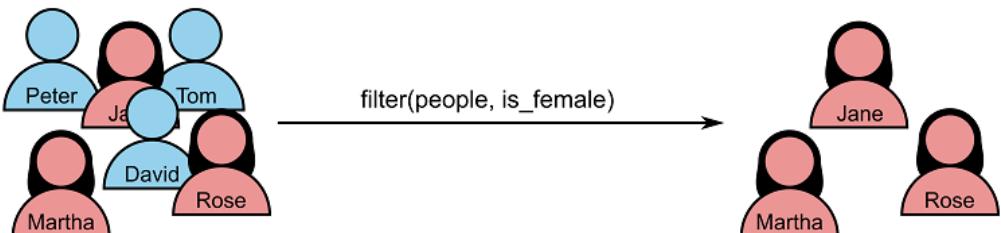
So far, we have seen how to write different types of function objects in C++. Function objects like these are perfect for passing them to the algorithms from the standard library, or to our own higher-order functions. The only problem is that their syntax is a bit verbose and that we need to define them outside of the scope we are using them in.

3.2 Lambdas and closures

In all previous examples, we relied on the fact that the function we passed to the algorithm already exists outside of the function we are using the algorithm in. It is sometimes tedious having to write a proper function, or even a whole class in order just to be able to call an algorithm from the standard library, or some other higher-order function. It can also be seen as bad software design, since we are forced to create and name a function that might not be useful to anyone else, and is used only in one place in the program — as an argument to some algorithm call.

Fortunately, C++ has lambdas, which are a syntactic sugar for creating anonymous (unnamed) function objects. Lambdas allow us to create function objects inline — at the place where we want to use them instead of having to do it outside of the function we are currently writing. Let's see this on an example from earlier — we have a group of people, and we want to collect only females from that group.

Figure 3.3. Getting all females from a group of people. We have been using a predicate called `is_female` which was a free-standing function for this. Now, we are going to do the same with a lambda.



We have seen how we can use the `std::copy_if` to filter a collection of people based on their gender in the previous chapter (Listing 2.8). We relied on the fact

that the function `is_female` exists as a free function that accepts a `person_t` and returns true if that person was female. Since the `person_t` type has a member function that returns its gender, it is a bit of an overkill to have to define free-standing predicate functions for all of the possible genders, just to be able to use those predicates when calling an STL algorithm.

Instead, we can just use a lambda, and achieve the same effect while keeping the code localized and not polluting the program name space.

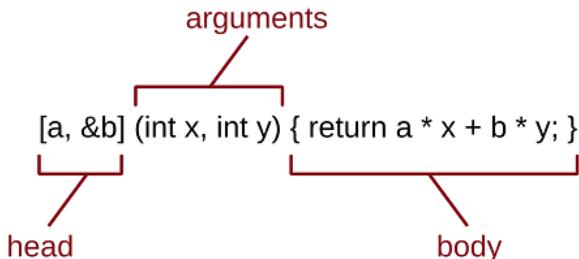
```
std::copy_if(people.cbegin(), people.cend(),
            std::back_inserter(females),
            [](&const person_t& person) {
                return person.gender() == person_t::female;
            });

```

We called `std::copy_if` just like in the original example, but instead of passing an already existing predicate function, we told the compiler to create an anonymous function object that is used only here, a function that takes a constant reference to a person, and returns whether that person is a female or not, and to pass that function object to the `std::copy_if` algorithm.

3.2.1 Lambda syntax

Syntactically, lambda expressions in C++ have three main parts — a head, an argument list, and the body:



The head of the lambda specifies which variables from the surrounding scope will be visible inside of the lambda body. The variables can be captured as values (the variable `a` in the above snippet), or by references (specified by prefixing the variable name with an ampersand - variable `b` in the above snippet). If they are passed in as values, their copies will be stored inside of the lambda object itself, while if they are passed in as references, only a reference to the original variable will be stored.

It is also allowed not to specify all the variables that need to be captured, but to tell the compiler to capture all variables from the outer scope that are used in the body. If you want to capture all variables by-value, you can define the lambda head to

be [=], and if you want to catch them by-reference, you could write [&].

Let's see a few examples of different lambda heads:

- [a, &b] — lambda head from the previous example, a is captured by-value, b is by-reference;
- [] — a lambda that doesn't use any variable from the surrounding scope. These lambdas don't have any internal state and can be implicitly cast to ordinary function pointers;
- [&] — captures all variables that are used in the lambda body by-reference;
- [=] — captures all variables that are used in the lambda body by-value;
- [this] — captures this pointer by-value;
- [&, a] — captures all variables by-reference, except a which is captured by-value;
- [=, &b] — captures all variables by-value, except b which is captured by-reference.

Explicitly enlisting all the variables that need to be used in the lambda body and not using the *wildcards* like [&] and [=] is maybe tedious, but is the preferred approach when creating lambdas because it stops us from accidentally using a variable that we didn't intend to use (the usual problem is the this pointer since we might use it implicitly by accessing a class member or by calling a class member function).

3.2.2 Under the hood of lambdas

While lambdas do provide a nice inline syntax to create function objects, they aren't magic — they can't do anything we weren't able to do without them. They are just that — a nicer syntax to create and instantiate a new function object.

Let's see this on an example. We are still going to work with a list of people, but this time they will be employees in a company. The company is separated into different teams where each team has a name. We are going to represent a company using a simple `company_t` class. This class will have a member function to get the name of the team each employee belongs to. Our task is to implement a member function that will accept the team name, and return how many employees it contains. Here is the setup:

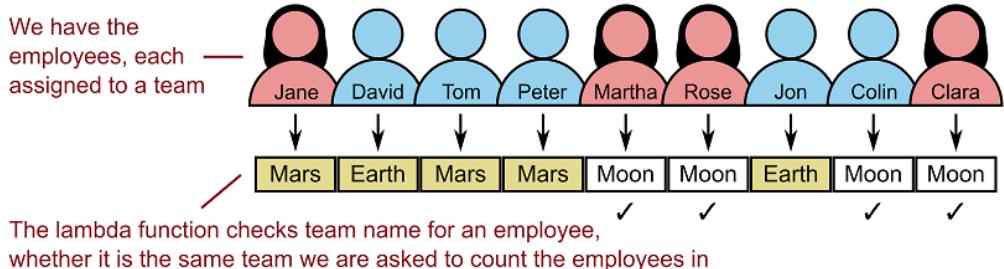
```
class company_t {
public:
    std::string team_name_for(const person_t& employee) const;

    int count_team_members(const std::string& team_name) const;

private:
    std::vector<person_t> m_employees;
    // ...
};
```

We need to implement the `count_team_members` member function. We can do it by checking which team each employee belongs to, and counting only those whose team matches the function arguments.

Figure 3.4. We are passing a lambda as a predicate to the `std::count_if` algorithm. The lambda will receive a single employee, and it will check whether it person belongs to the team we are checking the count for. The `std::count_if` algorithm will count all the "yeses" and return us the answer.



Like before, we will use `std::count_if`, but we will pass a bit more complicated lambda as the predicate function:

Listing 3.10. Counting the number of employees in a given team (example:counting-team-members/main.cpp)

```
int company_t::count_team_members(
    const std::string& team_name) const
{
    return std::count_if(
        m_employees.cbegin(), m_employees.cend(),
        [this, &team_name]
            (const person_t& employee)
        {
            return team_name_for(employee) ==
                team_name;
        }
    );
}
```

- ➊ We need to capture `this` because we are implicitly using it when calling the `team_name_for` member function (it would be more obvious if we have written `this->team_name_for`), and we have captured `team_name` because we need to use it for comparison
- ➋ As before, this function object has only one argument — the employee for which we need to return whether he or she belongs to the specified team

So, what actually happens in C++ when we write this? It will create a new class that has two members — a pointer to the `company_t` object, and a reference to a `std::string` — one for each captured variable. And it will implement the call

operator on that class with the same arguments and the same body of the lambda. It will create a class equivalent to the following

Listing 3.11. Lambda converted to a class

```
class lambda_implementation {
public:
    lambda_implementation(
        const company_t* _this,
        const std::string& team_name)
        : m_this(_this)
        , m_team_name(team_name)
    {
    }

    bool operator()(const person_t& employee) const
    {
        return m_this->team_name_for(employee) == m_team_name;
    }

private:
    const company_t* m_this;
    const std::string& m_team_name;
};
```

Apart from creating a class, evaluating the lambda expression will also create an instance of that class—a closure—the object containing some state or environment along with code that should be executed on that state.

One thing worth noting is that the call operator of lambdas is constant by default (contrary to the other parts of the language where you need to explicitly specify that something is `const`). If you want to change the value of the captured variables, when they are captured by-value and not by-reference, you will need to declare the lambda as `mutable`. In the following example, we are going to use the `std::for_each` algorithm to write all words beginning with an uppercase letter, and use the `count` variable to count how many words we wrote. This is sometimes useful for debugging, but mutable lambdas should be avoided.

Obviously, there are better ways to do this, but the point here is to demonstrate how mutable lambdas work.

Listing 3.12. Creating a mutable lambda

```
int count = 0;
std::vector<std::string> words{"An", "ancient", "pond"};

std::for_each(words.cbegin(), words.cend(),
    [count]      ①
    (const std::string& word)
    mutable     ②
```

```

    {
        if (isupper(word[0])) {
            std::cout << word
                << " " << count << std::endl;
            count++;
        }
    };
);

```

- ➊ we are capturing count by-value, all changes to it are localized and visible only from the lambda
- ➋ mutable comes after the argument list, and tells the compiler that the call operator on this lambda shouldn't be const

Lambdas don't allow us to do anything that we weren't able to do before, but they do free us from writing too much boiler-plate code and allow us to keep the logic of our code localised instead of forcing us to create functions or function objects outside of the function in which they are used. In these examples, the boiler-plate code would be larger than the code that actually does something useful.

The type of the lambda

When a lambda is created, the compiler will create a class with a call operator and will give that class an internal name. That name isn't accessible to the programmer, and different compilers will give it different names. The only way to save a lambda (without excess baggage of `std::function` that we will see in a moment) is to declare the variable using `auto` like we did in the previous example.

3.2.3 Creating arbitrary member variables in lambdas

So far, we have created lambdas that can access the variables from the surrounding scope by either storing the references to the used variables, or storing the copies of them inside of the closure object.

On the other hand, by writing our own classes with the call operator, we were able to create as many member variables as we wanted, without needing to tie them to a variable in the surrounding scope — we could initialize them to some fixed value, to initialize them to a result of some function and similar.

While this might sound like it isn't a big deal (one could always declare a local variable with the desired value before creating the lambda, and then capture it), in some cases it is essential.

If we only had the ability to capture objects by value or by reference, we wouldn't be able to store objects that are moved-only inside of a lambda (instances of classes that define the move-constructor, but that don't have the copy-constructor).

The most obvious example of this issue is when you want to give the ownership of a `std::unique_ptr` to a lambda.

Say we want to create a network request, and we have the session data stored in a unique pointer. We are creating a network request, and scheduling a lambda to be executed when that request is completed. We still want to be able to access the session data in the completion handler, so we need to capture it in the lambda.

Listing 3.13. Error when trying to capture a move-only type

```
std::unique_ptr<session_t> session = create_session();

auto request = server.request("GET /", session->id());

request.on_completed(
    [session]      ①
    (response_t response)
{
    std::cout << "Got response: " << response
    << " for session: " << session;
}
);
```

① error: There is no copy-constructor for std::unique_ptr<session_t>

In cases like these, we can use the extended syntax (generalized lambda captures). Instead of just specifying which variables we want to capture, we can define arbitrary member variables simply by specifying the variable name and its initial value separately. The type of the variable will be automatically deduced from the specified value.

Listing 3.14. Generalized lambda captures

```
request.on_completed(
    [ session = std::move(session),   ①
      time = current_time()        ②
    ]
    (response_t response)
{
    std::cout
        << "Got response: " << response
        << " for session: " << session
        << " the request took: "
        << (current_time() - time)
        << "milliseconds";
}
);
```

① Moving the ownership of session into the lambda

② Creating a time lambda member and setting its value to the current time

This allows us to move objects from the surrounding scope into the lambda. By

using `std::move` in the above example, we are calling the move-constructor for the `session` member variable of the lambda, and it will take the ownership over the session object.

We have also created a new member variable called `time` which doesn't capture anything from the outer scope — it is a completely new variable whose initial value is the result of the `current_time` function which is evaluated when lambda is constructed.

3.2.4 Generic lambdas

So far, we have seen that with lambdas we can do most of the things we were able to do with normal function objects.

Earlier in this chapter, we implemented a generic function object which allowed us to count how many items in a given collection were older than some predefined age limit, without caring about the type of those items by making the call operator be a template function (Listing 3.9).

Lambdas also allow us to create generic function objects simply by specifying the argument types to be `auto`. We can easily create a generic lambda which can accept any object that has a `.age()` member function that checks whether the age of a given object is above some predefined limit like this:

**Listing 3.15. Creating a generic lambda that can accept any object with
a `.age()` member function**

```
auto predicate = [limit = 42](auto&& object) { ❶
    return object.age() > limit;
}; ❷

std::count_if(persons.cbegin(), persons.cend(),
             predicate);
std::count_if(cars.cbegin(), cars.cend(),
             predicate);
std::count_if(projects.cbegin(), projects.cend(),
             predicate);
```

- ❶ We don't have a name for the type of the `object` argument, so we can't use it for perfect forwarding.
We would need to write `std::forward<decltype(object)>(object)` instead.

It is important to note that a generic lambda is a class on which the call operator is templated, and not a template class that has a call operator. This means that the lambda will deduce the type for each of its arguments which were declared as `auto` when it is called, and not when it is constructed, and that the same lambda can be used on completely different types.

More generic lambdas in C++20

If we create a lambda with multiple arguments declared as `auto`, the types of all those arguments will be deduced separately when the lambda is called.

If we wanted to create a generic lambda that has all the arguments of the same type, but for that type to be deduced automatically, we need to resort to some trickery and use `decltype` to define the types of lambda arguments.

For example, we might want to create a generic lambda that gets two values of the same type, and checks whether they are equal. We could specify the type of the first argument to be `auto`, and use `decltype(first)` to specify the type of the second argument like so:

```
[] (auto first, decltype(first) second) { ... }
```

When the lambda is called, the type for the first argument will be deduced, and the second argument will have the same type.

In C++20, the lambda syntax will be extended to allow specifying the template parameters explicitly. Instead of relying on the `decltype` trick, we will be able to write this:

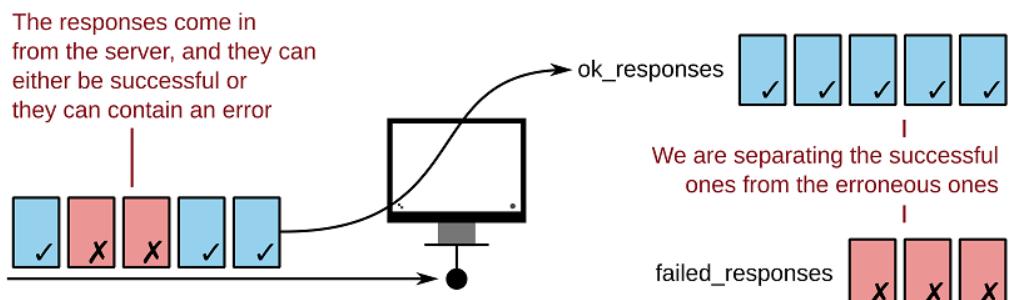
```
[] <typename T> (T first, T second) { ... }
```

While this new syntax is planned for inclusion in C++20, it is already supported by GCC, and other compilers will likely be quick to support it as well.

3.3 Writing function objects that are even terser than lambdas

As we saw, lambdas are nice and remove the need to write large chunks the boiler-plate code when creating function objects. But, on the other hand, they introduce new boiler-plate code (albeit much smaller amount) to the call site.

Figure 3.5. We are filtering the responses based on whether they contain an error or not in order to process them separately.



Consider the following situation — we are writing a web client. We sent a few

requests to the server, and got a collection of responses (with a type named `response_t`) in return. Since every request can fail, the `response_t` provides the `.error()` member function which will provide us with some information in the case of failure. If the request has failed, this function will return `true` (or some object that when cast to a Boolean returns `true`). It will be `false` otherwise.

And we want to filter the given collection depending whether the responses were erroneous or not. We want to create one collection that will contain the valid responses, and one containing the invalid ones. We can easily pass a lambda to the previously defined `filter` function:

```
ok_responses = filter(responses,
    [](const response_t& response) {
        return !response.error();
    });
failed_responses = filter(responses,
    [](const response_t& response) {
        return response.error();
});
```

If we had to do this often, and for various other types that have a `.error()` member function that returns a Boolean value (or some other type that is convertible to a Boolean), the amount of boiler-plate code will significantly surpass the boiler-plate that we would have if we just wrote the function object by hand.

It would be ideal if we had a terse syntax for creating lambdas that would allow us to write something like this:

```
ok_responses      = filter(responses, _.error() == false);
failed_responses = filter(responses, _.error());
```

Unfortunately, this isn't valid C++, but we now know enough to be able to create something similar.

What we want to create is a function object that will be able to work on any class that provides a `.error()` member function, and that will provide us with a nice and terse syntax. While we can't achieve the same syntax we had in the previous code snippet, we can do something even nicer — to provide the user with a few different ways of writing the predicate (different people prefer different notations when testing bools) which will look like regular C++ code without any boiler-plate at all:

```
ok_responses      = filter(responses, not_error);
// or              filter(responses, !error);
// or              filter(responses, error == false);

failed_responses = filter(responses, error);
// or              filter(responses, error == true);
// or even         filter(responses, not_error == false);
```

In order to achieve this, we only need to implement a simple class with an overloaded call operator. It will need to store a single Boolean value that will tell us whether we are looking for correct or erroneous responses (or other objects).

Listing 3.16. Basic implementation of a predicate to test for errors

```
class error_test_t {
public:
    error_test_t(bool error = true)
        : m_error(error)
    {}

    template <typename T>
    bool operator()(T&& value) const
    {
        return m_error ==
            (bool)std::forward<T>(value).error(); ① ②
    }

private:
    bool m_error;
};

error_test_t error(true);
error_test_t not_error(false);
```

- ① The usage of `std::forward` here might seem strange, but it is nothing more than perfectly forwarding of the argument passed to the call operator as the forwarding reference. We are doing it because the `error()` member function could be implemented differently for lvalue and rvalue references.

This will allow us to use `error` and `not_error` as predicates. In order to support the alternative syntax options we saw earlier, we just need to provide the `operator==` and `operator!=`.

Listing 3.17. Defining convenient operators for the predicate function object

```
class error_test_t {
public:
    // ...

    error_test_t operator==(bool test) const
    {
        return error_test_t(
            test ? m_error : !m_error ① ② ③
        );
    }

    error_test_t operator!() const
    {
```

```

        return error_test_t(!m_error);      ②
    }

    // ...
};

```

- ① If the test is true, we are returning the same predicate we currently have, while if it is false we are returning its negation
- ② We are returning the negation of the current predicate

While this seems like an awful amount of code to write just to make creating a predicate function at the call site simpler than writing a lambda, it really pays off in the cases where the same predicate is used often. And it isn't a difficult thing to imagine that a predicate that tests for errors might be used often.

My advice is to never refrain from simplifying the important parts of code. By creating the `error_test_t` predicate function object, we are writing a large chunk of code in some separate header file that (if tested properly) nobody will ever need to open while debugging the main parts of the program. And it will make said main parts of the program shorter and easier to understand, thus also making them easier to debug.

3.3.1 Operator function objects in STL

As mentioned in the previous chapter, many algorithms from the standard library allow customizing their behaviour. For example, `std::accumulate` allows replacing addition with another operation, `std::sort` allows changing the comparison function used to order the elements.

We can write a function or a lambda, and pass it to the algorithm, but it can be an overkill in some cases. For this reason, like in the case where we implemented the `error_test_t` function object, the standard library provides wrappers over all common operators. For example, if we wanted to multiply a collection of integers, we could write our own function that multiplies two numbers and returns a result, but we could also rely on the operator wrapper from the STL called `std::multiplies`:

Listing 3.18. Calculating the product of a vector of integers

```

std::vector<int> numbers{1, 2, 3, 4};

product = std::accumulate(numbers.cbegin(), numbers.cend(), 1,
                         std::multiplies<int>());

// product is 24

```

In the same manner, if we wanted to use `std::sort` to order elements in a descending order, we might want to use `std::greater` as the comparison function:

Listing 3.19. Sorting number in a descending order

```
std::vector<int> numbers{5, 21, 13, 42};

std::sort(numbers.begin(), numbers.end(), std::greater<int>());
// numbers now contain {42, 21, 13, 5}
```

Table 3.1. Operator wrappers available in the standard library:

Group	Wrapper name	Operation
Arithmetic operators	std::plus	arg_1 + arg_2
	std::minus	arg_1 - arg_2
	std::multiplies	arg_1 * arg_2
	std::divides	arg_1 / arg_2
	std::modulus	arg_1 % arg_2
	std::negates	- arg_1 (a unary function)
Comparison operators	std::equal_to	arg_1 == arg_2
	std::not_equal_to	arg_1 != arg_2
	std::greater	arg_1 > arg_2
	std::less	arg_1 < arg_2
	std::greater_equal	arg_1 >= arg_2
	std::less_equal	arg_1 <= arg_2
Logical operators	std::logical_and	arg_1 && arg_2
	std::logical_or	arg_1 arg_2
	std::logical_not	!arg_1 (a unary function)
Bitwise operators	std::bit_and	arg_1 & arg_2
	std::bit_or	arg_1 arg_2
	std::bit_xor	arg_1 ^ arg_2

The diamond alternative

Since C++14, it is possible to omit the type when using the operator wrappers from the standard library. Instead of writing `std::greater<int>()`, you can write just `std::greater<>()`.

For example, when calling `std::sort`, you can do it like this:

```
std::sort(numbers.begin(), numbers.end(),
          std::greater<>());
```

If you have a C++14-compliant compiler and the standard library, this version is the preferred one unless you want to force the conversion to a specific type before the comparison takes place.

3.3.2 Operator function objects in other libraries

While the operator wrappers from the STL can cover the most rudimentary cases, they are a bit awkward to write and they aren't easily composed.

Because of this, a few libraries have been created as a part of the `boost` project to remedy this problem (and a few outside of `boost`).

Boost libraries

Initially, `boost` libraries were created as a testbed for features planned for inclusion in future versions of the C++ STL.

Afterwards, the collection of libraries grew and they now cover many both common and niche programming domains. Many C++ developers consider them to be an integral part of the C++ ecosystem, and a go-to place when they need something that is missing in the standard library.

We are going to show a few examples of creating function objects using the `Boost.Phoenix` library⁴. We aren't going to cover this library in detail, but the provided examples should be enough to serve as teasers for the reader to investigate it further, or to implement own version of the same concepts.

Let's start with a small motivator example. We want to use `std::partition` algorithm on a collection of numbers to separate those that are less than or equal to 42, from those that aren't.

We saw that the STL provides `std::less_equal` function object. But we can't use it with `std::partition`. The partitioning algorithm expects a unary function that returns a Boolean result, whereas `std::less_equal` requires two arguments. While the standard library does provide a way to bind one of those arguments to a fixed value (we will see how to do it in the next chapter) it isn't really pretty, and has significant downsides.

Libraries like `Boost.Phoenix` provide an alternative way of defining function objects that heavily relies on operator overloading. It defines *magic* argument placeholders, and operators which allow to compose them. Solving our problem becomes rather trivial:

```
using namespace boost::phoenix::arg_names;

std::vector<int> numbers{21, 5, 62, 42, 53};

std::partition(numbers.begin(), numbers.end(),
    arg1 <= 42);
```

⁴ Boost.Phoenix library documentation is available at www.boost.org/doc/libs/release/libs/phoenix/doc/html/index.html

```
// numbers now contain {21, 5, 42,    62, 53}
//                      <= 42          > 42
```

The `arg1` is a placeholder defined in the Boost.Phoenix library that binds itself to the first argument passed to the function object. When we call the `operator<=` on the placeholder, it won't actually compare the `arg1` object to the value `42`, but will return us a unary function object. When that function object is called, it will return whether the passed argument is less than or equal to `42`. This behaviour is quite similar to the `operator==(bool)` we created for our `error_test_t` class.

We could create much more intricate function objects in this manner. We could, for example, easily calculate the sum of half-squares of all numbers in a collection (if we would ever need to do something this strange):

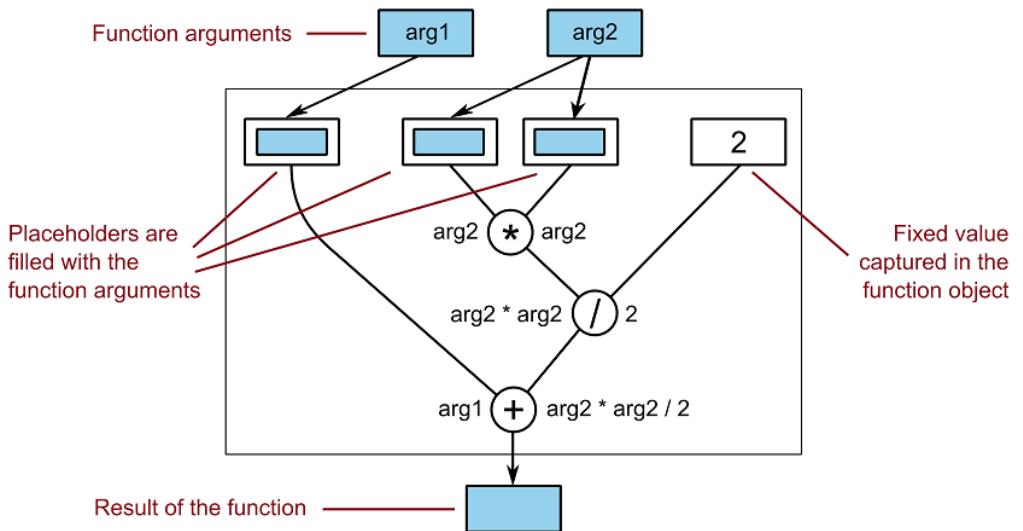
Listing 3.20. Using Boost.Phoenix instead of STL operator wrappers

```
std::accumulate(numbers.cbegin(), numbers.cend(), 0,
               arg1 + arg2 * arg2 / 2); ①
```

- ① Just a reminder, when folding, the first argument is the accumulated value, and the second argument is the value we are currently processing

The expression `arg1 + arg2 * arg2 / 2` produces a function object that takes two arguments, square the second one, divides it by two, and adds it to the first argument.

Figure 3.6. Phoenix expression is composed of captured values and placeholders that represent the function arguments. When the arguments are provided by calling the function object, the expression is evaluated and the calculated value returned as the result of the function



Or we can even replace the operator wrappers provided by the standard library that we have used in the previous example.

Listing 3.21. Using Boost.Phoenix instead of STL operator wrappers

```
product = std::accumulate(numbers.cbegin(), numbers.cend(), 1,
                         arg1 * arg2);

std::sort(numbers.begin(), numbers.end(), arg1 > arg2);
```

While these allow creating quite complex function objects, they are most useful for writing simple ones. If you need to create something complex, you should always turn to lambdas — if a function is big enough, the boiler-plate that lambdas introduce becomes insignificant, and lambdas make it easier for the compiler to optimize the code.

The main downside of libraries like Boost.Phoenix is that they slow down the compilation time significantly. If that proves to be a problem for your project, you should revert back to using lambdas, or create your own simpler functional objects like the `error` object we implemented in this chapter.

3.4 Wrapping function objects with `std::function`

So far, we have relied on automatic type deduction when we wanted to either

accept a function object as an argument (by making the function object type parametrized with a template) or when we wanted to create a variable to store the function object like lambda (by using `auto` instead of specifying the type explicitly).

While this is the preferred and optimal way, it is sometimes not possible. In the cases where we need to save a function object as a member in a class that can't be templated on the type of that function object (thus having to explicitly specify its type), or when we want to use a function between separate compilation units, we need to be able to provide a concrete type.

Since there is no super-type for all different kinds of function objects that we could use in these cases, the standard library provides a template class `std::function` that can wrap any type of function object.

Listing 3.22. Storing different function objects inside `std::function`

```
std::function<float(float, float)> test_function; ①

// ordinary function
test_function = std::fmaxf;

// class with a call operator
test_function = std::multiplies<float>();

// class with a generic call operator
test_function = std::multiplies<>();

// lambda
test_function = [x](float a, float b) { return a * x + b; };

// generic lambda
test_function = [x](auto a, auto b) { return a * x + b; };

// boost.phoenix expression
test_function = (arg1 + arg2) / 2;

// lambda with a wrong signature
test_function = [](std::string s) { return s.empty(); } // ERROR!
```

① The result of the function is written first, followed by the list of arguments inside the parenthesis

The `std::function` isn't templated on the contained type, but on the signature of the function object. The template argument specifies the return type of the function and its arguments. This means that we are able to use the same type to store ordinary functions, function pointers, lambdas, and other callable objects (as seen in the previous example) — anything that has the signature specified in the `std::function` template parameter.

We can even go a bit further — we can even store some things that don't provide the usual call syntax like class member variables and class member functions. For example, the C++ core language stops us from calling the `.empty()` member function of `std::string` like it was a free-standing function (i.e. `std::string::empty(str)`), and because of that, we don't consider data members and member functions to be function objects, but if we store them in a `std::function`, we can call it with the normal call syntax⁵

Listing 3.23. Storing callables that aren't function objects

```
std::string str{"A small pond"};
std::function<bool(std::string)> f;

f = &std::string::empty;

std::cout << f(str);
```

We will refer to function objects (as previously defined) together with pointers to member variables and functions, as *callables*. We will see how we can implement functions that can call any callable object, and not only function objects in the chapter 11, where we will explain the usage of the `std::invoke` function.

While all the above makes the `std::function` really useful, it shouldn't be overused because it introduces significant performance penalties. In order to be able to hide the contained type, and provide a common interface over all different callable types, it uses the technique known as type-erasure. We aren't going to dive deeper into this now (we will see some applications of type erasure in chapter 11), but it is enough to be aware that type-erasure is usually based on virtual member function calls. Since virtual calls are resolved at runtime, the compiler can't inline the call and has limited optimization opportunities.

An additional issue of `std::function` is that although its call operator is marked as `const`, it can invoke a non-constant callable. This can lead to all kinds of problems in multi-threaded code.

Small function object optimization

When the wrapped callable is a function pointer or a `std::reference_wrapper` (as produced by the `std::ref` function), the small object optimization is guaranteed to be performed. This means that these callables will be stored directly inside the `std::functionobject`, without the need for any dynamic memory allocation.

⁵ Mind that if you are using clang, you might get a linking error due to the libc++ bug that it doesn't export the `std::string::empty` symbol.

Larger objects may be constructed in dynamically allocated memory and accessed by the `std::function` object through a pointer. This has a performance impact on the construction and destruction of the `std::function` object, and when its call operator is invoked.

The maximum size of the callable object for which the small function object optimization will be performed varies between different compilers and standard library implementations.

3.5 Summary

- It is possible to use objects that are castable to function pointers as they were ordinary functions, but the call operator is the preferred way of creating function objects.
- For functions whose return type isn't important enough to be explicitly stated, we can leave it up to the compiler to deduce it from the value we are returning using the `auto` keyword.
- Using the automatic return type deduction avoids any type of conversion or narrowing that might arise when we specify the type explicitly (for example, returning a `double` value in a function that should return an `int`).
- If we want to create a function object that can work on a multitude of different types, we will want to make its call operator a template function.
- Lambdas are useful syntactic sugar for creating function objects. It is usually better to use them than to whole classes with call operators by hand.
- Lambdas in C++14 have become a true replacement for most function objects that we could write by-hand, they add new possibilities when capturing variables, and they support creating generic function objects.
- While the lambdas provide terser syntax for creating function objects, they are still overly verbose for some cases. In those situations, we can use the libraries like Boost.Phoenix, or roll out our own function objects.
- The `std::function` is a useful, but it comes with a performance penalty equivalent to a virtual function call.

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch03

Creating new functions from the old ones



This chapter covers:

- What is partial function application?
- Fixing arguments of functions to specific values with `std::bind`
- Using lambdas for partial function application
- Are all functions in the world actually unary?
- Easily creating functions that operate on collections of items

Most programming paradigms tend to provide a way to increase code reusability. In the object-oriented world, we create classes that we can later use in various situations. We can use them directly, or we can combine them in order to implement more complex ones. The possibility to break complex systems into smaller components that can be used and tested separately is a very powerful one.

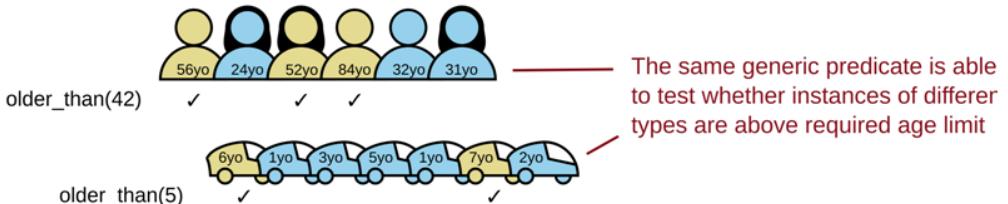
In the same way that OOP gives us the tools to combine and modify types, the functional programming paradigm gives us ways to easily create new functions by combining the functions we already have written, by *upgrading* specialized functions to cover more general use-cases, or the other way round, by taking more general functions and simplifying them to perfectly fit a specific use-case.

4.1 Partial function application

In the previous chapter, we had an example where we counted the number of objects in a collection which were older than some predefined age limit. If we think about that example a bit, we conceptually have a function that takes two arguments

— an object (like a person or a car) and a value which will be compared against the age of said object. This function would return true if the age of the object is greater than that value.

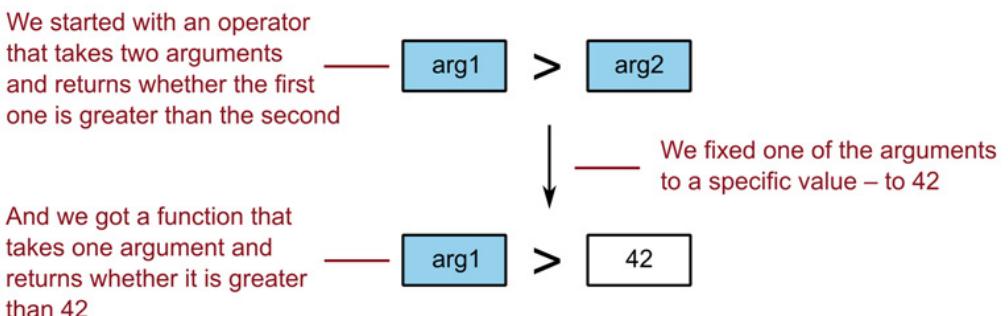
Figure 4.1. On a higher level, we have a function of two arguments - one object, and an integer value representing the age limit. It returns whether the object is older than said age limit. When we fix the age limit to a specific value, we get a unary function that compares the object's age against that predefined value.



Now, since the `std::count_if` algorithm expects us to give it a unary predicate, we had to create a function object that stores the age limit inside itself, and use that value when it gets the actual object whose age needs to be checked. The notion that we don't need to pass in all the arguments that a function requires at once is something that we will explore a bit more in this section.

Let's look at a simpler version of the example from the previous chapter. Instead of checking whether a person's age is greater than some specified value, we are taking the general *greater than* operator (a function that takes two arguments) and we will bind its second argument to a fixed value, thus creating a unary function (Figure 4.2).

Figure 4.2. Fixing the second argument of the greater-than comparison operator to 42 turns it into a unary predicate that tests whether a value is greater than 42



We are creating an ordinary function object that gets a single integer value on its construction, stores it inside of the function object, and then uses it to compare the values passed to its call operator against it.

Instances of this class can be used with any higher-order function like `std::find_id`, `std::partition` or `std::remove_if` that takes a predicate, given that it will call that predicate on integer values (or values that can be implicitly converted to an integer).

Listing 4.1. Creating a function object that compares its argument to a predefined value

```
class greater_than {
public:
    greater_than(int value)
        : m_value
    {
    }

    bool operator()(int arg) const
    {
        return arg > m_value;
    }

private:
    int m_value;
};

...

greater_than greater_than_42(42); ①
greater_than_42(1); // false ①
greater_than_42(50); // true ①

std::partition(xs.begin(), xs.end(), greater_than(6)); ②
```

- ① We can create an instance of this object and use it multiple times to check whether other specific values are greater than 42
- ② And we can create an instance directly when calling a specific algorithm like `std::partition` to separate items greater than 6 from those that are less

We haven't done anything overly complicated here. We just took a binary function — the `operator>` : `(int, int) → bool` — and created a new unary function object from it that does the same thing as the greater-than operator but with the second argument fixed to a specific value (Figure 4.2).

This concept of creating a new function object from an existing one by fixing one or more of its arguments to a specific value is called *partial function application*. The word *partial* in this case means that we have provided only some, but not all arguments needed to calculate the result of the function.

4.1.1 A generic way of converting binary functions into unary ones

Let's try to make the previous implementation more generic. We want to create a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/functional-programming-in-cplusplus>

Licensed to Alexey Fadeev <alexey.s.fadeev@gmail.com>

function object that can wrap any binary function that the user passes to it and bind one of its arguments. For consistency sake, we are going to bind the second argument, like the `greater_than` class did.

Note

The standard library used to contain a function called `std::bind2nd` which implemented the same concept as the one we are presenting in this section.

While this function is removed from the standard in favor of lambdas and `std::bind` which will be covered in the next section, it is a nice example of how partial application can be implemented in C++.

The function object needs to be able to store a binary function, and one of its arguments. Since we don't know in advance the types of the function and the second argument, we will need to create a class templated on these two types. The constructor will only need to initialize the members and nothing more. Note that we are going to capture both the function and the second argument that will later be passed to it by-value for simplicity.

Listing 4.2. Basic structure of the templated class

```
template <typename Function, typename SecondArgType>
class partial_application_on_2nd_impl {
public:
    partial_application_bind2nd_impl(Function function,
                                     SecondArgType second_arg)
        : m_function(function),
          , m_value(second_arg)
    {
    }

    ...

private:
    Function m_function;
    SecondArgType m_second_arg;
};
```

Now, we don't know the type of the first argument in advance, so we need to make the call operator to be a template as well. The implementation is quite straightforward, we just need to call the function that we have stored in the `m_function` member, and forward it the argument passed to the call operator as the first argument, along with the value stored in the `m_second_arg` member as the second argument.

Listing 4.3. Implementation of the call operator for partial function application of a binary function

```
template <typename Function, typename SecondArgType>
```

```

class partial_application_bind2nd_impl {
public:
    ...

    template <typename FirstArgType>
    auto operator()(FirstArgType&& first_arg) const
        -> decltype(m_function(
            std::forward<FirstArgType>(first_arg),           ③
            m_second_arg));                                ③
    {
        return m_function(
            std::forward<FirstArgType>(first_arg),           ①
            m_second_arg);                                ②
    }

    ...
};


```

- ① The argument of the call operator is passed to the function as the first argument
- ② The saved value is passed as the second argument to the function
- ③ If we don't have a compiler that supports automatic return type deduction, we need to use decltype to achieve the same effect. Otherwise, we could have written: decltype(auto)

Beyond function objects

Just a quick reminder, since we are using the regular function call syntax, this class will only work with function objects, and not other callables such as class member pointers. If we wanted to support them as well, we will need to use `std::invoke` which will be explained in chapter 11.

Now that we have the complete class defined, we just need one more thing to make it useful. Namely, if we want to use it directly from our code, we would need to specify the template argument types explicitly when creating an instance of this class. This would be quite ugly, and in some cases even impossible (for example, we don't know a type of the lambda).

Template argument deduction for classes in C++17

The requirement of class template instantiation for the template arguments to be explicitly defined has been removed in C++17, but since this standard hasn't yet been widely-accepted, we won't rely on this feature in these examples.

For that, we need to create a template function whose job is only to make an instance of this class. Since the template argument deduction⁶ works when calling

⁶ You can check out en.cppreference.com/w/cpp/language/template_argument_deduction for more information regarding the template argument deduction

functions, we won't need to specify the types when calling it. The function just calls the constructor for the class we defined previously, and forwards its arguments to it. It is mostly boiler-plate that the language forces us to write in order to have a way to instantiate a template class without having to explicitly specify the template arguments.

Listing 4.4. A wrapper function that allows easy creation of the previous function object

```
template <typename Function, typename SecondArgType>
partial_application_bind2nd<Function, SecondArgType>
bind2nd(Function&& function, SecondArgType&& second_arg)
{
    return partial_application_bind2nd<Function, SecondArgType>(
        std::forward<Function>(function),
        std::forward<SecondArgType>(second_arg));
}
```

Now, let's use our newly created function to replace the usage of `greater_than` in our starting example.

Listing 4.5. Using bind2nd to create a function object that compares its argument to a predefined value

```
auto greater_than_42 = bind2nd(std::greater<int>(), 42);

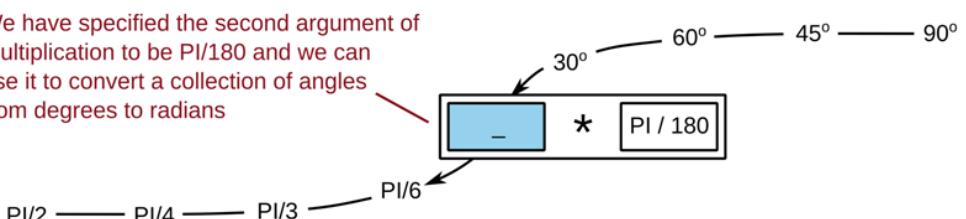
greater_than_42(1); // false
greater_than_42(50); // true

std::partition(xs.begin(), xs.end(), bind2nd(std::greater<int>(), 6));
```

We have created a more general function that can be used in many more places than `greater_than`, and trivially replaced all usages of `greater_than` with it. In order to see that it is truly more general, we will cover another short example where we could use the newly created `bind2nd` function.

Figure 4.3. Binding one argument of multiplication to PI/180 gives us a function that converts degrees to radians

We have specified the second argument of multiplication to be $\text{PI}/180$ and we can use it to convert a collection of angles from degrees to radians



Imagine you have a collection of angle sizes given in degrees. And the problem is

that the rest of your code relies on angles being given in radians. This problem appears every so often in many graphics programs — the graphics libraries usually define angles for rotation in degrees, while most maths libraries that are used alongside these graphics libraries tend to require radians. Converting one to another is simple — the value in degrees just need to be multiplied by PI / 180.

Listing 4.6. Using bind2nd to convert a collection of angles from degrees to radians

```
std::vector<double> degrees = {0, 30, 45, 60};
std::vector<double> radians(degrees.size());

std::transform(degrees.cbegin(), degrees.cend(),    1
              radians.begin(),           2
              bind2nd(std::multiplies<double>(),
                      PI / 180));      3
```

- ➊ We are iterating through all elements in the degrees vector
- ➋ And saving the converted results to the vector radians
- ➌ We are passing multiplication with the second argument bound to PI / 180 as the transformation function

This example has shown that we aren't limited to creating only predicate functions (functions that return `bool` values), but that we can take any binary function and turn it into a unary one by binding its second argument to a specific value. This allows us to use said binary function in contexts which require unary functions like in the above example.

4.1.2 Using std::bind to bind values to specific function arguments

Pre-C++11, the standard library provided two functions similar to the function we created in the previous section. These functions were `std::bind1st` and `std::bind2nd`, and they provided a way to turn a binary function into a unary one by binding its first or second argument to a specific value. Much like our `bind2nd` function did.

In C++11, these two functions were deprecated (and later removed in C++17) in favor of a much more general one called `std::bind`. The `std::bind` isn't limited only to binary functions, and can work with functions that have an arbitrary number of arguments. It also doesn't limit the user which arguments can be bound — the user can bind any number of arguments, in any order, while leaving the rest unbound.

Let's start with a most basic example of using `std::bind` — binding all arguments of a function to specific values, but without actually invoking it. The first argument of `std::bind` is the function whose arguments we want to bind to specific values, and other arguments are said values that the function arguments will be bound to. We will bind the arguments of the `std::greater` comparator function to fixed

values — 6 and 42.

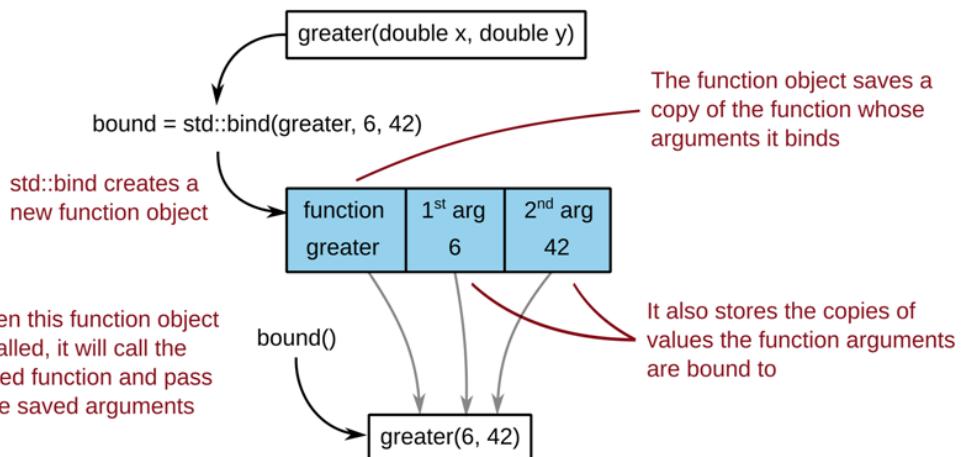
Listing 4.7. Binding all arguments of a function with std::bind

```
auto bound =
    std::bind(std::greater<double>(), 6, 42); ①
bool is_6_greater_than_42 = bound();           ②
```

- ① The std::greater isn't yet invoked, we have just created a function object that will call it with the specified values 6 and 42
- ② When calling the bound function object, only then the elements 6 and 42 are actually compared

By providing values for all needed arguments when binding a function, we are creating a new function object that just saves the function whose arguments we are binding (the std::greater comparator in our case) and all the passed values (Figure 4.4). Binding defines the values for the arguments, but it doesn't invoke the function. The function will be invoked only when someone calls the function object returned by std::bind. In our example, the std::greater comparator is called only when calling bound().

Figure 4.4. We have a two-argument function that tests whether its first argument is greater than its second. Binding both arguments to specific values like 6 and 42 creates a nullary function object that, when called, returns the result of greater(6, 42).



Technically, this isn't partial function application since we are binding all the arguments of a function, but it is a good start to introduce the syntax of std::bind.

Now, let's see how we can leave just one of the arguments unbound, while binding the other one. We can't just define one value, and skip the other one because std::bind wouldn't know which argument we wanted to bind — the first,

or the second one. Because of that, `std::bind` introduces a concept of placeholders. If we want to bind an argument, we will pass a value for it to the `std::bind` like we did in the example above. But if we want to say that an argument should remain unbound, we will have to pass a placeholder instead.

The placeholders look and behave in a similar way to what we have seen in the previous chapter when we created function objects using the Boost.Phoenix library. This time, they have a bit different names — `_1` instead of `arg1`, `_2` instead of `arg2` and so on.

Note

Placeholders

The placeholders are defined in the `<functional>` header in the `std::placeholders` namespace. This is one of the rare cases where we will omit explicitly specifying the namespace since it would significantly hinder the readability of the code.

For all the `std::bind` examples, we will consider that using namespace `std::placeholders` is used.

Let's take our previous example, and modify it to bind only one of the arguments. We want to create one predicate that tests whether a number is greater than 42, and one predicate that tests whether a number is less than 42. Both by using only `std::bind` and the `std::greater` comparator. In the first case, we will bind the second argument to the specified value, while passing a placeholder as the first; and vice-versa for the second case.

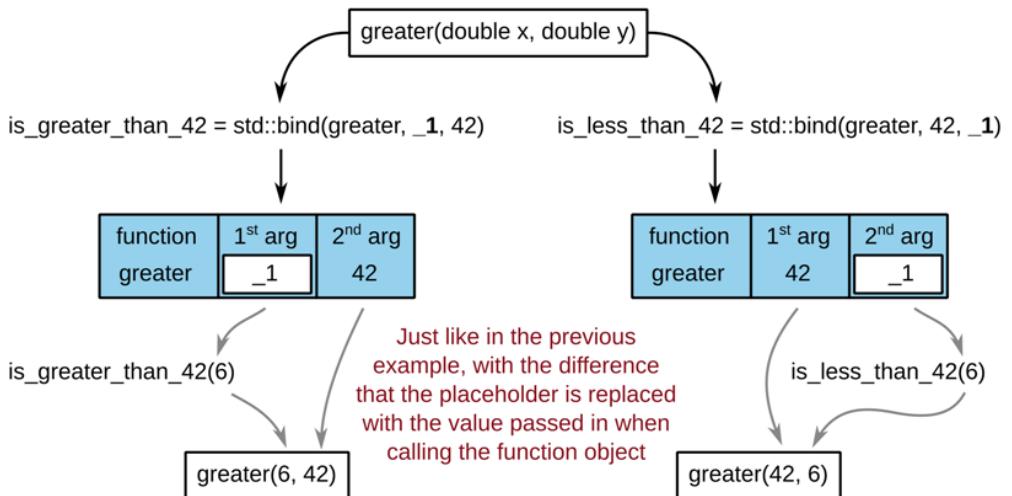
Listing 4.8. Binding all arguments of a function with `std::bind`

```
auto is_greater_than_42 =
    std::bind(std::greater<double>(), _1, 42);
auto is_less_than_or_equal_to_42 =
    std::bind(std::greater<double>(), 42, _1);

is_less_than_or_equal_to_42(6); // returns true
is_greater_than_42(6);        // returns false
```

What is going on here? We are taking a two-argument function — the `std::greater` comparator — and we are binding one of its arguments to a value, and one to a placeholder. Binding an argument to a placeholder effectively states that we don't have a value for that argument at the moment, and that we are leaving a hole that we are going to fill later.

Figure 4.5. We have a two-argument function that tests whether its first argument is greater than its second. Binding one of the arguments to a value and the other to a placeholder, creates a unary function object that, when called with a single argument, uses that argument to fill the hole defined by the placeholder.



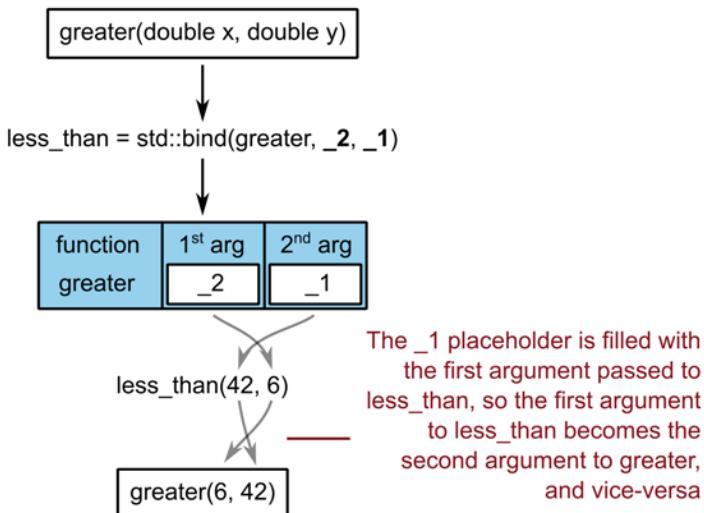
When we call the function object returned by `std::bind` with a specific value, that value will be used to fill that hole defined by the `_1` placeholder (Figure 4.5). If we bind several arguments to the same placeholder, all these arguments will be filled with the same value.

4.1.3 Reversing the arguments of a binary function

We have seen that we can bind all arguments to specific values, or leave one of the arguments unbound by using the `_1` placeholder. Can we use multiple placeholders at the same time?

Say we have a vector of doubles that we want to sort in ascending order. Usually, we would do this with `std::less` (which is the default behaviour of `std::sort`) but this time, just for demonstration purposes, we want to do it with `std::greater`. In order to do so, we can create a function object using `std::bind` that just reverses the arguments before they are passed to `std::greater`.

Figure 4.6. We are specifying the first placeholder to be the second argument passed to greater, and vice-versa. This results in a new function that is the same as greater, but with arguments switched.



What happens here is that we are creating a two-argument function object (since the highest numbered placeholder is `_2`) which, when called, will invoke `std::greater` with the same arguments it received, but swapped. We can use it to implement sorting of our scores in ascending order.

Listing 4.9. Sorting film scores in ascending order using `std::greater`

```
std::sort(scores.begin(), scores.end(),
          std::bind(std::greater<double>(), _2, _1)); ①
```

- ① Note that we have reversed the arguments by passing the `_2` placeholder first, and the `_1` placeholder second

Now that we have a basic grasp of `std::bind` and what it can do, let's move on to more complex examples.

4.1.4 Using `std::bind` on functions with more arguments

We will use our collection of people yet again, but this time, it is our desire to write all the people to the standard output, or some other output stream. We will start by defining a free-standing function that will write the person information in one of the predefined formats. The function will have three arguments — a person, a reference to the output stream, and the desired output format.

```
void print_person(const person_t& person,
                  std::ostream& out,
```

```

        person_t::output_format_t format)
{
    if (format == person_t::name_only) {
        out << person.name() << '\n';

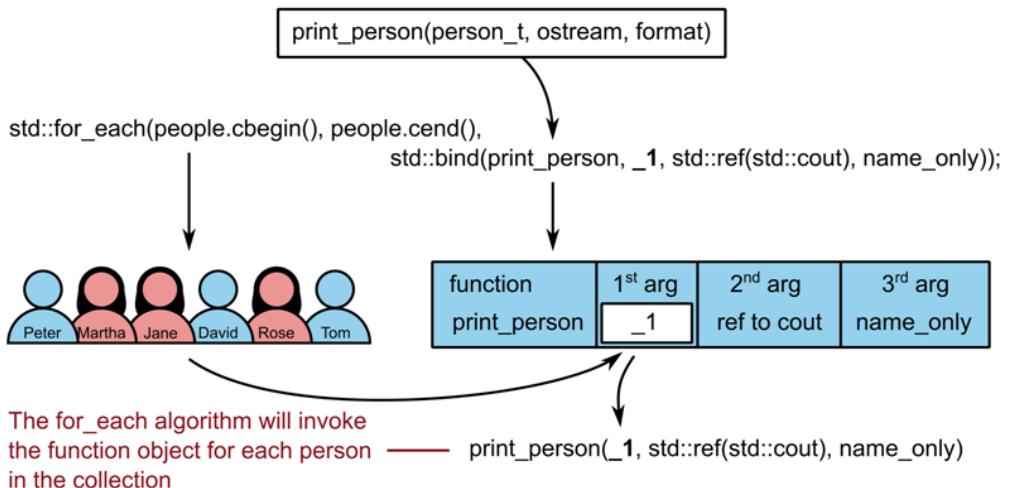
    } else if (format == person_t::full_name) {
        out << person.name() << ' '
            << person.surname() << '\n';

    }
}

```

Now, if we wanted to write out all people in our collection, we could do it simply by passing the `print_person` to the `std::for_each` algorithm, with the `out` and `format` arguments bound.

Figure 4.7. We want to create a function that prints names from people to the standard output. We have a function that is able to write a person's information in multiple different formats and to different output streams. We are specifying the output stream and the format, while leaving the person empty, to be defined by the `std::for_each` algorithm



By default, `std::bind` stores the copies of the bound values in the function object it returns. Since copying is disabled on `std::cout`, we need to bind `out` argument to a reference to `std::cout` and not its copy. For this, we are using the `std::ref` helper function.

Listing 4.10. Binding the arguments of `print_person` function to create a unary function acceptable by `std::for_each` (example:`printing-people/main.cpp`)

```

std::for_each(people.cbegin(), people.cend(),
    std::bind(print_person,

```

```

        _1,
        std::ref(std::cout),
        person_t::name_only   1
    ));

    std::for_each(people.cbegin(), people.cend(),
        std::bind(print_person,
            _1,
            std::ref(file),
            person_t::full_name 2
        ));

```

- ➊ We are creating a unary function that will print the name of a person to the standard output
- ➋ Here, the function print the name and the surname of a person to the specified file

We started with a function that needs three arguments — the person, the output stream and the output format, and used it to create two new ones that both take a person as their argument, but that behave differently. One writes just the first name of a person to the standard output, and another that writes person's full name to the specified file. And we did it without actually writing those new functions by-hand — we just used the existing function, bound a couple of its arguments, and got a unary function that we can use in the `std::for_each` algorithm as the result.

So far, we have preferred free-standing functions or static member functions to proper class member functions simply because class member functions aren't considered to be function objects since they don't support the function call syntax.

This limitation is artificial. Member functions are essentially the same as the ordinary ones, with a difference that they have an implicit first argument `this` that points to the instance of the class that the member function was called on.

We have a `print_person` function that takes three arguments — a `person_t`, an output stream and a format. We could replace this function with a member function inside the `person_t` type like this:

```

class person_t {
    ...
    void print(std::ostream& out, output_format_t format) const
    {
        ...
    }
    ...
};

```

In essence, there is no difference between `print_person` and `person_t::print` functions apart from the fact that C++ doesn't allow us to call the later using the usual call syntax. The `print` member function also has three arguments — an implicit argument `this` that refers to a `person`, and two explicit

arguments `out` and `format`; and it does the same thing as `print_person` does.

Fortunately, `std::bind` can bind arguments of any callable and it treats both `print_person` and `person_t::print` the same. If we wanted to convert the previous example to use this member function, we just need to replace `print_person` with a pointer to `person_t::print` member function.

Listing 4.11. Binding the arguments of print member function to create a unary function acceptable by std::for_each

```
std::for_each(people.cbegin(), people.cend(),
    std::bind(&person_t::print,           ①
              _1,
              std::ref(std::cout),
              person_t::name_only
            ));
```

① We are creating a unary function object from a member function pointer

Note

Exercise

It would be a nice exercise to check which free-standing functions in our previous examples could be easily replaced by member functions with the help of `std::bind`.

We have seen that `std::bind` allows us to perform partial function application by binding some arguments, but also to reorder the function arguments. It has slightly unorthodox syntax for the object-oriented world, but it is terse enough and easy to understand. It supports any callable object making it easy to use both normal function objects and member variable and function pointers with standard algorithms and other higher-order functions.

4.1.5 Using lambdas as an alternative for std::bind

While `std::bind` provides a nice and terse syntax for creating function objects that bind or reorder arguments of already existing functions, it does come with a cost. Namely, it makes the job of the compiler much more difficult, and it is harder to optimize. It is implemented on the library level, and it uses many complex template meta-programming techniques to achieve its goal.

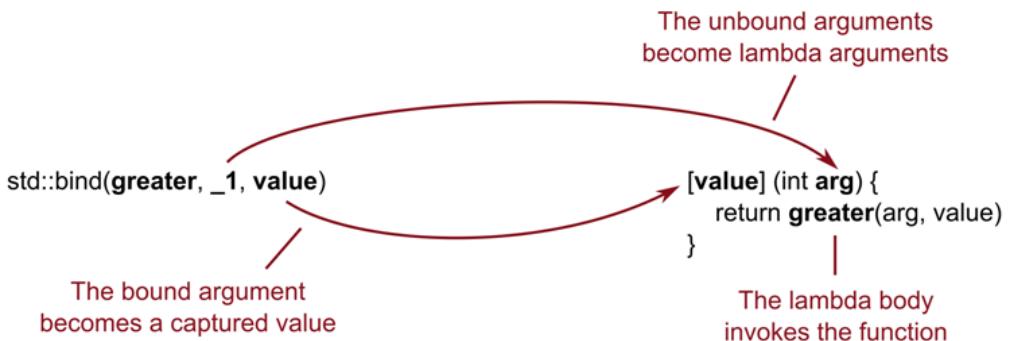
An alternative to using `std::bind` is to use lambdas for partial function application. Lambdas are a core-language feature, which means that the compiler will optimize them more easily. The syntax will be a bit more verbose, but it will allow the compiler more freedom to optimize the code.

Turning all `std::bind` calls to lambda is rather simple:

- turn any argument bound to a variable or a reference to a variable into a captured variable

- turn all placeholders into lambda arguments
- and specify all the arguments bound to a specific value directly in the lambda body

Figure 4.8. If we want to use lambdas instead of std::bind, we need to convert bound arguments to captured values, and the unbound ones will become lambda arguments



Let's see what our examples from the previous section would look like. First, we had a two-argument function and we bounded both its arguments to a specific value. There are no arguments bound to variables, so we don't need to capture anything. There are no placeholders, so our lambda won't have any arguments. And we will pass the bound values directly when calling the `std::greatercomparator`.

```
auto bound = [] {
    return std::greater<double>()(6, 42);
};
```

Now, this can become even shorter in this case—if we replaced `std::greater` call with the operator`<`, but we will leave it as is since not all functions can be replaced with infix operators, and we are demonstrating the general approach of replacing `std::bind` with lambdas.

Our next example was to bind one of the arguments to a specific value, while binding the other one to a placeholder. This time, since we have a single placeholder, the lambda will have a single argument. We still don't need to capture any variable.

```
auto is_greater_than_42 =
    [](double value) {
        return std::greater<double>()(value, 42);
};
auto is_less_than_42 =
    [](double value) {
        return std::greater<double>()(42, value);
};
```

```
is_less_than_42(6);    // returns true
is_greater_than_42(6); // returns false
```

As before, we passed the bound values directly in the lambda body when calling the `std::greater` comparator.

Now, going further, we want to reimplement the example where we sorted a vector of scores in ascending order using `std::greater` for which we just swapped the arguments when calling it. We now have two placeholders, so our lambda will have two arguments.

```
std::sort(scores.begin(), scores.end(),
          [](double value1, double value2) {
              return std::greater<double>()(value2, value1);
          });
```

Like in the example with `std::bind`, we are calling `std::sort` with a function object that will, when called with two arguments, pass those arguments to `std::greater` in reversed order.

And we are left with our last example in which we used `std::for_each` to print out the names of people contained in a collection. In this example we have a few different entities we need to consider:

- we have a single `_1` placeholder, so we will create a unary lambda
- we are binding the values `person_t::name_only` and `person_t::full_name` which we can pass in directly when calling the `print_person` function
- we are using a reference to `std::cout` which we don't need to capture since it will anyhow be visible in the lambda
- and we have a reference to the output stream called `file` which we need to capture by-reference in the lambda

```
std::for_each(people.cbegin(), people.cend(),
             [](const person_t& person) {
                 print_person(person,
                             std::cout,
                             person_t::name_only);
             });

std::for_each(people.cbegin(), people.cend(),
             [&file](const person_t& person) {
                 print_person(person,
                             file,
                             person_t::full_name);
             });
```

All these examples have been generally identical to those that used `std::bind`, just with a different, slightly more verbose syntax. But, there are some subtle differences. The `std::bind` stores the copies of all values, references and functions in the function object it creates. It even needs to store the information about which

placeholders are used. Lambdas, on the other hand, store only what we want them to. This can make `std::bind` slower than an ordinary lambda if the compiler doesn't manage to optimize it properly.

We have seen a few different approaches to do partial function application in C++. From doing partial application of the operators like multiplication or comparison using the Boost.Phoenix library (as we have seen in the previous chapter) which gave us nice and terse syntax, to partial application of any function we want using the `std::bind` function or lambdas. Lambdas and `std::bind` give us similar level of expressiveness, but with a different syntax. Lambdas tend to be more verbose, but are more likely to lead to more performant code (it is always worth it to benchmark your particular use-cases).

4.2 Currying – a different way to look at functions

We have seen what partial function application is and how to use it in C++. Now, we are moving on to something else, something that has a strange name—currying, that often looks much like partial function application to the untrained eye. In order not to confuse these two, we are going to first define the concept, and then see a few examples.

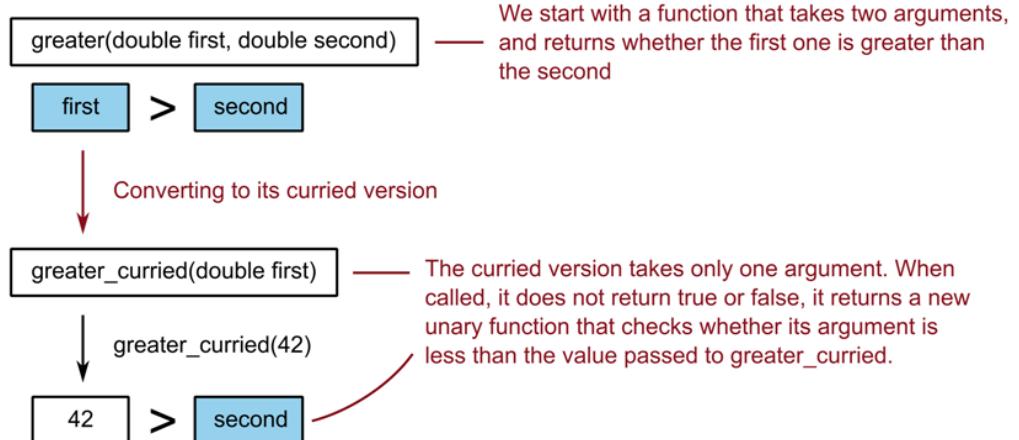
Haskell Curry

Currying is named after Haskell Curry, a mathematician and logician who perfected this concept from the ideas first introduced by Gottlob Frege and Moses Schönfinkel.

Say we are working in a programming language that doesn't allow us to create functions that have more than a single argument. While this seems limiting at first, we will see that it allows us all the expressiveness we have with proper n-ary functions with a simple, yet very clever trick.

Instead of creating a function that takes two arguments, and returns a single value, we can create a unary function that returns a second unary function. When this second function is called, this means that we have received both needed arguments and that we can return the resulting value. If we had a function that takes three arguments, we could convert it into a unary function that returns the previously defined curried version of the two-argument function. And so on for as many arguments as we would want.

Figure 4.9. We have a function that takes two arguments, and we are converting it to a unary function that doesn't return a value, but a new unary function. This allows us to pass in the arguments one at a time, and not all at once



Let's see this idea on a simple example. We have a function called `greater` that takes two values, and checks whether the first one is greater than the second. On the other hand, we have its curried version which can't return a `bool` value because it only knows the value of the first argument. It returns a unary lambda that captures the value of that argument. The resulting lambda will compare the captured value with its argument when it gets invoked.

Listing 4.12. The greater function and its curried version

```

// greater : (double, double) → bool
bool greater(double first, double second)
{
    return first > second;
}

// greater_curried : double → (double → bool)
auto greater_curried(double first)
{
    return [first](double second) {
        return first > second;
    };
}

// Invocation
greater(2, 3);      ①
greater_curried(2); ②
greater_curried(2)(3)③
  
```

① returns false

- ② returns a unary function object that checks whether its argument is less than 2
- ③ returns false

If we had a function with more arguments, we could just keep nesting as many lambdas as we need until we gather all the arguments needed for us to return a proper result.

4.2.1 *Creating curried functions the easier way*

If you recall our function `print_person`, it was a function that took three arguments — a person, the output stream and the output format. If we wanted to call that function, we would need to pass all three arguments at once, or to perform partial function application in order to separate the arguments into those that we can define immediately, and those that we will define later.

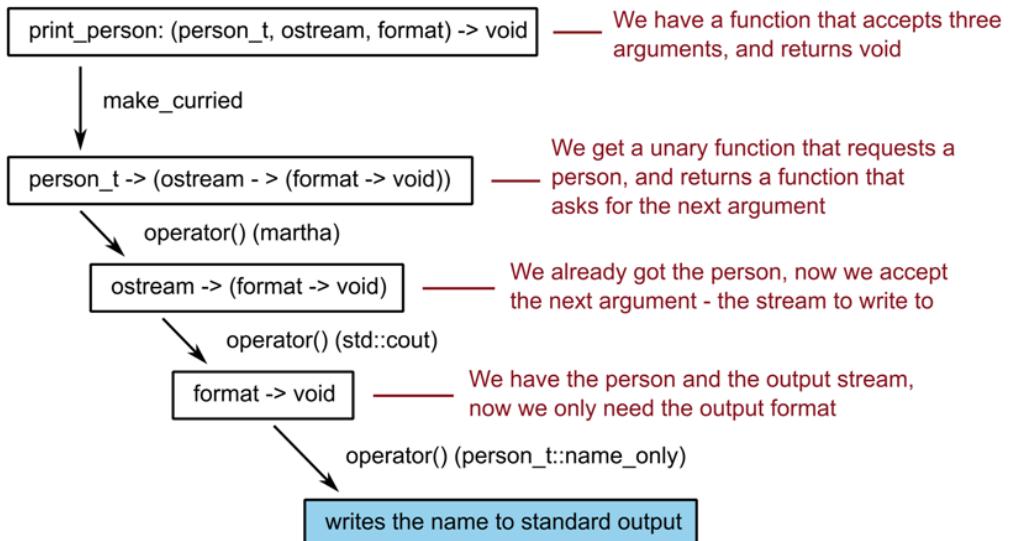
```
void print_person(const person_t& person,
                  std::ostream& out,
                  person_t::output_format_t format);

// We can call it like so:
print_person(person, std::cout, person_t::full_name);
```

We could convert this function to its curried version in the same way we did for `greater` — by nesting enough lambda expressions to capture all the arguments needed to execute `print_person` one by one.

```
auto print_person_cd(const person_t& person)
{
    return [&](std::ostream& out) {
        return [&](person_t::output_format_t format) {
            print_person(person, out, format);
        };
    };
}
```

Figure 4.10. We started with a function of three arguments, and converted it to its curried version. The curried version is a function that accepts a person, and returns a function that takes an output stream. That function, again, returns a new function that takes the output format, and prints the person's info.



Since it is quite tedious to write code like this, we are going to use a helper function called `make_curried` from now on. This function will be able to convert any function we give to it into its curried version. Even more, the resulting curried function will provide us also with syntactic sugar which will allow us to specify more than one argument at a time if we need to. This is just a syntactic sugar, the curried function is still just a unary function like we said before, just more convenient to use.

Note Implementation of `make_curried`

We will see how to implement a function like this in the chapter 11. At this point, the actual implementation isn't important, we just need it to do what we want.

Before we move on to the possible usages of currying in C++, let's first demonstrate what the `make_curried` function will allow us to do, and what the usage syntax will be like.

Listing 4.13. Using the curried version of the `print_person` function

```

using std::cout;

// Using make_curried instead of nested lambdas
auto print_person_cd = make_curried(print_person);

```

```

print_person_cd(martha, cout, person_t::full_name); ①
print_person_cd(martha)(cout, person_t::full_name); ①
print_person_cd(martha, cout)(person_t::full_name); ①
print_person_cd(martha)(cout)(person_t::full_name); ①

auto print_martha = print_person_cd(martha);          ②
print_martha(cout, person_t::name_only);

auto print_martha_to_cout =
    print_person_cd(martha, cout);                   ③
print_martha_to_cout(person_t::name_only);

```

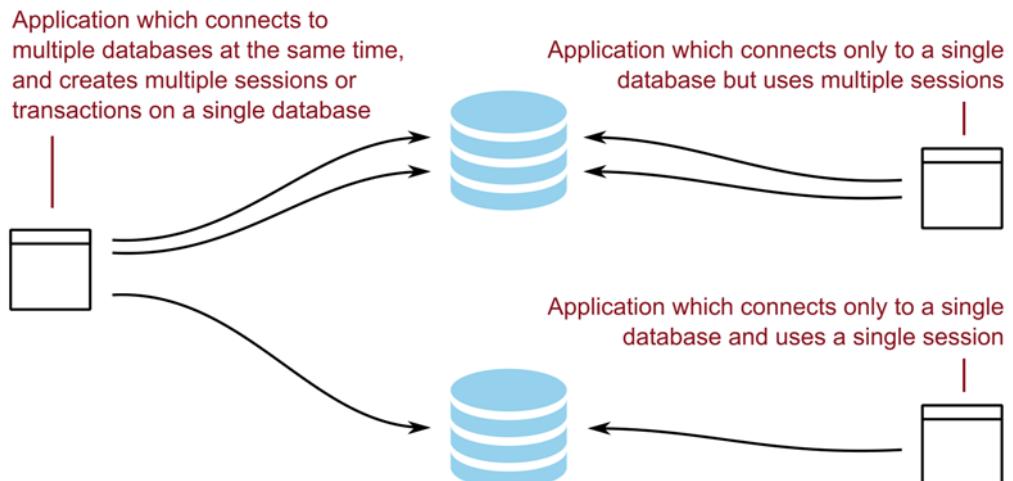
- ① All these write the 'martha's name and surname to the standard output. We can choose how many arguments we want to provide in the single call
- ② Returns a curried function object that will write 'martha's info to an output stream we pass it to, and in the format we specify
- ③ Returns a function object can print 'martha's info to the standard output with the format of our choosing

Now, is this only a cool idea that some mathematician came up with because it was easier for him to think about unary functions instead of having to consider those with more arguments, or is it really useful in our life as programmers?

4.2.2 Using currying with database access

Let's consider a real-world example. We are writing an application that connects to a database and performs some queries on it. Maybe we want to get a list of all people that rated a particular movie.

Figure 4.11. Different applications have different needs – some might want to connect to multiple databases, some might want to have multiple sessions, and some just need a single session while connected to a single database



We are using a library that allows us to create multiple database connections, to initiate connection sub-sessions (for handling database transactions and such) and, obviously, to query the stored data. Let's say that the main query function looks like this:

```
result_t query(connection_t& connection,
               session_t& session,
               const std::string& table_name,
               const std::string& filter);
```

It queries all the rows in a given table that match the given filter. All the queries need to have connection and session specified, so that the library is able to tell which database to query, and what to include in a single database transaction.

Now, many applications don't have the need to create multiple connections, and just need a single one for all queries. One approach that library developers use in this case is to make our query function to be a member function in the `connection_t` type. Another viable alternative would be to create an overloaded query function that wouldn't have the connection as one of the arguments, and it would use the default system connection instead.

This now becomes more entangled when we consider the possibility that some users also don't need multiple sessions. If they access a read-only database, there isn't much point in defining transactions. Now, the library author can also add a query function to a session object which the user would use throughout the application, or we could get another overload of our original query function, this time without both session and connection arguments.

After this, it would also be easy to imagine that parts of user's application might only need access to a single table (in our case, we just need a table that contains the user scores). The library might then provide a `table_t` class that has its own query member function and so on.

It isn't easy to predict all the different use-cases that a user might have. Ant those that we have predicted already made the library more complex than it needed to be. Let's see whether having just a single query function, like the one above, but curried can be sufficient to cover all these cases without making the library developer go nuts over all the overloads and classes we forced him to create for our use-cases.

If the user needs to have multiple database connections, he can either use the query function directly, by always specifying the connection and session, or create a separate function for each connection that will just apply the first argument of the curried function. The later would also be useful if there is only one database connection needed for the whole application.

If only a single database is needed, and a single session, it is easy to create a function that covers that case exactly —we just need to call the

curried query function and pass it the connection and session, and it will return us a function that we can use from there on. And if we just need to query a single table multiple times, we can also bind the table argument as well.

Listing 4.14. Solving the API boom with curried functions

```
auto table  = "Movies";
auto filter = "Name = \"Sintel\"";
```

```
results = query(local_connection, session,
    table, filter);          1
                           1
```

```
auto local_query = query(local_connection);      2
auto remote_query = query(remote_connection);    2
                           2
```

```
results = local_query(session, table, filter);    2
```

```
auto main_query = query(local_connection,
    main_session);           3
                           3
                           3
```

```
results = main_query(table, filter);              3
```

```
auto movies_query = main_query(table);           4
                           4
```

```
results = movies_query(filter);                  4
```

- ① Using the query function as a normal function, by passing all required arguments to it, and getting a list of query results directly
- ② Creating separate functions that are tied to their respective database connections. This is useful in the cases where we often use the same connection over and over again.
- ③ For the cases where we just need a single connection and a single session, we can create a function that binds these two values, so that we can omit them in the rest of the program.
- ④ If we often perform queries over the same table, we can even create a function that always queries that table.

By providing a curried version of the query function, we allowed its user to create exactly the functions needed for his particular use-case, without complicating our API. This is a big improvement on code reusability since we aren't creating separate classes that might end up with separate bugs that only get discovered when a particular use-case appears, and that specific class is actually used. Here, we have a single implementation that can be used in all the different use-cases.

4.2.3 Currying and partial function application

Currying and partial function application seem quite similar at first. They both seem to allow creating new functions by binding a few arguments to specific values while leaving other arguments unbound. So far, we have only seen that they just have different syntax, and that currying is more limiting in the sense that it must bind the arguments in order — first argument first, and the last one last.

They are similar, and allow us to achieve similar things. We might even implement one using the other. But they have an important distinction that makes them both useful.

The advantage of `std::bind` over currying is obvious, we can take the `query` function and bind any of its arguments, while the curried function first binds the first argument.

Listing 4.15. Binding arguments with curried functions and with `std::bind`

```
auto local_query = query(local_connection);           ①
auto local_query = std::bind(
    query, local_connection, _1, _2, _3);           ②
auto session_query = std::bind(
    query, _1, main_session, _2, _3);               ②
```

- ① We are binding only the connection argument to be the `local_connection`.
- ② We can bind the first argument, but we aren't required to. We can bind just the second one.

Now, we might think that `std::bind` is better, just that it has a bit more complex syntax. But it has one important drawback that should be obvious from the above example. In order to use `std::bind`, we need to know exactly how many arguments the function that we are passing to `std::bind` has. We need to bind each argument to either a value (or a variable, a reference) or to a placeholder. With the curried `query` function, we didn't need to care about that, we just defined the value for the first function argument, and it returned us a function that accepts all other arguments, no matter how many there are.

It might seem like only a syntactic difference — we need to type less, but it isn't only that. Beside the `query` function that we have already had, we would probably want to have an `update` function as well. The function would update the values of rows matched by a filter. It would accept the same arguments as `query` did, but with an additional one — the instruction how each matching result should be updated.

```
result_t update(connection_t& connection,
                session_t& session,
                const std::string& table_name,
                const std::string& filter,
                const std::string& update_rule);
```

Now, it would be expected that we will need create the same functions for `update` like we did for `query`. If we have only one connection, and we have created a specialized `query` function that works with that connection, it is to be expected that we will need to do the same for `update` as well. If we wanted to do it with `std::bind`, we would need to pay attention to the number of arguments, whereas with the curried versions of said functions, we just copy-paste the previous

definition and replace `query` with `update`:

```
auto local_query = query(local_connection);
auto local_update = update(local_connection);
```

Now, you might say it is still only the syntactic difference — we know how many arguments a function has, and we could always define enough placeholders to bind them to. But what happens when we don't know the exact number of arguments. If the above approach of creating many functions for which only the first argument is bound to a specific database connection is a common use-case, we might want to create a generic function that automatically binds the `local_connection` to the first argument of any function we pass to it.

With `std::bind`, we would need to use some clever meta-programming to create different implementations depending on the number of arguments. While with curried functions, it would be quite trivial:

```
template <typename Function>
auto for_local_connection(Function f) {
    return f(local_connection);
}

auto local_query = for_local_connection(query);
auto local_update = for_local_connection(update);
auto local_delete = for_local_connection(delete);
```

As we have seen, while similar, both currying and partial function application have their advantages and disadvantages. And both have their use-cases. The partial function application is useful in the places where we have a specific function whose arguments we want to bind. In this case, we know how many arguments that function has, and we can choose which exactly arguments we want to have bound to a specific value. Currying is particularly useful in generic contexts where we can be given a function with any number of arguments. In this case, `std::bind` isn't useful, because if we don't know how many arguments a function has, we can't know how many arguments we need to bind to a placeholder — we don't even know how many placeholders we will need.

4.3 Function composition

Back in 1986, the famous Donald Knuth was asked to implement a program for the "Programming pearls" column in the Communications of ACM journal⁷. The task was to "read a file of text, determine the n most frequently used words, and print out a sorted list of those words along with their frequencies". Knuth produced a solution in Pascal, that spanned 10 pages in said journal. It was thoroughly

⁷ *Communications of the ACM* is the monthly magazine of the Association for Computing Machinery (ACM)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/functional-programming-in-cplusplus>

commented and contained even a novel data data structure for managing the word count list.

As a response to that, Doug McIlroy wrote a Unix shell script that solved the same problem, but in only six lines.

Listing 4.16. Doug McIlroy's script

```
tr -cs A-Za-z '\n' |
  tr A-Z a-z |
  sort |
  uniq -c |
  sort -rn |
  sed ${1}q
```

While we don't care about shell scripting, this example is a nice example of the functional way of problem solving.

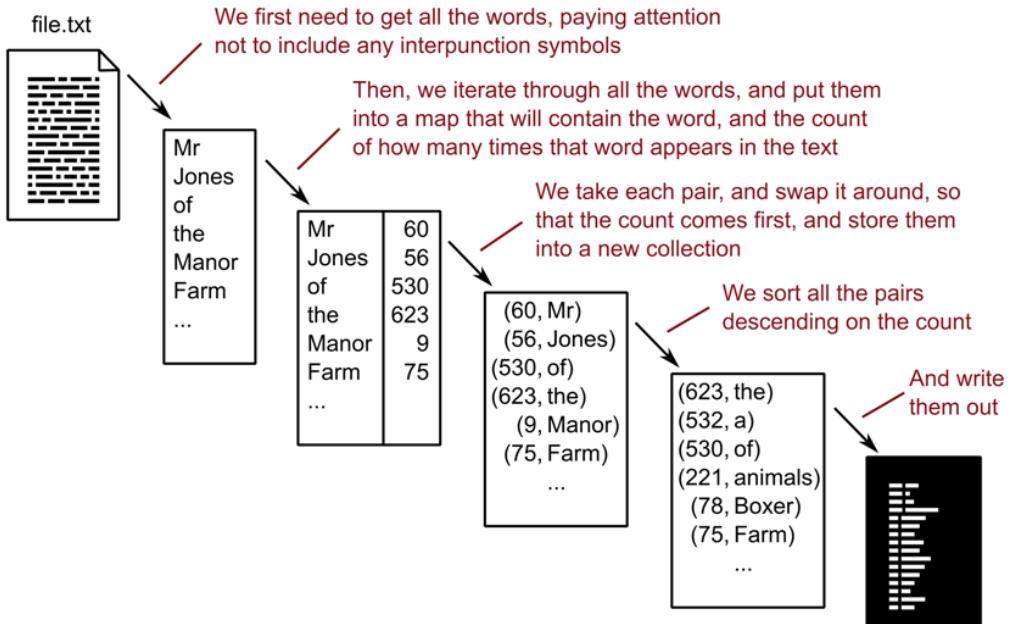
First of all, it is surprisingly short. It doesn't define the algorithm steps, but rather the transformations that need to be done on the input so that we can get the desired output. It also has no state — we don't have any variables at all. This means that it is a pure implementation of the given problem.

We are separating the given problem into a set of simpler but powerful functions that somebody else wrote for us (Unix commands in this case), and we are passing results that one function returns to another function.

This ability to compose functions is exactly what gives us the expressiveness to solve the problem elegantly in just a few lines of code.

Since most of us don't speak "shell" on the same level as Doug does, we are going to analyze this problem from scratch, to see how we could decompose it in C++. We won't deal with the letter capitalization, as it is the least interesting part of this example.

Figure 4.12. We are transforming the original text to a list of words, then assigning count to each word, and sorting the words based on the number of times they appeared in the text



The process (Figure 4.12) consists of the following transformations:

- First, we have a file. We can easily open it, and read the text it contains.
- Instead of a single string, we want to get a list of words that appear in that string.
- Next, we can put all those words into a `std::unordered_map<std::string, unsigned int>` where we will keep the count for each of the words we find (we are using `std::unordered_map` since it is faster than `std::map` and we don't need the items to be sorted at this point).
- Once that is done, we need to iterate through all items in said map (the items will be instances of `std::pair<const std::string, unsigned int>`) and swap each pair around — to have the count first, and then the word.
- Now we have a collection of pairs that we can sort lexicographically (first on the first item of the pair, but descending, then on the second).
- This gives us the most frequent words first, and the least frequent ones last. We just need to print them out.

Looking at this example, we need to create five different functions. All functions should do only one simple thing, and be created in a way that makes them compose easily. This means that the output of one function can be easily passed as the input of another. The original file can be seen as a collection of characters, say a `std::string`, so our first function, let's call it `words`, will need to be able to receive it as its argument. Its return type should be a collection of words, so we can

use `std::vector<std::string>` for that (we could also use something more efficient here that would avoid making unnecessary copies, but that is out of the scope of this chapter — we will return to it in chapter 7).

```
std::vector<std::string> words(const std::string& text);
```

The second function, let's call it `count_occurrences`, gets a list of words, so, again a `std::vector<std::string>` and creates a `std::unordered_map<std::string, unsigned int>` that stores all the words, and the number of times each word appeared in the text. We can even make it a template function so that it could be used for other things, not just strings.

```
template <typename T>
std::unordered_map<T, unsigned int> count_occurrences(
    const std::vector<T>& items);
```

Note

With a bit of light template magic

We could have also made a template for the collection, and not only the contained type.

```
template <typename C,
          typename T = typename C::value_type>
std::unordered_map<T, unsigned int> count_occurrences(
    const C& collection)
```

This function would be able to accept any collection that allows us to deduce the type of the contained items (`C::value_type`). This means that we can invoke it to count occurrences of characters in a string, strings in a vector of strings, integer values in a list of integers and so on.

The third function takes each pair of values from the container, and creates a new pair with elements reversed. It needs to return a collection of newly created pairs. Since we are planning to sort it later, and we want to create easily composable functions, we can return it as a vector.

```
template <typename C,
          typename P1,
          typename P2>
std::vector<std::pair<P2, P1>> reverse_pairs(
    const C& collection);
```

After that, we only need a function that sorts a vector (`sort_by_frequency`), and a function that can write that vector to the standard output (`print_pairs`).

When we compose these functions, we get the final result. Generally, function composition in C++ is done by composing the functions at the point in code where we want to call them on a specific original value. In our case, it would look like this:

```
void print_common_words(const std::string& text)
{
    return print_pairs(
```

```

        sort_by_frequency(
            reverse_pairs(
                count_occurrences(
                    words(text)
                )
            )
        );
}

```

In this example, we have seen how a problem that takes more than a few pages when implemented in the imperative style can be split into a few functions that are fairly small and easy to implement. We have started with a bigger problem and, instead of analyzing what steps we need to perform in order to achieve the result, we started thinking about what transformations we need to perform on the input. We created a short and simple function for each of those transformations. And, finally, we composed them into one bigger function that solves the problem.

4.4 Function lifting revisited

We have touched a bit on the topic of lifting in the chapter 1, and this is the perfect place to expand on it. Broadly speaking, lifting is a programming pattern that gives us a way to transform a given function into a similar function that is applicable in a broader context. For example, if we have a function that operates on a string, lifting will allow us to easily create functions that operate on a vector or a list of strings, on pointers to strings, a map that maps integers to strings, and other different structures that contain strings.

Let's start with a simple function, one that takes a string and converts it to uppercase:

```
void to_upper(std::string& string);
```

If we have a function that operates on a string (the `to_upper` function or any other), what it takes to implement a function that operates on a pointer to a string (it would transform the string that the pointer points to if the pointer isn't null)? What about using that function to transform all people names stored in vector of strings, or in a map that maps movie identifiers to their names? We could easily create all these functions like this:

Listing 4.17. If we know how to process a single string, we can create functions that operate on a collection of strings

```
void pointer_to_upper(std::string* str)
{
    if (str) to_upper(*str);
}
```

1
1
1
1

```

void vector_to_upper(std::vector<std::string>& strs) ②
{
    for (auto& str : strs) {
        to_upper(str);
    }
}

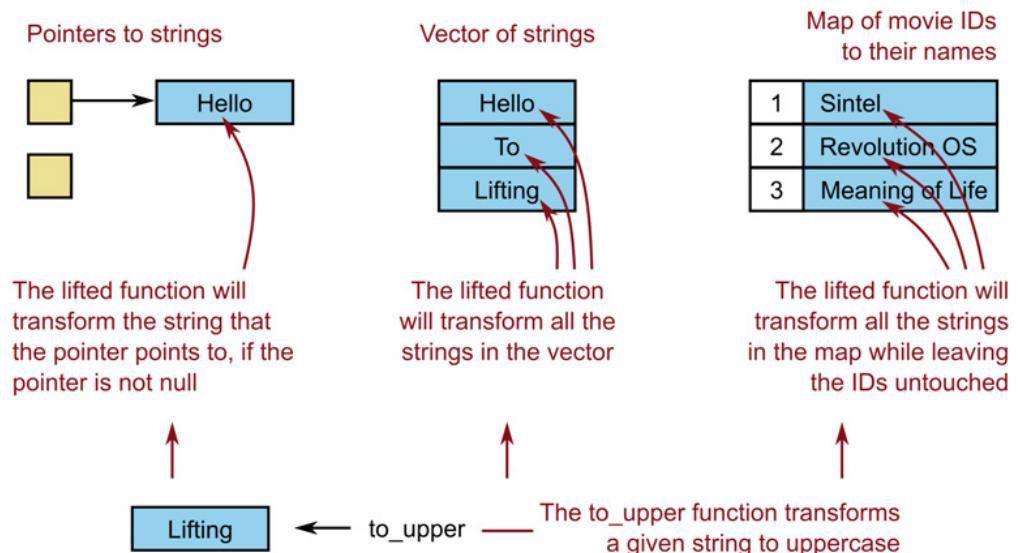
void map_to_upper(std::map<int, std::string>& strs) ③
{
    for (auto& pair : strs) {
        to_upper(pair.second);
    }
}

```

- ① We can see a pointer to a string as a collection that contains either one element, or nothing. If it points to a string, that string will be transformed.
- ② A vector can contain as many strings as we want. This function will convert them all to uppercase.
- ③ The map contains pairs of `<const int, std::string>`. We are going to leave the integer values untouched while converting all strings to uppercase.

It is worth noting that these functions would be implemented in the exact same way if the function `to_upper` was to be replaced with something else. The implementation does, however, depend on the used container type, so we have three different implementations for a pointer to a string, a vector of strings and a map.

Figure 4.13. We are lifting a function that transforms a single string to a few different functions that transform strings contained in various different container-like types



This means that we could have created a higher-order function that takes any function that operates on a single string, and creates a function that operates on a pointer to a string. We would create a separate one that operates on a vector of strings and on a map. These functions are called *lifting functions* because they lift the function that operates on some type to a structure or a collection containing that type.

We are going to implement these using C++14 features for the sake of brevity, but it would be easy to implement them as regular classes with the call operator as we have seen in the previous chapter.

Listing 4.18. Lifting functions that transform an item to create functions that transform collection of items

```
template <typename Function>
auto pointer_lift(Function f)
{
    return [f](auto* item) {
        if (item) {
            f(*item);
        }
    };
}
template <typename Function>
auto collection_lift(Function f)
{
    return [f](auto& items) {
        for (auto& item : items) {
            f(item);
        }
    };
}
```

- ➊ Note that we are using `auto*` as the type specifier. This means that we will be able to use this function not only for pointers to strings, but for pointers to any type we want.
- ➋ The same goes for this function - we can use it not only for vectors of strings, but any iterable collection holding any type.

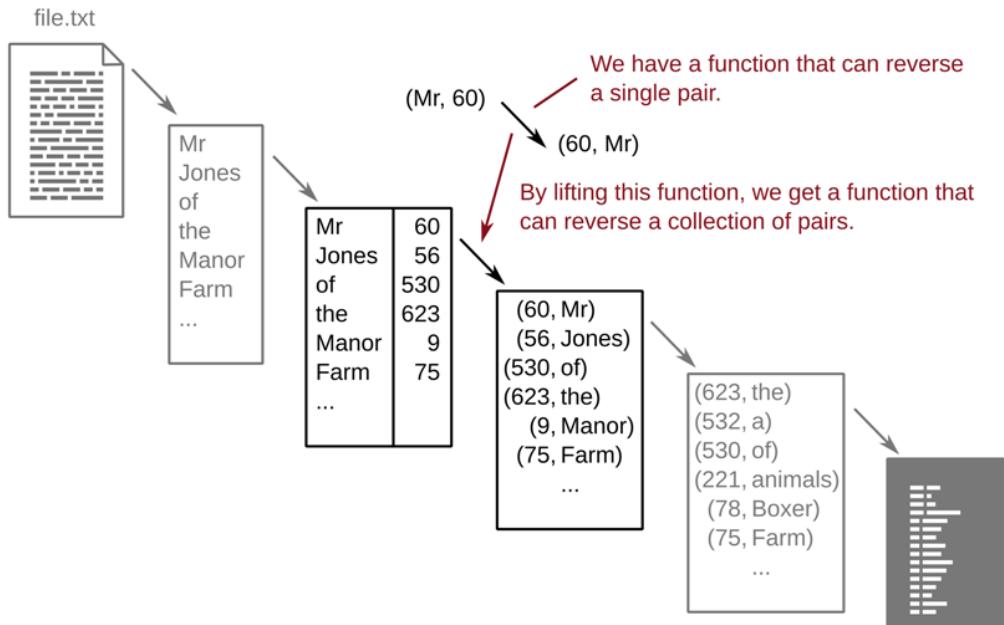
We could easily do the same for any other container-like type, structures that have a few strings in them, smart pointers like `std::unique_ptr` and `std::shared_ptr`, for tokenized input streams, and also to some of the more unorthodox container types that we are going to cover at a later point. This would allow us to always implement the simplest case possible, and then lift it up to the structure we need to use it on.

4.4.1 Reversing a list of pairs

Now that we have seen what lifting is, let's return to the Knuth problem. One of

the transformation steps we had was reversing a collection of pairs, and we used the `reverse_pairs` function for that. Let's look a bit deeper into this function.

Figure 4.14. We have a collection of pairs that we want to reverse and a function that is capable of reversing a single pair. By lifting it, we are getting a function that can reverse a whole collection of pairs



Like in the previous case, we can make it generic to accept any iterable collection (vectors, lists, maps etc.) that contains pairs of items. Instead of modifying the items inside the collection, like in the previous example, we are going to implement it in a pure way. The function will apply the same transformation to all elements (swap the elements in a pair of values), and collect the results in a new collection. We have already said that we can do this with `std::transform`.

Listing 4.19. Lifting a lambda that reverses a single pair of values into a function that does the same for the whole collection of values (example:knuth-problem/main.cpp)

```
template <
    typename C,
    typename P1 = typename std::remove_cv<
        typename C::value_type::first_type>::type,
    typename P2 = typename C::value_type::second_type
>
std::vector<std::pair<P2, P1>> reverse_pairs(const C& items)
{
    std::vector<std::pair<P2, P1>> result(items.size());
```

```

    std::transform(
        std::begin(items), std::end(items),
        std::begin(result),
        [](const std::pair<const P1, P2>& p)
    {
        return std::make_pair(p.second, p.first);
    }
);

return result;
}

```

- ➊ Initializing the type C to be the type of the collection, P1 to be the type of the first item in a pair coming from the source collection (with const removed), and P2 to be the type of second item in the pair
- ➋ We are passing a lambda that reverses values in a single pair to the `std::transform` lifting it to a new function that can perform the same task, but on multiple items in a collection

We have lifted a function that takes a pair of values and returns a new pair with the same values, but in reversed order to work on an arbitrary collection of pairs.

When building more complex structures and objects, we do it by composing simpler types. We can create structures that contain other types, vectors and other collections that contain multiple items of the same type and so on. Since we need these objects to do something, we need to create functions that operate on them (be it free-standing or member functions). Lifting allows us to easily implement these functions for the cases where they just need to pass the torch on to a function that operates on the underlying type.

This means that for each of our more complex types, we just need to create a lifting function, and we will be able to call all the functions that work on the underlying type on our type as well. This is one of the places where the object-oriented and functional paradigms really go hand-in-hand.

In fact, most of what we have seen in this chapter is quite useful in the OOP world as well. We have seen that we can use the partial application to bind a member function pointer to a specific instance of the object whose member it is (Listing 4.11). It might be a nice exercise for the reader to investigate the relation between member functions, and their implicit `this` argument and currying.

4.5 Summary

- Higher-order functions like `std::bind` can be used to transform existing functions into new ones. We can get a function of n arguments, and easily turn it into a unary or a binary function that we could pass to algorithms like `std::partition` or `std::sort` with `std::bind`.
- The placeholders give us a high level of expressiveness when defining which function arguments shouldn't be bound. They allow us to reorder the arguments in

the original function, to pass the same argument multiple times, etc.

- While `std::bind` provides a terse syntax to do partial function application, it might have some performance penalties. You might want to consider using lambdas when writing performance-critical code that is often invoked. They are more verbose, but the compiler will be able to optimize them better than the function objects created by `std::bind`.
- Designing API for your library is hard. There are many different use-cases that need to be covered. Consider creating an API that uses curried functions.
- One of the often heard statements about functional programming is that it allows writing shorter code. If we create functions that are easily composed, we can solve complex problems with a fraction of the code that the usual imperative programming approach would take.

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch04

5

Purity – avoiding mutable state

This chapter covers:

- Problems of writing correct code with mutable state
- What is referential transparency and how it relates to purity?
- Programming without changing values of variables
- What are the situations when mutable state is not evil?
- How can `const` be used to enforce immutability?

We have touched a bit on the topic of immutability and pure functions in the first chapter. We said that one of the main reasons for existence of bugs is that it is hard to manage all different states a program can be in. In the object-oriented world, we tend to approach this issue by encapsulating parts of the program state into objects. We are hiding the actual data behind the class API, and allow modification of that data only through that API.

This allows us to control which state changes are valid and should be performed, and which not. For example, when setting the birth date of a person, we might want to verify that the date is not in the future, that the person isn't older than her parents and so on. It lowers the number of different states our program can be in only to the states we consider valid. While this improves our chances of writing correct programs, it still keeps us open to various problems.

5.1 Problems with the mutable state

Let's check out the following example. We have a `movie_t` class that contains the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/functional-programming-in-cplusplus>

Licensed to Alexey Fadeev <alexey.s.fadeev@gmail.com>

movie name and a list of scores the users gave it. The class has a member function which calculates the average score this movie has.

```
class movie_t {
public:
    double average_score() const;

    // ...

private:
    std::string name;
    std::list<int> scores;
};
```

We have already seen how to implement a function that calculates the average score for a movie in chapter 2, so we can reuse it here. The only difference is that we used to have the list of scores passed in as the parameter of the function, and now it is a member variable in the `movie_t` class.

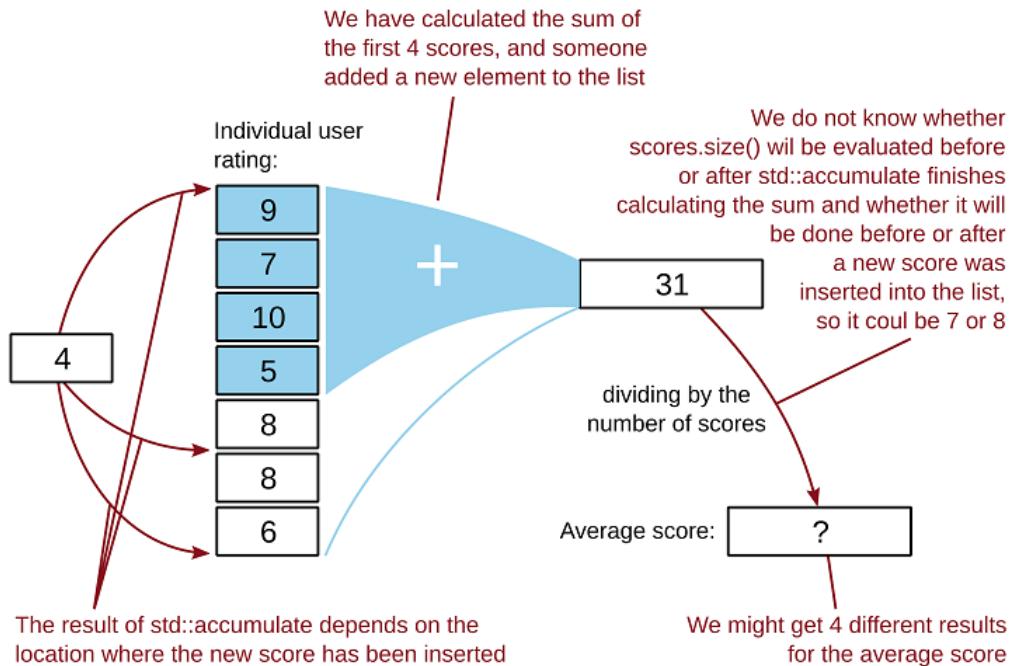
Listing 5.1. Calculating average score for a movie

```
double movie_t::average_score() const
{
    return std::accumulate(scores.begin(),      ①
                          scores.end(),       ①
                          0) / (double)scores.size();
}
```

- ① Calling `begin` and `end` on a `const` value is the same as calling `cbegin` and `cend`

Now the question is whether this code is correct — whether it does what we wanted it to do. We have a sum of all items in a list, and we are dividing it by the number of scores. It appears to be correct.

Figure 5.1. The result of the calculation will depend on the order of evaluation of the division operands and on the place where the new item was inserted while we were calculating the average score



But, what will happen if someone adds a new score to the list *while* we are calculating the average score? Since we are using `std::list`, the iterators won't become invalid, and `std::accumulate` will finish without any errors. It will return the sum of all items it processed. The problem is that the newly added score might be included in that sum, but might also not be, depending on where it was inserted (Figure 5.1). Also, `.size()` might return the old size because C++ doesn't guarantee which argument of the division operator will be calculated first.

This means that we can't guarantee that the result we calculated is correct in all possible cases. It would even be useful to know that it is incorrect in all possible cases (so we could avoid using it, or fix it)—but we don't know that either. Equally bad, the result of this function depends on how other functions in `movie_t` class are implemented (whether adding the score adds it to the beginning or to the end of the list), while it doesn't call any of those functions.

While we need another function to run concurrently (at the same time) with `average_score` for it to fail, this isn't the only case where we might have problems because we allowed changing the contents of the list of scores. It isn't rare after a few years of a project's existence to get a few variables whose values that depend on each other.

For example, before C++11, the `std::list` wasn't required to remember its size, and there was a possibility that the `.size()` member function will need to traverse the whole list in order to calculate the size. A perceptive developer might have seen this and decided to add a new member to our class that saves the number of scores in order to increase the performance of all parts of the code that need to know the number of scores we keep.

```
class movie_t {
public:
    double average_score() const;

    // ...

private:
    std::string name;
    std::list<int> scores;
    size_t scores_size;
};
```

Now, `scores` and `scores_size` are tightly bound, and a change to one should be reflected on the other. This means that they aren't properly encapsulated. They aren't accessible from outside of our class, so the `movie_t` class is properly encapsulated, but inside our class we can do whatever we want with them without the compiler ever complaining. It isn't hard to imagine that some time in the future, there will be a new developer working on the same code who will forget to update the size in some rarely executed corner-case of this class.

In this case, the bug would probably be easy to catch, but bugs like these tend to be much more subtle and difficult to triage. During the evolution of a project, the number of intertwined sections of the code usually tends to grow until someone decides to refactor the project.

Would we have these problems if the list of scores and the `scores_size` member variable were immutable — declared to be `const`? The first issue we had relied on the fact that someone could change the list while we are calculating the average score. If the list is immutable, no one can change it while we are using it, so this issue no longer exists.

What about the second issue? If these variables were immutable, they would have to be initialized on construction of `movie_t`. It would be possible to initialize the `scores_size` to a wrong value, but that mistake would need to be explicit. It can not happen because somebody forgot to update its value — it would have to be because someone wrote incorrect code to calculate it. Another thing is that once the wrong value has been set, it will persist being wrong, the error won't be some special case that is difficult to debug.

5.2 Pure functions and referential transparency

All these issues come from one simple design flaw — having several components in the software system be responsible for the same data without knowing when another component is changing that data. The *simplest* way to fix this is just to forbid changing any data. And all problems we saw will go away.

This is easier said than done. Any communication we have with the user changes some state. If one of our components reads a line of text from the standard input, we have changed the input stream for all other components — they will never be able to read that exact same line of text. If one button reacts to a mouse click, other buttons will (by default) never be able to know that the user clicked on something at all.

Side-effects are sometimes not isolated only to our program. When we create a new file, we are changing the data that other programs in the system have access to — namely the hard disk. This means that we are changing the state of all other components in our software and all programs on the system as a side-effect of wanting to store some data to the disk. This is usually compartmentalized by having different programs access different directories on the file system. But still, we could easily fill up the whole free space on the disk thus making all other programs unable to save anything.

Therefore, if we wanted to commit never to change any state, we wouldn't be able to do anything useful. The only kind of programs we could write are programs that calculate a result based on the arguments we passed in. We couldn't make interactive programs, we couldn't save any data to the disk, nor to a database, we couldn't send network requests etc. This would make our programs rather useless.

Instead of saying that we can't change any state, we are going to see how to design software in a way that keeps mutations and side-effects at a minimum. But first, we need to better understand the difference between pure and non-pure functions. In the first chapter, we said that pure functions are functions that only use the values of the arguments passed to them in order to return the result. They need to have no side-effects that influence any other function in our program, nor any other program in our system. Pure functions also need to always return the same result when called with the same arguments.

We are going to define purity a bit more precisely now through a concept called *referential transparency*. Referential transparency is a characteristic of expressions, not just functions. Let's say that an expression is anything that specifies a computation, and returns a result. We will say that an expression is referentially transparent, if the program wouldn't behave any differently if we just replaced the whole expression with just its return value. If an expression is referentially transparent, it means that it has no *observable side-effects* and therefore, that all functions used in that expression are pure.

Let's see what this means on an example. We are going to write a simple function with a vector of integers as its parameter. The function will return the largest value in that collection. In order to be able later to check that we calculated it correctly, we are going to log the result to the standard output for errors (`std::cerr` — character error stream). We will call this function a couple of times from `main`.

Listing 5.2. Function that searches for the maximum value in a vector, and logs its result to `std::cerr`

```
double max(const std::vector<double>& numbers)
{
    assert(!numbers.empty()); ①
    auto result = std::max_element(numbers.cbegin(),
                                   numbers.cend()); ②
    std::cerr << "Maximum is: " << *result << std::endl; ③
    return *result; ④
}

int main()
{
    // Let's call the max function a few times
    auto sum_max =
        max({1}) +
        max({1, 2}) +
        max({1, 2, 3});

    std::cout << sum_max << std::endl; // writes out 6
}
```

- ① We will assume that the `numbers` vector isn't empty to have the `std::max_element` return a valid iterator

Is the `max` function pure? Are all expressions we use it in referentially transparent — does the program change its behaviour if we replace all calls to the `max` function with the return values?

Listing 5.3. Calls to `max` function replaced by its return values

```
int main()
{
    // Let's call the max function a few times
    auto sum_max =
        1 + ①
        2 + ②
        3; ③

    std::cout << sum_max << std::endl;
}
```

- ① Result of `max({1})`

- ② Result of `max({1, 2})`
- ③ Result of `max({1, 2, 3})`

The main program still calculates and writes out '6'. But overall, it doesn't behave the same. The original version of the program did a bit more — it did write '6' to `std::cout`, but before that, it also wrote out numbers '1', '2' and '3' (not necessarily in that order) to `std::cerr`. So, the `max` function isn't referentially transparent, and therefore it isn't pure.

We said earlier that a pure function only uses its arguments to calculate the result. Our `max` function does use its arguments to calculate the `sum_max` value. But it also uses `std::cerr` which isn't passed in as the function parameter. And furthermore, it doesn't only use `std::cerr`, but it also changes it by writing to it.

When we remove the part of the `max` function that writes to `std::cerr`, it will become pure:

```
double max(const std::vector<double>& numbers)
{
    auto result = std::max_element(numbers.cbegin(),
                                  numbers.cend());
    return *result;
}
```

Now it only uses the `numbers` argument to calculate the result, and it uses `std::max_element` (which is a pure function) to do so.

We now have a more precise definition of a pure function. And we have the precise meaning for the phrase "has no observable side-effects" — as soon as a function call can't completely be replaced by its return value, without changing the behaviour of the program, it has *observable side effects*.

One thing worth noting is that, in order to stay on the pragmatical side of things and not to become overly theoretical, we will again use this definition in a more relaxed manner. It isn't very useful to abandon logging debugging messages just to be able to call our functions "pure". If we don't think that the output we are sending to `std::cerr` is of any real importance to the program's functionality, and that we can write whatever we want to it without anyone noticing or caring about what was written, we could consider our `max` function to be pure even if it doesn't fit the above definition. Namely, when we replace the call to the `max` function with its return value, the only difference to the program execution will be that it writes fewer things to `std::cerr` which we don't care about — we can say that the program behaviour didn't change.

5.3 Programming without side-effects

It isn't enough just to say "don't use mutable state and you will live happily ever after", since it isn't really obvious nor intuitive how we can do it. We have been

taught to think about software (heck, we were taught to think about real life) like it was all one big state machine that is changing all the time. The usual example for this is — we have a car which isn't running. Then we turn the key or press the 'start' button, and it goes into a new state — it is *switching on*. When the *switching on* state finishes, it goes into the *running* state.

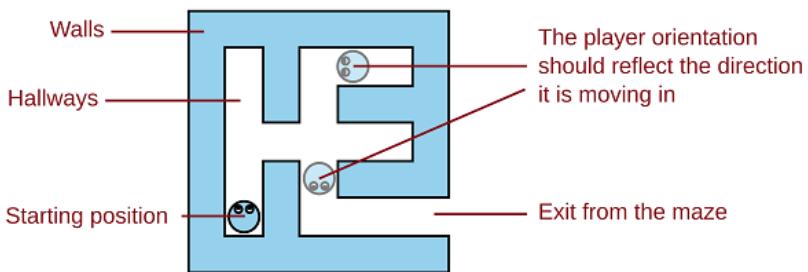
In pure functional programming, instead of changing a value, we create a new one. Instead of changing a property of an object, we are creating a copy of that object in which just the value of that property is changed to the new value. When designing software like this, it isn't possible to have the issues like in the previous example where we had someone change the movie scores while we are processing them because once we have a list of movie scores, nobody can change it — they can only create new lists with the new scores added.

5.3.1 Creating new worlds instead of changing the current one

This approach sounds counter-intuitive at first, and contrary to what we usually think of how the world works. But strangely enough, the idea behind it was a part of the science fiction scene for decades now, and even was discussed by some of the ancient Greek philosophers. There is a popular notion that we live in one of the many parallel worlds. That every time we make a choice, we in fact just create a new world in which we made *that* exact choice, and a few parallel worlds in which we made different choices.

Now, when programming, we don't usually care about all possible worlds, we care just about a single one. And that is why we think we are changing a single world instead of creating new ones all the time and discarding the previous ones. It is an interesting notion, and regardless of whether it is how the world works or not, it does have its merits when developing software.

Figure 5.2. The maze is defined by a matrix where each cell can be a wall or a hallway. The player can move around, but it can't go through the walls.



Let's demonstrate this on a small example. Imagine we have a small game where the user tries to get out of a maze. The maze is defined with a square matrix where each field can be `Hallway` if it is a hallway, `Wall` if it is a wall, `Start` if it is a starting point, and `Exit` if it is an exit from the maze. We want our game to look

pretty, so we will want to show different images depending on direction in which the player is moving.

If we wanted to implement this in a imperative way, the program logic would look something like this (minus handling the *pretty graphics* — player orientation and transitions):

Listing 5.4. Moving around the maze

```
while (1) {
    - draw the maze and the player
    - read user input
    - if we should move (the user pressed one of the arrow keys)
        check whether the destination cell is a wall,
        and move the player to the destination cell if it isn't
    - check whether we have found the exit,
        show a message if we have,
        and exit the loop
}
```

This is a perfect example of when mutable state is useful. We have a maze and a player, and we are moving the player around the maze thus changing its state. Let's see how we would model this in a world of immutable state.

First, we can notice that the maze never changes, and that the player is the only thing in our program that changes its state. This means that we need to focus on the player, and that the maze could easily be an immutable class.

In the each step of the game, we need to *move* the player to the new position. What data is needed so that we could calculate the new position?

- We need to know in which direction we should move;
- Then, we need to know the previous position in order to know where are we moving from;
- The last one is the maze — we need to know whether we can move to the desired position or not.

So, we need to create a function that gets has these three parameters, and returns the new position.

Listing 5.5. Function to calculate the next position of the player

```
position_t next_position(
    direction_t direction,
    const position_t& previous_position,
    const maze_t& maze
)
{
    // We are calculating the desired position
    // even if it might be a wall
```

```

const position_t desired_position(position_t previous_position, direction);

// If the new position is not a wall,
// we are going to return it,
// otherwise we will return the old one
return maze.is_wall(desired_position) ? previous_position
                                         : desired_position;
}

```

We have implemented the moving logic without a single mutable variable. We need to define the `position_t` constructor which calculates the coordinates of the neighbouring cell in the given direction.

Listing 5.6. Calculating the coordinates of the neighbouring cell when given the cell and the direction

```

position_t::position_t(const position_t& original,
                      direction_t direction)
    : x { direction == Left   ? original.x - 1 : ①
          direction == Right  ? original.x + 1 : ①
                                original.x       } ①
    , y { direction == Up    ? original.y + 1 : ①
          direction == Down  ? original.y - 1 : ①
                                original.y       } ①
{
}

```

- ① We are using the ternary operator to be able to match against the possible direction values and initialize the correct values of `x` and `y`. We could have also done it with a switch statement in the body of the constructor

Now that we have the logic nailed down, we need to know how to show the player. We have two choices here — we can change the player orientation when it successfully moves to another cell, or we could change the orientation even if we haven't moved to show the user that we did understand the command, but are unable to comply because we encountered a wall there. This is a part of the presentation layer, and is not something that we need to consider as the integral part of the player — both these choices are equally valid.

In the presentation layer, we just need to be able to draw the maze and the player inside. Since the maze is immutable anyhow, showing it to the user won't change anything. The function which draws the player also doesn't need to change it. It needs to know the position of the player, and to which direction it should be turned to.

```

void draw_player(const position_t& position,
                 direction_t direction)

```

We mentioned that we have two choices when showing the which way the player is

facing. One is based on whether the player was actually moved — this can be easily calculated since always create new instances of `position_t`, and we have the previous position unchanged. And the second option which changes the player orientation even if it didn't move can easily be implemented just by orienting it in the same direction we passed to the `next_position` function.

We can also take this further, we could choose different images depending on the current time, how long since the player was last moved, etc.

Now, we just need to tie this logic together. We are going to implement the event loop recursively just for the demonstration purposes. We said in the chapter 2 that we do not want to overuse recursion because it isn't guaranteed that the compiler will optimize it out when compiling our program. We will use recursion in this example only to demonstrate that the whole program can be implemented in a pure way — without ever changing any object after it is created.

Listing 5.7. Recursive event loop

```
void process_events(const maze_t& maze,
                    const position_t& current_position)
{
    if (maze.is_exit(current_position)) {
        // show message and exit
        return;
    }

    const direction_t direction = ...;           ①

    draw_maze();                                ②
    draw_player(current_position, direction);    ②

    const auto new_position = next_position(   ③
        direction,
        current_position,
        maze);                                     ③
                                                ③
                                                ③
    process_events(maze, new_position);          ④
}

int main()                                     ⑤
{
    const maze_t maze("maze.data");            ⑤
    process_events(                            ⑤
        maze,
        maze.start_position());                ⑤
}
```

- ① Calculating the direction based on the user input
- ② Showing the maze and the player
- ③ Getting the new position

- ④ Continuing processing the events, but now with the player moved to the new cell
- ⑤ The main function only needs to load the maze, and call the process_events function with the initial position for the player

This example shows a common way to model stateful programs while avoiding any state changes and keeping the functions pure. In the real-world programs, things are usually much more complicated than this. Here the only thing that needs changing is the position of the player. Everything else, including how the player is shown to the user can be calculated from the way the position changes.

In larger systems, there are many movable parts. We could create a huge all-encompassing *world* structure, that we would recreate every time we need to change something in it. This would have a big performance overhead (even if we use data structures optimized for functional programming which we will cover in chapter 8) and would significantly increase the complexity of our software.

Instead, we will usually have *some* mutable state for which we think would be inefficient to always have to copy and pass around, and will model our functions so that they return statements about what should be changed in the world instead of always returning new copies of the world. What this approach brings to the table is a clear separation between mutable and pure parts of the system.

5.4 **Mutable and immutable state in a concurrent environment**

Most of today's software systems have some kind of concurrency in them. Be it splitting complex calculations like image processing into a few parallel threads so that the whole job would be finished faster, or having different tasks performed at the same time like a web browser downloading a file while the user browses a web page.

Mutable state can lead to problems because it allows shared responsibility (which is even against the best OOP practices), and it is elevated in concurrent environments because the responsibility can be shared by multiple components **at the same time**.

A minimal example that shows the essence of this problem is creating two concurrent processes that try to change the value of a single integer variable. Imagine we want to count the number of clients currently connected to our server, so that we could go into the power-save mode when everyone disconnects. We start with two connected clients, and both of them disconnect at the same time.

```
void client_disconnected(const client_t& client)
{
    // Free the resources used by the client
    ...

    // Decrement the number of connected clients
    connected_clients--;
}
```

```

if (connected_clients == 0) {
    // go to the power-save mode
}
}

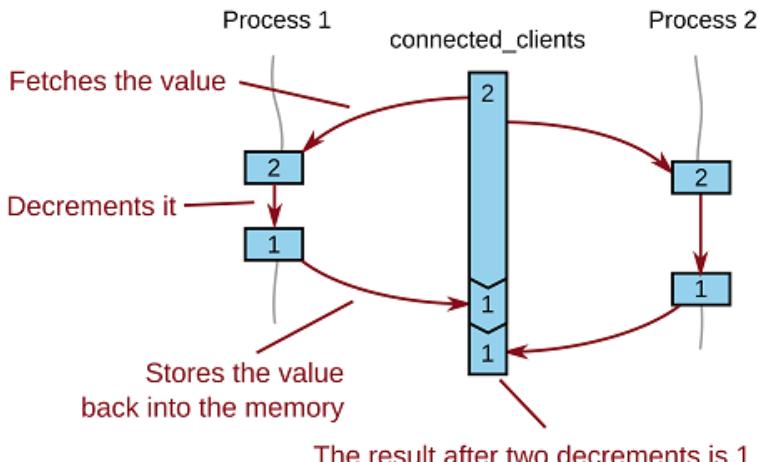
```

We will just focus on the `connected_clients--` line. It seems rather innocent. At the beginning, if all is well, `connected_clients == 2` since we have two clients. When one client disconnects, it will become 1, and when the second one disconnects, it will become 0 and we will go to the power-save mode. The problem is that even if it looks like a simple statement to us, it isn't for the computer. It needs to:

1. get the value from the memory to the processor;
2. decrement the retrieved value;
3. and it needs to store the changed value back into memory.

As far as the processor is concerned, these steps are separate and it doesn't care that it is a single command from our point of view. If the `client_disconnected` function is called twice at the same time these three steps from one call can be interleaved in any way imaginable with the steps from the second call. This means that one call might retrieve the value from the memory even before the other one has finished changing it. In that case, we will end up with the value decremented only once (or to be more precise, it will be decremented from 2 to 1 two times) and it will never go back to zero.

Figure 5.3. We have two concurrent processes that want to decrement a value. Each of them needs to retrieve the value from the memory, decrement that value, and store it back into the memory. Since the initial value is 2, we are expecting the result after decrementing it twice to be 0. But, if the second process is allowed to read the value from the memory before the first process has stored its result back into the memory, we might end up with a wrong result.



If we get problems with the concurrent access to a single integer, imagine what problems will we get with more complex data structures. Fortunately, this problem was solved a long time ago. People realized that it is problematic to allow changing data when there is a possibility of concurrent access to that data. And the solution was quite simple — to allow the programmer to forbid concurrent access to the data with *mutexes*.

The problem with this approach is that we **want** to have things running in parallel — be it for efficiency or something else. And *mutexes* solve the problem with concurrency by removing concurrency.

I've often joked that instead of picking up Djikstra's cute acronym (mutex — which stands for mutual exclusion) we should have called the basic synchronization object "the bottleneck". Bottlenecks are useful at times, sometimes indispensable—but they're never GOOD. At best they're a necessary evil. Anything. ANYTHING that encourages anyone to overuse them, to hold them too long, is bad. It's NOT just the straight-line performance impact of the extra recursion logic in the mutex lock and unlock that's the issue here—it's the far greater, wider, and much more difficult to characterize impact on the overall application concurrency.

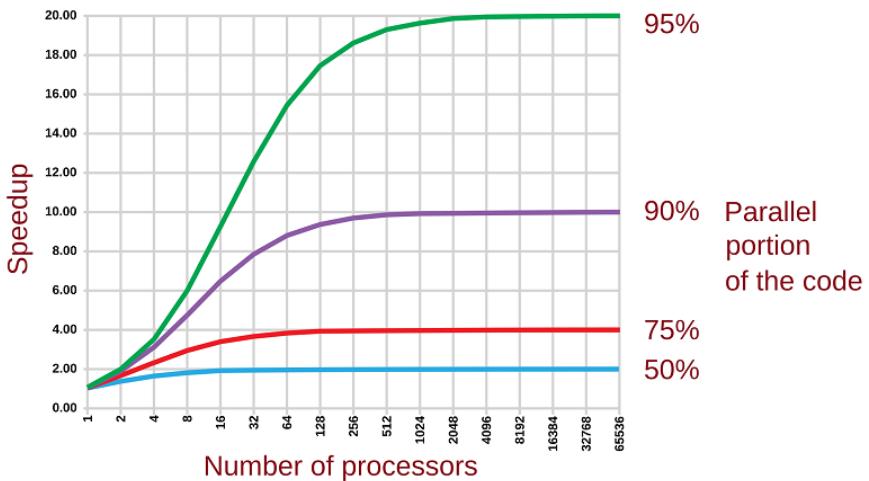
-- David Butenhof on comp.programming.threads

Mutexes are necessary sometimes⁸, but they should be used rarely, and not as an excuse for bad software design. Mutexes, like for loops and recursion, are low-level constructs that are useful for implementing higher-level abstractions for concurrent programming, but aren't something that should appear too often in normal code.

It might sound strange, but according to Amdahls's law, if we have only 5% of our code serialized (and 95% fully parallelized) the maximum speedup we can expect compared to not having anything parallelized is 20 times. No matter how many thousands of processors we throw at the problem.

⁸ You can check out the *C++ Concurrency in action* book by Anthony Williams for more details about concurrency, mutexes and how to write multithreaded code in the usual way. We will cover functional programming techniques for writing concurrent and asynchronous in Chapters 10 and 12.

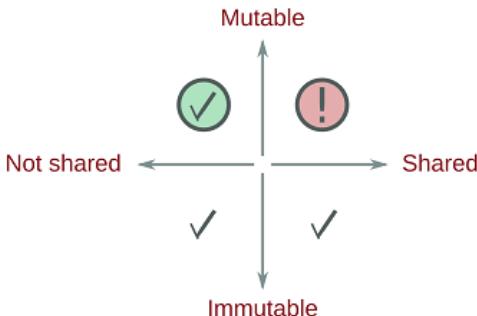
Figure 5.4. It would be ideal to have a ten-fold speedup when increasing the number of processors ten times. Unfortunately, the speedup doesn't increase linearly with the number of processors. According to Amdahl's law, if you have only 5% of your code serialized (with mutexes or through other means) the most speedup you can achieve compared to running the code on a single processor is 20. Even if you throw in 60 thousand processors.



If lowering the level of parallelism our code can achieve isn't what we want, we need to find alternative solutions to mutexes for the concurrency problem. Remember that we said that the problems only appear if we have mutable data shared across different concurrently running processes. Therefore, one solution isn't to have concurrency, and another not to use mutable data.

But there is also a third option. To have mutable data, but not share it. If we don't share our data, we will never have a problem that someone has changed it without us knowing about it.

Figure 5.5. We can have both mutable and immutable state, we can share it or not. The only option which will produce problems in the concurrent environment is when the data is both shared and mutable. Other three are safe.



We have four options — to have immutable data and not to share it, to have mutable data and not to share it, to have immutable data and to share it, and to have mutable data and to share it. Of all four, only the last situation is problematic. And yet, it seems like 99% of all software somehow belongs to that one.

We will talk further on this topic in chapters 10 and 12 where we will see the FP abstractions that make concurrent and asynchronous programming, both with and without mutable data almost as easy as ordinary functional programming.

5.5 ***The importance of being const***

Now that we have seen some of the problems that arise from having mutable state, let's investigate what tools C++ provides us to facilitate the development of code without mutable state. We have two ways of restricting mutation in C++, the `const` and `constexpr` keywords (with the latter being available since C++11).

The basic idea behind `const` is to prevent data mutation. The way it is done in C++ is through its type system — when you declare an instance of some type `T` to be `const`, you are actually declaring the type to be `const T`. This is a really simple concept with far-reaching consequences. Say we have declared a constant `std::string` variable called `name` and we want to assign it to another variable.

```
const std::string name{"John Smith"};

std::string name_copy          = name;
std::string& name_ref          = name; // error
const std::string& name_constref = name;
std::string* name_ptr          = &name; // error
const std::string* name_constptr = &name;
```

The first assignment tries to assign our constant string `name` to a non-constant string variable called `name_copy`. This will work — a new string will be constructed, and it will hold a copy of the data `name` contains. This is OK because if we later change the value of `name_copy`, the original string won't be changed.

The second assignment creates a reference to a `std::string` and tries to set it to refer to the `name` variable. When we try to compile it, we will get an error because we are trying to create a reference to a `std::string` from a variable that has a different type — `const std::string` (and there is no inheritance here which would allow us to do so). We get a similar error when trying to assign the address of `name` to `name_ptr` which is a pointer to a non-constant string. This is the desired behaviour because if the compiler allowed us to do this, the `name_ref` variable would allow us to call functions on it that would change its value, and since it is only a reference to `name`, the changes would actually be performed on the `name` variable. The compiler stopped us even before we even tried to change the value of `name`.

The third assignment creates a reference to `const std::string` from a variable whose type is exactly that. It will pass without any errors. The assignment to a pointer to a constant string (`name_constptr`) will also work.

Constant references are special

While the fact that the compiler allows us to create a reference to `const T` from a variable that has the same type is something that is to be expected, constant references (references to constant types) are special.

Apart from being able to bind to a variable of type `const T`, a constant reference can also bind to a temporary value, in which case it prolongs its lifetime.

For more information on the lifetime of temporary objects, you can check out

en.cppreference.com/w/cpp/language/lifetime#Temporary_object_lifetime

Next, what happens when we try to pass this constant string as an argument to a function? We have `print_name` function that accepts a string as its parameter, the `print_name_ref` function that accepts a reference to a string, and the `print_name_constref` that accepts a constant reference.

```
void print_name(std::string name_copy);
void print_name_ref(std::string& name_ref); // error when called
void print_name_constref(const std::string& name_constref);
```

Similarly to the previous case, it is OK to assign a value to a string variable from a constant string (a copy of the original string will be created), and to assign a string value to a constant reference to a string because we are preserving constness of the `name` variable in both cases. The only one that isn't allowed is assigning it to a non-`const` reference.

We have seen how `const` works with variable assignment and ordinary functions. Let's see how it relates to member functions. We will use the `person_t` class and check out how C++ enforces constness.

Consider the following member functions:

```
class person_t {
public:
    std::string name_nonconst();
    std::string name_const() const;

    void print_nonconst(std::ostream& out);
    void print_const(std::ostream& out) const;

    // ...
};
```

When we write member functions like these, C++ actually sees normal functions

with an additional implicit argument — `this`. The `const` qualifier on a member function is nothing more than a qualifier for the type that `this` is pointing to. Therefore, the above member functions would internally look something like this (in reality, `this` is a pointer, but we are going to pretend it is a reference for the sake of clarity):

```
std::string person_t_name_nonconst(
    person_t& this
);
std::string person_t_name_const(
    const person_t& this
);
void person_t_print_nonconst(
    person_t& this,
    std::ostream& out
);
void person_t_print_const(
    const person_t& this,
    std::ostream& out
);
```

When we start looking at member functions as if they were ordinary functions with an implicit parameter `this`, the meaning of calling member functions on constant and non-constant objects becomes obvious — it is exactly the same as in the previous case where we tried to pass constant and non-constant objects to ordinary functions as arguments. Trying to call a non-constant member function on a constant object will yield an error because we are trying to assign a constant object to a non-constant reference.

Why is `this` a pointer?

`this` is a constant pointer to an instance of the object the member function belongs to, and, according to the language rules, it can never be null (though, in reality, some compilers allow calling non-virtual member functions on null pointers).

The question is then, if `this` can't be changed to point to another object, and it can't be null, why isn't it a reference instead of a pointer?

The answer to this is rather simple — when `this` was introduced, C++ didn't have references yet. It was impossible to change this later since it would break back-compatibility with the previous language versions and break a lot of code.

As we have seen, the mechanism behind `const` is rather simple, but it works surprisingly well. By using `const`, we are telling the compiler that we don't want something to be mutable, and that it should give us an error any time we try to change it.

5.5.1 Logical and internal constness

We said that we want to implement immutable types to avoid the problems of mutable state. Probably the simplest way to do it in C++ is to create classes in which all members are constant. Similar to implementing the `person_t` class like this:

```
class person_t {
public:
    const std::string name;
    const std::string surname;

    // ...
};
```

We don't even need to create accessor member functions for these variables, we can just make them public. This approach has a downside — some compiler optimizations will stop working. As soon as we declare a constant member variable, we lose the move constructor and the move assignment operator (see Appendix A for more information about move semantics).

We will choose a different approach — instead of making all member variables constant, we will make all (public) member functions constant.

```
class person_t {
public:
    std::string name() const;
    std::string surname() const;

private:
    std::string m_name;
    std::string m_surname;

    // ...
};
```

This way, while the members aren't declared as `const`, it won't be possible for the user of our class to change them because `this` will point to an instance of `const person_t` in all member functions we implemented. This provides us both with *logical constness* (the user visible data in the object never changes) and *internal constness* (no changes to the internal data of the object). At the same time, we won't lose any optimization opportunities because the compiler will generate all the necessary move operations for us.

The problem we are left with is that sometimes we need to be able to change our internal data but to hide those changes from the user. As far as the user is concerned, we need to be immutable. One of the examples where this is necessary is when we want to cache the result of some time-consuming operation.

Say we want to implement the `employment_history` function that returns the list of

all previous jobs a person had. This function will need to contact the database in order to be able to retrieve this data. Since all member functions are declared as `const`, the only place where we can initialize all member variables is in the `person_t` constructor. Querying the database isn't a particularly fast operation, so we don't want to do it every time a new instance of `person_t` is created if it isn't useful for all users of this class. At the same time, for the users who do need to know the employment history, we don't want to query the database every time the `employment_history` function is called. This would be problematic not only because of performance, but also because this function might return different results when called multiple times (if the data in the database changes) which would ruin our promise that `person_t` is immutable.

This can be solved by creating a `mutable` member variable in our class — that is, a member variable that can be changed even from `const` member functions. Since we want to guarantee that our class is immutable from the user's perspective, we have to ensure that two concurrent calls to `employment_history` can't return different values — we need to ensure that the second call doesn't get executed until the first one finishes loading the data.

Listing 5.8. Using `mutable` to implement caching

```
class person_t {
public:
    employment_history_t employment_history() const
    {
        std::unique_lock<std::mutex> lock{m_employment_history_mutex}; ①

        if (!m_employment_history.loaded()) { ②
            load_employment_history(); ②
        } ②

        return m_employment_history; ③
    }

private:
    mutable std::mutex m_employment_history_mutex; ⑤
    mutable employment_history_t m_employment_history; ⑤

    // ...
};
```

- ① We are locking the mutex to guarantee that another concurrent invocation of `employment_history` function can't be executed until we finish retrieving the data from the database
- ② If the data isn't already loaded, let's get it
- ③ The data is loaded, we are returning it to the caller
- ④ When we exit this block, the lock variable will be destroyed and the mutex will be unlocked

- 5 We want to be able to lock the mutex from a constant member function, so it needs to be mutable as well as the variable we are initializing

This is the usual pattern of implementing classes that are immutable on the outside, but that sometimes have the need to change internal data. It is required that all constant member functions either keep the class data unchanged or that all changes are synchronized (unnoticeable even in the case of concurrent invocations).

5.5.2 Optimizing member functions for temporaries

When we design our classes to be immutable, whenever we want to create a *setter* member function, we need to create a function that returns a copy of our object in which a specific member value is changed instead. Creating copies all the time just to get a modified version of an object isn't efficient (though the compiler might be able to optimize this in some cases). This is especially true when we don't need the original object anymore.

Imagine we have an instance of `person_t` and we want to get an updated version with the name and surname changed. We would have to write something like this:

```
person_t new_person {
    old_person.with_name("Joanne")
        .with_surname("Jones")
};
```

The `.with_name` function will return a new instance of `person_t` with the name "Joanne". Since we aren't assigning that instance to a variable it will be destroyed as soon as this whole expression is calculated. We are just calling `.with_surname` on it, which creates yet another instance of `person_t` that now has both the name "Joanne" and surname "Jones".

This means that we will construct two separate instances of `person_t` and copy all the data that `person_t` holds twice, whereas we only wanted to create a single person — to assign to the `new_person` variable.

It would be better if we could avoid this by detecting that the `.with_surname` has been called on a temporary object, and that we don't need to create a copy of it, but that we can just move the data from it into the result.

Fortunately, we can do this by creating two separate functions for each of `.with_name` and `.with_surname` — one that works on proper values, and one for temporaries:

Listing 5.9. Separating member functions for normal values and temporaries

```
class person_t {
public:
    person_t with_name(const std::string& name) const & ①
{
```

```

    person_t result(*this);          ②

    result.m_name = name;          ③

    return result;                ④
}

person_t with_name(const std::string &name) &&
{
    person_t result(std::move(*this));  ⑤

    result.m_name = name;            ⑥

    return result;                ⑦
}

};


```

- ① This member function will work on normal values and lvalue references
- ② We are creating a copy of the person
- ③ Setting the new name for that person. It is Ok to change this name here since this `person_t` instance is still not visible from the outside world
- ④ We are returning the newly created `person_t` instance. From this point on, it is immutable
- ⑤ This one will be called on temporaries and other rvalue references
- ⑥ Calling the move constructor instead of copy constructor
- ⑦ Setting the name
- ⑧ Returning the newly created person

We are declaring two overloads of the `.with_name` function. Just like we saw with the `const` qualifier, specifying the reference type qualifier on a member function really only affects the type of `thispointer`. The second overload will be called on objects for which we are allowed to steal the data from — temporary objects and other rvalue references.

When the first overload is called, we will create a new copy of the `person_t` object pointed to by `this`, set the new name and return the newly created object. In the second overload, we will create a new instance of `person_t` and move (instead of copying) all the data from the `person_t` object pointed to by `this` into the newly created instance.

Note that the second overload isn't declared to be `const`. If it was, we wouldn't be allowed to move the data from the current instance to the new one. We would need to copy it like in the first overload.

Here we have seen one of the possible ways to optimize our code to avoid making copies of temporary objects. Like all optimizations, this one should be used only when it actually makes a measurable improvement — we should always avoid premature optimizations, and perform benchmarks whenever we decide to optimize something.

5.5.3 Pitfalls with `const`

The `const` keyword is one of the most useful things in C++ and you should design your classes to use it as much as possible, even when not programming in the functional style. Just like the rest of the type system, it makes a lot of common programming errors detectable during compilation.

But it isn't all fun and games. There are a few pitfalls one might encounter when using `const` too much.

CONST DISABLES MOVING THE OBJECT

When writing a function that returns a value of some type (not a reference or a pointer to a value), we often define a local variable of said type, do something with it, and return it. That is what we have done in both of our `.with_name` overloads earlier. Let's extract the essence of this pattern:

```
person_t some_function()
{
    person_t result;

    // do something before returning the result

    return result;
}

// ...

person_t person = some_function();
```

If the compilers didn't perform any optimizations when compiling this code, `some_function` would create a new instance of `person_t` and return it to the caller. In this case, that instance would be passed on to the copy (or move) constructor in order to initialize the `person` variable. When the copy (or move) is made, the instance that `some_function` returned would be deleted.

Just like in the previous case where we had consecutive calls to `.with_name` and `.with_surname`, we create two new instances of `person_t`, one of which is temporary and will be deleted immediately after the `person` was constructed.

Fortunately, compilers tend to optimize this to make `some_function` construct its local `result` variable directly into the space reserved by the caller for the `person` variable thus avoiding any unnecessary copies or moves. This optimization is called *named return value optimization* (or NRVO for short).

This is one of the rare cases where using `const` might actually hurt us. In the cases when the compiler can't perform NRVO, returning a `const` object will incur a copy instead of being able to just move it into the result.

SHALLOW CONST

Another problem is that `const` can easily be subverted. Imagine the following situation, we have a `company_t` class which keeps a vector of pointers to all its employees. And we have a constant member function that returns a vector containing the names of all employees.

```
class company_t {
public:
    std::vector<std::string> employees_names() const;

private:
    std::vector<person_t*> m_employees;
};
```

The compiler would forbid us to call any non-constant member functions of `company_t` and `m_employees` from the `employees_names` member function because it is declared `const`. So, we are forbidden to change any data in this instance of `company_t`. But are the `person_t` instances that represent the employees data a part of this instance of `company_t`? They aren't. Only the pointers to them are. This means that we aren't allowed to change the pointers from the `employees_names` function, but we are allowed to change the objects those pointers point to.

This is a tricky situation which might lead us into trouble if we hadn't designed the `person_t` class to be immutable. If there were methods that mutate `person_t`, we would be allowed to call them from `employees_names` which is something that would be nice if the compiler guarded us against since `const` should be a promise that we aren't changing anything.

The `const` keyword gives us the ability to declare our intent that an object shouldn't be modified, and gets the compiler to enforce this constraint. While it is a bit tedious to write it all the time, if we do, every time we see a non-`const` variable, we will just *know* it is meant to be changed. In that case, every time we want to use the variable, we need to check all the places where it could have been modified. If it was declared as `const` we would be able just to check its declaration and know exactly what is its value when we decide to use it.

The propagate_const wrapper

There is a special wrapper for pointer-like objects called `propagate_const` which is able to remedy this problem.

Currently, it is published under the *Library fundamentals technical specification* and should become a part of some future C++ standard after C++17.

If your compiler and STL vendor provides experimental library features like these, you'll be able to find it in the `<experimental/propagate_const>` header. Like with all things that live in

the `experimental::` namespace, the API and behaviour of this wrapper might change in the future revisions, so be warned.

Anyhow, if you are lucky enough to be able to use it, it is as simple as wrapping types of all the pointer member variables (and other pointer-like types such as smart pointers) in your class with `std::experimental::propagate_const<T*>`. It will detect when you use it from a constant member function and in that case it will behave like a pointer to `const T`.

5.6 Summary

- Most computers have multiple processing units in them. When writing programs, we need to pay attention that our code works correctly in multithreaded environments.
- If we overuse mutexes, we will limit the level of parallelism that our program can achieve. Because of the need for synchronization, program speed doesn't grow linearly with the number of processors we are running it on.
- Mutable state isn't necessarily bad; we are safe to have it as long as it isn't shared between multiple system components at the same time.
- When you make a member function to be `const`, you promise that said function won't change any data in the class (the object will remain bit-by-bit the same), or that any changes to the object (to members declared as `mutable`) will be atomic as far as the users of the object are concerned.
- Copying the whole structure when we want just to change a single value in it isn't efficient. We can use special immutable data structures to remedy this (we will talk about immutable data structures in chapter 8).

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch05



Lazy evaluation

This chapter covers:

- Calculating values when they are needed
- Caching values of pure functions
- Modifying the quicksort algorithm to sort only parts of the collection
- Using expression templates for lazy evaluation of expressions
- Representing infinite or near-infinite structures

Calculations take time. Say we have two matrices — A and B , and we are told we might need their product at some point. One thing that we could do is to immediately calculate the product, and it will be ready for when we need it.

```
auto P = A * B;
```

The problem is that we might end up not needing the product at all — and we wasted our precious CPU cycles calculating it.

An alternative approach is just to say that if someone needs it, P should be calculated as $A * B$. Instead of actually doing the calculation, we are just defining it. When a part of our program needs the value of P , we will calculate it. But not before.

We usually define calculations by creating functions. So, instead of P being a value that holds the product of A and B , we can turn it into a lambda that captures A and B , and returns their product when invoked:

```
auto P = [A, B] {
    return A * B;
};
```

If someone needs the value, they just need to call it like a function `P()`.

We have optimized our code for the case when we have a complex calculation whose result might not be needed. But we have created a new issue — what if said value is needed more than once? With this approach, we would need to calculate the product every time it is needed. Instead, it would be better to remember the result the first time it is calculated.

This is what being lazy is all about — instead of doing the work in advance, we are postponing it as much as possible, and since we are lazy, we want to avoid doing the same thing multiple times, so once we get the result, we are going to remember it.

6.1 Laziness in C++

Unfortunately, C++ does not support lazy evaluation out of the box like some other languages do, but it does provide us with enough tools that we can use to simulate this behaviour in our programs.

We are going to create a template class called `lazy_val` which will be able to store a computation, and which will cache the result of that computation once it is executed (this is often also called *memoization*). Therefore, this type needs to be able to hold the following:

- the computation;
- a flag indicating whether we already have calculated the result;
- and the calculated result.

We said in the chapter 2 that the most efficient way to accept an arbitrary function object is as a template parameter. We will follow that advice in this case as well — the `lazy_val` will be templated on the type of the function object (the definition of the computation). We do not need to specify the resulting type as a template parameter because it can be easily deduced from the computation.

Listing 6.1. Member variables of the `lazy_val` class template

```
template <typename F>
class lazy_val {
private:
    F m_computation;                                1
    mutable bool m_cache_initialized;                 2
    mutable decltype(m_computation()) m_cache;        3
    mutable std::mutex m_cache_mutex;                 4
public:
```

```
// ...
};
```

- ❶ Storing the function object that defines the computation
- ❷ We need to be able to tell whether we have already cached the computation result
- ❸ Cache for the computed result. The type of the member variable is the return type of the computation
- ❹ We need the mutex in order to stop multiple threads from trying to initialize the cache at the same time

We are going to make the `lazy_val` template class immutable, at least when looked at from the outside world. This means that we will mark all member functions we create as `const`. Internally, we need to be able to change the cached value once we first calculate it, so all cache-related member variables need to be declared as `mutable`.

There is a downside to the fact that we have chosen to have the computation type as the template parameter for `lazy_val`, because automatic template deduction for types is supported only since C++17. Most of the time we will not be able to specify it explicitly because we will be defining the computation through lambdas, and we can not write the type of a lambda. Because of that, we will need to create a `make_lazy_val` function because automatic template argument deduction works for them, so that we can use the `lazy_val` template with compilers not supporting C++17.

Listing 6.2. Constructor and a factory function

```
template <typename F>
class lazy_val {
private:
    // ...

public:
    lazy_val(F computation)
        : m_computation(computation)
        , m_cache_initialized(false) ❶
    {
    }
};

template <typename F>
inline lazy_val<F> make_lazy_val(F& computation) ❷
{
    return lazy_val<F>(std::forward<F>(computation)); ❷
}
```

- ❶ We are initializing the computation definition and noting that the cache is not initialized yet
- ❷ Convenience function that creates a new `lazy_val` instance automatically deducing the type of the computation

The constructor does not need to do anything except to store the computation and set the flag that indicates whether we already have calculated the result to false.

Note **Supporting results that are not default-constructible**

In this implementation, we are requiring that the result type of the computation is default-constructible — we have the `m_cache` member variable which we do not assign an initial value to, so the constructor of `lazy_val` will implicitly call the default constructor for it.

This is a limitation of this particular implementation, and not a requirement for the lazy value concept itself. The full implementation of this class template which does not have this problem can be found in the code example `06-lazy-val`.

The last step here is to create a member function that returns a value — that either calculates and caches it, or the one that returns the cached version. We can either define the call operator on `lazy_val` thus making it a function object, or we can create a casting operator which would allow it to look like a normal variable. Both approaches have their advantages, and choosing one over another is mostly a matter of taste. Will go for the latter.

The implementation is straight-forward — we are locking the mutex so that nobody can touch the cache until we are finished with it, and if the cache is not initialized, we initialize it to whatever the stored computation returns.

Listing 6.3. Casting operator for the `lazy_val` class template

```
template <typename F>
class lazy_val {
private:
    // ...

public:
    // ...

    operator const decltype(m_computation())& () const ①
    {
        std::unique_lock<std::mutex>          ②
            lock{m_cache_mutex};              ②

        if (!m_cache_initialized) {          ③
            m_cache = m_computation();      ③
            m_cache_initialized = true;     ③
        }

        return m_cache;
    }
};
```

- ① We are allowing implicit casting an instance of `lazy_val` to the const-ref of the return type of the computation
- ② We are forbidding concurrent access to the cache

③ We are caching the result of the computation for later use

If you are well-versed writing multi-threaded programs, you might have noticed that this implementation is suboptimal. Namely, every time the program needs the value, we are locking and unlocking a mutex while we only need to lock the `m_cache` variable the first time the function is called — while we are calculating the value.

Instead of using a mutex, which is a general low-level synchronization primitive, we can use something that is tailored exactly for our use-case.

The casting operator does two things — it initializes the `m_cache` variable and it returns the value stored in that variable, where the initialization needs to be performed only once per instance of `lazy_val`. So we have a part of the function that needs to be executed only the first time the function is called. The standard library provides us with a solution for this — the `std::call_once` function:

```
template <typename F>
class lazy_val {
private:
    F m_computation;
    mutable decltype(m_computation()) m_cache;
    mutable std::once_flag m_value_flag;

public:
    // ...

    operator const decltype(m_computation())& () const
    {
        std::call_once(m_value_flag, [this] {
            m_value = m_computation();
        });

        return m_cache;
    }
};
```

We have replaced the mutex and the Boolean `m_cache_initialized` indicator we had in the previous implementation with an instance of `std::once_flag`. When the `std::call_once` function is invoked, it will check whether the flag is set or not, and if it isn't, it will execute the function that we have passed to it — the function that initializes the `m_value` member variable.

This solution guarantees that the `m_value` will be initialized safely (no concurrent access by other threads while it is being initialized), and that it will be initialized only once. It is simpler than our first implementation — it clearly communicates what is being done, and it will be more efficient because once the value is calculated, no further locking is needed.

6.2 Laziness as an optimization technique

Now that we have seen the most basic variant of laziness — having a single value calculated the first time it is needed, and then caching it for later use, we can move on to a bit more advanced examples.

We will often encounter problems that we can not simply solve by changing a type of a single variable to its lazy equivalent like above. It will sometimes be necessary to develop alternative lazy versions of common algorithms that we are used to. Any time we have an algorithm that processes a whole data structure, while we just need a few resulting items, we have the potential to optimize the code by being lazy.

6.2.1 Sorting collections lazily

Say we have a collection of a few hundred employees stored in a vector. And we have a window that can show ten employees at a time, where the user has the option to sort the employees based on various criteria like their names, their age, number of years they have been working for the company, etc. When the user chooses to sort the list by how old the employees are, the program should show the ten oldest employees, and allow the user to scroll down to see the rest of the sorted list.

We can do this easily by sorting the whole collection, and displaying ten employees at a time. While this might be a good approach if we are expecting the user to be interested in the whole sorted list, but it would be an overkill if that was not the case. The user might be interested only in the *top 10* lists, and every time the criteria for sorting changes, we are sorting the whole collection.

If we wanted to make this as efficient as possible, we would need to think of a lazy way to sort the collection. We can base the algorithm on quicksort since it is the most commonly used in-memory sorting algorithm.

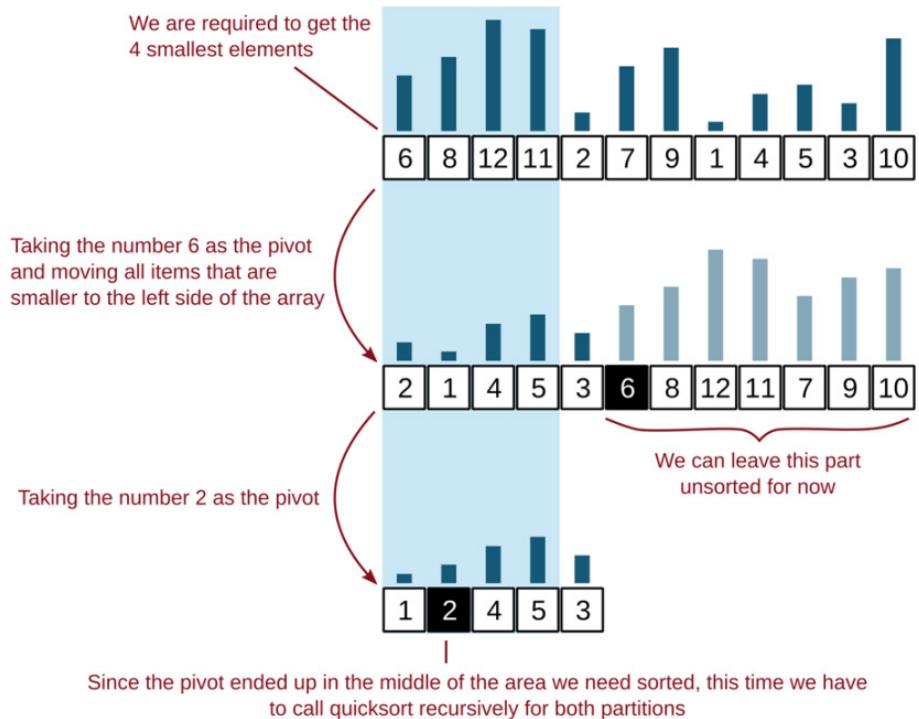
What the basic variant of quicksort does is simple — it takes an element from the collection, moves all elements that are greater than that element to the beginning of the collection, and the rest towards the end of collection (we could even use `std::partition` for this step). This is then repeated for both newly created partitions.

What should we change in order for the algorithm to be lazy — to sort only the part of the collection that needs to be shown to the user, while leaving the rest unsorted? We can not avoid doing the partitioning step, but what we can do is delay the recursive calls on the parts of the collection that we do not need sorted yet (Figure 6.1).

We will only sort them when (if) the need for it arises. This way, we have avoided all the recursive invocations of the algorithm on the section of the array that does

not need to be sorted. You can find a simple implementation of this algorithm in the `lazy-sorting` example that accompanies this book.

Figure 6.1. When we need to sort just a part of the collection, we can optimize the quicksort algorithm not to recursively call itself for the partitions that do not need to be sorted



The complexity of the lazy quicksort algorithm

Since the C++ standard specifies the required complexity for all algorithms it defines, someone might be interested in the complexity of the lazy quicksort algorithm that we have just defined.

Let the size of the collection be n , and we want to get the top k items.

For the first partitioning step, we will need $O(n)$ operations, for the second $O(n/2)$ and so on, until we reach the partition size that we actually need to fully sort. So, in total, for partitioning, we have $O(n)$.

For fully sorting the partition of size k , we will need $O(k \log k)$ operations since we need to perform the regular quicksort.

Therefore, the total complexity will be $O(n + k \log k)$ which is pretty neat because it means that if we are just searching for the maximum element in the collection, we will be in the same ballpark as `std::max_element` algorithm – $O(n)$, and if we are sorting the whole collection, it will be $O(n \log n)$ like the normal quicksort algorithm.

6.2.2 Item views in the user interfaces

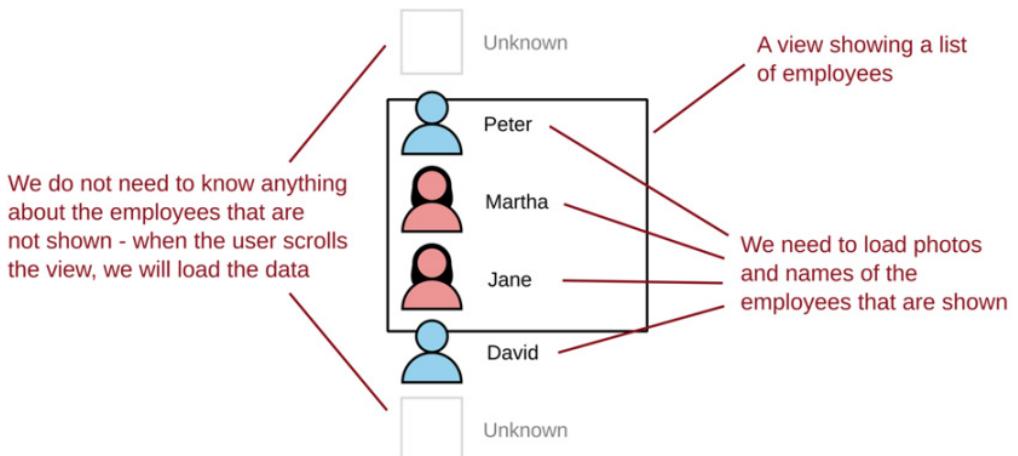
While the focus of the previous example was to show how to modify a particular algorithm to be more lazy, the background story of why we needed laziness in the first place is quite common.

Every time we have a large amount of data, while having only a limited amount of screen space to show that data to the user, we have the opportunity to optimize by being lazy. A common situation is that our data is stored in a database somewhere, and we need to show it to the user in one way or another.

To reuse the idea from the previous example, imagine we have a database containing the data about our employees. When showing the employees, we want to display their names along with their photos. In the previous example, the reason why we needed laziness is that sorting the whole collection while we need only to show ten items at a time is superfluous.

Now we have a database, and it is going to do the sorting for us. Does this mean that we should load everything at once? Of course not. We are not doing the sorting, but the database is — just like our lazy sort, databases tend to sort the data when it is requested. If we get all employees at once, it will have to sort all of them. But sorting is not the only problem — on our side, we need to display the pictures of each employee — loading a picture also takes time, and it also takes memory. If we were to load everything at once, we would slow down our program to a crawl, and take too much memory.

Figure 6.2. We do not need to get the data that is not shown to the user. We can just fetch the required information when it is needed.



Instead, the common approach is to load the data lazily — to load it only when we need to show it to the user. This way we use less processor time and less memory.

The only problem here is that we will not be able to be completely lazy — we will not be able to save all the previously loaded employee photos because it would, again, require too much memory. We would need to start forgetting previously loaded data, and reload it once it is needed again.

6.2.3 Recursion tree pruning by caching the function results

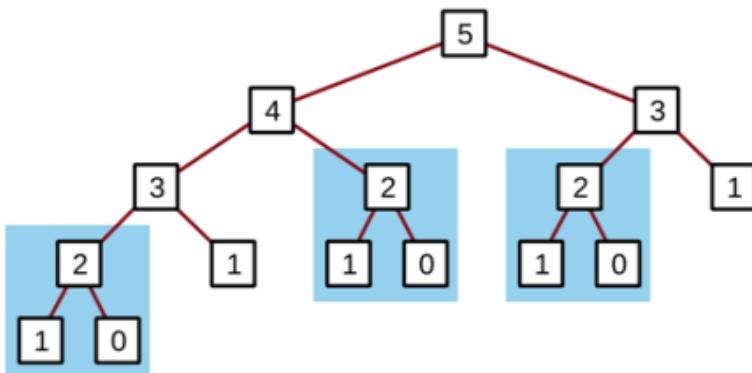
The good thing about C++ not directly supporting laziness is that we can implement it as we wish — and we are in control of how lazy we want to be on a case per case basis.

Let's use the common example — calculating Fibonacci numbers. Following the definition, we could implement it like this:

```
unsigned int fib(unsigned int n)
{
    return n == 0 ? 0 :
               n == 1 ? 1 :
                           fib(n - 1) + fib(n - 2);
}
```

This implementation is really inefficient — we have two recursive calls in the common case, and each of them will perform duplicate work because both `fib(n)` and `fib(n - 1)` will need to calculate `fib(n - 2)`. Both `fib(n - 1)` and `fib(n - 2)` will need to calculate `fib(n - 3)` and so on. The number of recursive calls will grow exponentially with the number n .

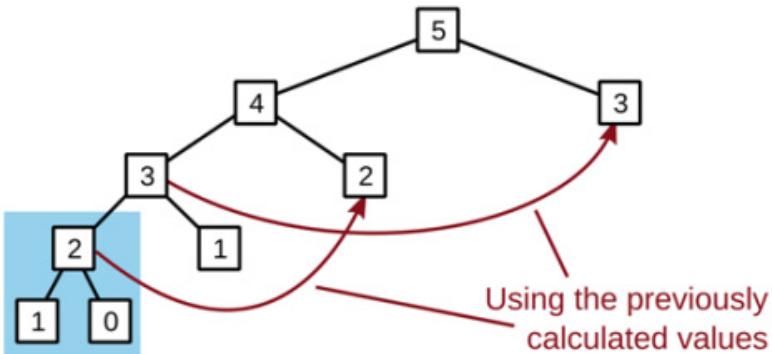
Figure 6.3. With `fib` implemented recursively, we will calculate the same value more than once. The recursion tree that calculates `fib(5)` contains three identical sub-trees calculating `fib(2)`, and two identical sub-trees calculating `fib(3)`. The multiplication of work grows exponentially with n .



This function is pure, so it will always return the same result when given a specific value as the input. This means that once we calculate `fib(n)`, we can store it somewhere, and reuse that value any time someone needs `fib(n)`. If we cached all

the previously calculated results, we would be able to remove all the duplicate sub-trees. This way, our evaluation tree would look like in the Figure 6.4 (the exact order of evaluation might not be like in the figure because C++ does not guarantee that `fib(n - 1)` will be called before `fib(n - 2)`, but all possible trees will essentially be the same — there will be no repeated sub-trees).

Figure 6.4. With `fib` implemented lazily, we will be able to avoid evaluating the same trees over and over. If one branch calculates `fib(2)`, other ones will just use the previously calculated value. Now, the number of nodes this tree has grows linearly with n .



Listing 6.4. Fibonacci implementation with caching

```
std::vector<unsigned int> cache{0, 1};  
①  
  
unsigned int fib(unsigned int n)  
{  
    if (cache.size() > n) {  
        return cache[n];  
    } else {  
        const auto result = fib(n - 1) + fib(n - 2);  
        cache.push_back(result);  
        ③  
        return result;  
    }  
}
```

- ① we can use a vector as the cache since we know that in order to find out `fib(n)`, we would need to know `fib(k)` for all $k < n$
- ② if the value is already in the cache, just return it
- ③ We are getting the result and are adding it to the cache. We can use `push_back` to add the n -th element because we know that all the previous ones are filled

The good thing about this approach is that we don't need to invent new algorithms for calculating the Fibonacci numbers — we don't even need to understand how the algorithm works. We just need to recognize that it calculates the same thing multiple times, and that the result is always the same because the `fib` function is

pure.

The only downside is that we are now taking much more memory to store all the cached values. If we wanted to optimize this, we would need to investigate a bit into how our algorithm works, and when the cached values are being used.

If we analyzed the code, we would see that we never use any values in the cache except the two last inserted ones. This means that we can replace the whole vector with a cache that stores just two values. In order to match the used `std::vector` API, we are going to create a class that looks like a vector, but it remembers only the last two inserted values.

Listing 6.5. Efficient cache for calculating the Fibonacci numbers (example:fibonacci/main.cpp)

```
class fib_cache {
public:
    fib_cache()
        : m_previous{0}
        , m_last{1}
        , m_size{2}
    {
    }

    size_t size() const
    {
        return m_size;
    }

    unsigned int operator[](unsigned int n) const
    {
        return n == m_size - 1 ? m_last :
            n == m_size - 2 ? m_previous :
                0;
    }

    void push_back(unsigned int value)
    {
        m_size++;
        m_previous = m_last;
        m_last = value;
    }
};
```

- ➊ The start of Fibonacci series — {0, 1}
- ➋ The size of the cache (not excluding the forgotten values)
- ➌ If we still have the requested numbers in the cache, we are returning them. Otherwise, we return 0. Alternatively, we could throw an exception.
- ➍ Adding a new value to the cache and increasing the size

6.2.4 Dynamic programming as a form of laziness

Dynamic programming is a technique of solving complex problems by splitting them into many smaller ones. When we solve the smaller ones, we store their solutions so that we could reuse them at a later stage. It is used in many real-world algorithms like shortest path finding, string distance calculation etc.

In a sense, the optimization we did for the function that calculates the nth Fibonacci number is also based on dynamic programming. We had a problem `fib(n)`, and we split it into two smaller problems — `fib(n - 1)` and `fib(n - 2)` (it was helpful that this was actually the definition of Fibonacci numbers). By storing the results of all the *smaller* problems we managed to optimize the initial algorithm significantly.

Now, while this was quite an obvious optimization for the `fib` function, it is not always the case.

Let's consider the problem of calculating the similarity between two strings. One of the possible measures for this is the Levenshtein distance⁹ (or *edit distance*) which is the minimal number of deletions, insertions and substitutions needed to transform one string to another. For example:

- "example" and "example" — the distance is zero because these are the same strings;
- "example" and "exam" — the distance is three because we need to delete three characters from the end of the first string;
- "exam" and "eram" — the distance is one because we just need to replace "x" with "r".

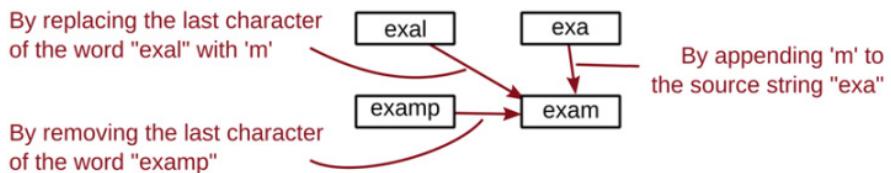
This problem can easily be solved. if we have already calculated the distance between two strings `a` and `b`, we can easily calculate the distances:

- between `a` and `b` with one character appended (representing the operation of adding a character to the source string),
- between `a` with one character appended and `b` (representing the operation of removing a character from the source string),
- and between `a` and `b` with both of them having a character appended to them (if it is the same character, the distance is the same, otherwise we have the operation of replacing one character with another).

This means that we have many different solutions of how to transform the source string `a` into the destination string `b`. We want to take the solution that will get us the minimum number of operations.

⁹ For more information on Levenshtein distance, check out en.wikipedia.org/wiki/Levenshtein_distance

Figure 6.5. Each invocation of $\text{lev}(m, n)$ calculates the minimum distance for all paths whose last operation is addition, for all paths whose last operation is removal, and all paths whose last operation was replacing a character



We will denote the distance between the first m characters of the string a and the first n characters of the string b as $\text{lev}(m, n)$. The distance function can be implemented recursively like so:

Listing 6.6. Recursively calculating the Levenshtein distance

```
unsigned int lev(unsigned int m, unsigned int n) {  
    return m == 0 ? n : n == 0 ? m : std::min({  
        lev(m - 1, n) + 1, ①  
        lev(m, n - 1) + 1, ②  
        lev(m - 1, n - 1) + (a[m - 1] != b[n - 1]) ③  
    }); ④  
}
```

- ① If any of the strings are empty, the distance is the length of the other one
- ② We are counting the operation of adding a character
- ③ We are counting the operation of removing a character
- ④ We are counting the operation of replacing a character, but only if we haven't replaced a character with the same one
- ⑤ We will skip passing the strings a and b as parameters for clarity

Implemented like this, our function lev will search the space of all possible transformations that convert the string a to b (at least paths that don't generate unnecessary operations like adding and consequently removing the same characters). Just like in the case of $\text{fib}(n)$, the number of function invocations will grow exponentially with m and n .

While we all know how to implement $\text{fib}(n)$ with a single loop, and nobody would write the implementation we started with, this time it is not obvious from the definition of the problem itself how it could be optimized.

But what is obvious is that just like like $\text{fib}(n)$, $\text{lev}(m, n)$ is a pure function — the result always depends just on its arguments and it has no side-effects. This means that we can not have more than $m * n$ different results. And the only way that our

implementation can have exponential complexity if it calculates the same thing multiple times.

So, what is the immediate solution that comes to mind? Cache all previously calculated results. Since we have two unsigned integers as arguments for the function, it is natural to use a matrix as the cache.

6.3 Generalized memoization

While it is usually better to write custom caches for each problem separately so that we can control how long a specific value stays in the cache (like in the case of our *forgetful* cache for the memoized version of `fib(n)`), and what is the best possible structure to keep the cache in (like using a matrix for `lev(m, n)`), it is sometimes useful just to be able to put a function into a wrapper, and to automatically get the *memoized* version of the function out.

The general structure for caching, when we can not predict with what arguments the function will be invoked with, is a map. Any pure function is a mapping from its arguments to its value, so the cache will be able to cover any pure function without problems.

Listing 6.7. Function that creates a memoized wrapper from a function pointer

```
template <typename Result, typename... Args>
auto make_memoized(Result (*f)(Args...))
{
    std::map<std::tuple<Args...>, Result> cache; ❶

    return [f, cache](Args... args) mutable -> Result
    {
        const auto args_tuple = ❷
            std::make_tuple(args...); ❷
        const auto cached = cache.find(args_tuple); ❷

        if (cached == cache.end()) { ❸
            auto result = f(args...); ❸
            cache[args_tuple] = result; ❸
            return result; ❸
        } else { ❹
            return cached->second; ❹
        }
    };
}
```

- ❶ We are creating a cache that maps tuples of arguments to the calculated results. If we want to use this in a multi-threaded environment, we would need to synchronize the changes to it with a mutex like we did in the first example
- ❷ Our lambda gets the arguments and checks whether we already have the result for them cached
- ❸ If we have a cache miss, call the function and store the result to the cache

- ④ If we found the result in the cache, we are returning it to the caller

Now that we have a way to turn any function into its memoized counter-part, let's try doing so with the Fibonacci number generation function.

Listing 6.8. Demonstration of how the `make_memoized` function is used

```
auto fibmemo = make_memoized(fib);

std::cout << "fib(15) = " << fibmemo(15)
    << std::endl;
std::cout << "fib(15) = " << fibmemo(15)
    << std::endl;
```

1
1
2
2

- ① We are calculating `fibmemo(15)` and it will be cached
 ② The next time we call `fibmemo(15)`, it will not calculate it again, it will just be loaded from the cache

If we try to benchmark this program, we will see that we have optimized the second call to `fibmemo(15)`. But the first call is slower, and it gets exponentially slower when the argument to `fibmemo` is increased. The problem here is that we still do not have the fully memoized version of the `fib` function. The `fibmemo` is caching the result, but the `fib` function is not using that cache. It calls itself directly. The `make_memoized` function that we have created works well for ordinary functions, but it does not optimize the recursive ones.

If we want to memoize the recursive calls as well, we will have to modify the `fib` function not to call itself directly, but it needs to allow us to pass it another function to call instead like this:

```
template <typename F>
unsigned int fib(F& fibmemo, unsigned int n)
{
    return n == 0 ? 0
        : n == 1 ? 1
        : fibmemo(n - 1) + fibmemo(n - 2);
}
```

This way, when we invoke the `fib` function, we will be able to pass it our memoized version for its recursive calls. Unfortunately, this time our memoization function will have to be more complex because we need to inject our memoized version into the recursion. Namely, if we just tried to memoize a recursive function, we would be caching just its last result, and not the results of the recursive invocations because the recursive calls would call the function itself, instead of our memoized wrapper.

We will need to create a function object that can store the function we need to memoize, and the cached results of previous invocations.

Listing 6.9. Memoization wrapper for recursive functions (example:recursive-memoization/main.cpp)

```

class null_param {};

template <class Sig, class F>
class memoize_helper;

template <class Result, class... Args, class F>
class memoize_helper<Result(Args...), F> {
private:
    using function_type = F;
    using args_tuple_type
        = std::tuple<std::decay_t<Args>...>;
    function_type f;
    mutable std::map<args_tuple_type, Result> m_cache;      1
    mutable std::recursive_mutex m_cache_mutex;                1

public:
    template <typename Function>
    memoize_helper(Function&& f, null_param)          2
        : f(f)
    {                                              2

        memoize_helper(const memoize_helper& other)      2
            : f(other.f)
    {                                              2

template <class... InnerArgs>
Result operator()(InnerArgs&&... args) const
{
    std::unique_lock<std::recursive_mutex>
        lock{m_cache_mutex};

    const auto args_tuple =
        std::make_tuple(args...);
    const auto cached = m_cache.find(args_tuple);          3
                                                    3
                                                    3

    if (cached != m_cache.end()) {                         4
        return cached->second;

    } else {                                              5
        auto&& result = f(
            *this,
            std::forward<InnerArgs>(args)...);           5
            5
            5
        m_cache[args_tuple] = result;                     5
        return result;
    }
}

```

```

};

template <class Sig, class F>
memoize_helper<Sig, std::decay_t<F>>
make_memoized_r(F&& f)
{
    return {std::forward<F>(f), detail::null_param()};
}

```

- ➊ We are defining the cache, and since it is mutable, we need to synchronize all the changes
- ➋ The constructors just need to initialize the wrapped function. If we wanted, we could have made the copy-constructor copy the cached values as well, but that is not necessary
- ➌ We are searching for the cached value
- ➍ If it is found, we can return it without calling `f`
- ➎ If we do not have the cached value, we are calling the function `f` and storing the result. Note that we are passing `*this` as the first argument — this is the function that will be used for the recursive call
- ➏ A dummy class we will use in the constructor to avoid overload collision with the copy-constructor

Now, when we want to create the memoized version of `fib`, we can simply write:

```

auto fibmemo = make_memoized_r<
    unsigned int(unsigned int)>(
    [](auto& fib, unsigned int n) {
        std::cout << "Calculating " << n << "!\n";
        return n == 0 ? 0
            : n == 1 ? 1
            : fib(n - 1) + fib(n - 2);
    });

```

All invocations will now be memoized. The lambda will indirectly call itself through a reference to `memoize_helper` class which will be passed to it as its first argument `fib`.

One benefit with this implementation, compared to `make_memoized` is that this one accepts any type of function object, while we had to pass a function pointer to the former.

6.4 Expression templates and lazy string concatenation

The previous examples focussed mostly on the runtime optimizations — the situations when our program might take different code paths, and we want to be able to optimize for all of them. Lazy evaluation can be beneficial even if we know in advance every step that our program is doing.

Consider one of the most common operations in our programs — string concatenation.

```
std::string fullname = title + " " + surname + ", " + name;
```

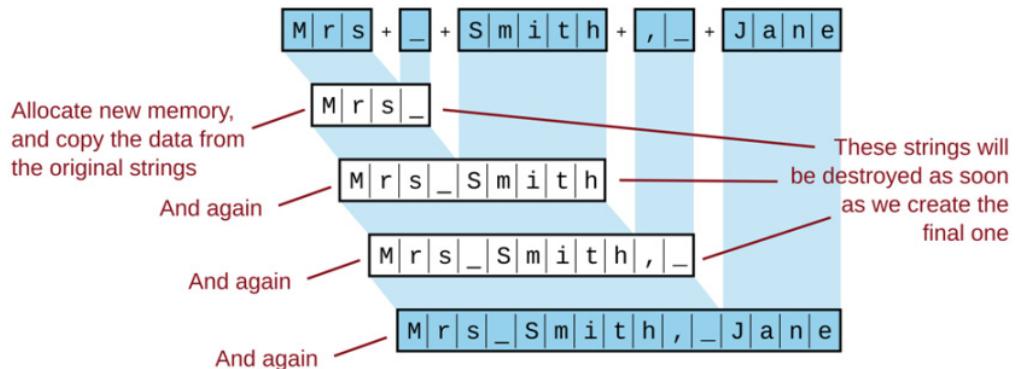
We have a few strings that we want to concatenate. This implementation is

perfectly valid, and does just what we want. But it is not as fast as it can be. Let's look at this from the compiler's perspective — the `operator+` is a left-associative binary operator, so the above expression will be equivalent to:

```
std::string fullname = (((title + " ") + surname) + ", ") + name;
```

When evaluating the `title + " "` sub-expression, a new temporary string will be created. For each of the following concatenations, the `append` function will be called on that temporary string. For each of the appends, the string needs to grow to be able to fit the new data. This means that sometimes the currently allocated buffer inside of this temporary string will not be able to hold all the data it needs to, and that a new buffer will need to be allocated and the data from the old one copied into it.

Figure 6.6. When we concatenate strings eagerly, we might need to allocate new memory because the previous buffer is not large enough to store the concatenated string



This is inefficient. We are generating (and destroying) buffers that we do not actually need. When we have a few strings to concatenate, it would be much more efficient just to delay calculating the result until we know all the strings that we need to concatenate. Then, we could create a buffer that is large enough to store the final result, and copy the data from the source strings just once.

This is where expression templates come to help, they allow us to generate expression definitions instead of calculating the value of some expression. Instead of implementing the `operator+` to return a concatenated string, we can make it return a definition of the expression it represents so that we could evaluate it at a later stage. In our case, the source of the problem is that the `operator+` is a binary one, whereas we need to concatenate more strings. So, let's create a structure that will represent an expression that concatenates multiple strings. Since we have to store an arbitrary number of strings, we will create a recursive structure template — each node will keep a single string (the `data` member), and a node that keeps the rest (the `tail` member).

Listing 6.10. Structure that holds an arbitrary number of strings (example:string-concatenation/main.cpp)

```

template <typename... Strings>
class lazy_string_concat_helper;

template <typename LastString, typename... Strings>
class lazy_string_concat_helper<LastString,
                                Strings...> {
private:
    LastString data;
    lazy_string_concat_helper<Strings...> tail;           1
                                                                2

public:
    lazy_string_concat_helper(
        LastString data,
        lazy_string_concat_helper<Strings...> tail)
    : data(data)
    , tail(tail)
{           3
}

int size() const
{
    return data.size() + tail.size();           3
                                                3
}           3
            3

template <typename It>
void save(It end) const
{
    const auto begin = end - data.size();
    std::copy(data.cbegin(), data.cend(),
              begin);           4
    tail.save(begin);           4
}           4
            4

operator std::string() const
{
    std::string result(size(), '\0');           5
    save(result.end());           5
    return result;           5
}

lazy_string_concat_helper<std::string,
                         LastString,
                         Strings...>
operator+(const std::string& other) const           6
{
    return lazy_string_concat_helper<std::string,           6
                                    LastString,           6
                                    Strings...>(           6
                                    other,           6
                                    *this);           6
}

```

```

        );
    }
};
```

- ① Storing the copy of the original string
- ② Storing the structure that contains other strings
- ③ We are calculating the size of all strings combined
- ④ Our structure stores strings in reversed order — the data member variable contains the string that comes last, so it needs to go to the end of the buffer
- ⑤ When we want to convert the expression definition into a real string, we need to allocate enough memory, and start copying the strings into it
- ⑥ The operator+ just creates a new instance of the structure with one string added to it

Since this is a recursive structure, we need to create the base case, so that we do not end up with an infinite recursion:

```

template <>
class lazy_string_concat_helper<> {
public:
    lazy_string_concat_helper()
    {
    }

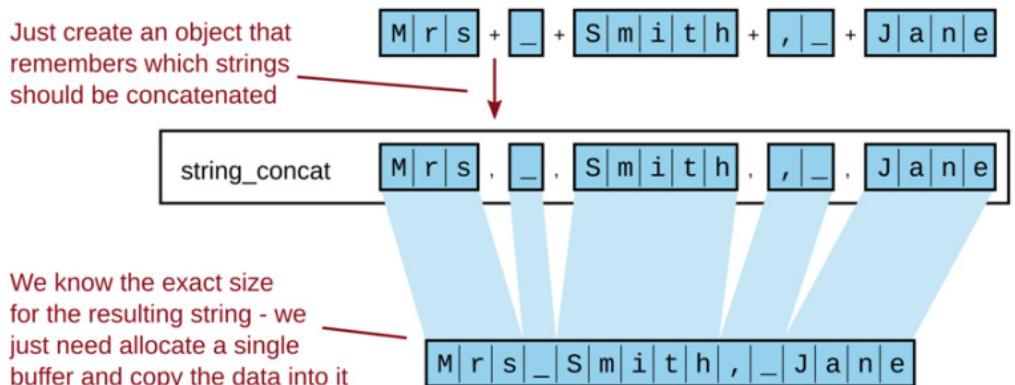
    int size() const
    {
        return 0;
    }

    template <typename It>
    void save(It) const
    {
    }

    lazy_string_concat_helper<std::string>
    operator+(const std::string& other) const
    {
        return lazy_string_concat_helper<std::string>(
            other,
            *this
        );
    }
};
```

This structure can hold as many strings as we want. It performs no concatenations until we ask it for the result by casting it to `std::string`. When that happens, it will create a new string, and copy the data from the ones stored inside into the resulting one (Figure 6.7).

Figure 6.7. Instead of having the operator+ generate a string, it just generates an object that remembers which strings should be concatenated. When we need to create the resulting string, we will know the exact size of the buffer that we need to generate.



Listing 6.11. Usage of our structure for efficient string concatenation

```
lazy_string_concat_helper<> lazy_concat;           ①

int main(int argc, char* argv[])
{
    std::string name = "Jane";
    std::string surname = "Smith";

    const std::string fullname =           ①
        lazy_concat + surname + ", " + name;  ①

    std::cout << fullname << std::endl;
}
```

- ① We can not overload the operator+ on `std::string` so we are using a small trick to force the usage of our concatenation structure by *appending* to an instance of it

This behaves similarly to the `lazy_val` structure we had in the beginning of this chapter. It is just that this time we are not defining the computation that should be lazily executed through a lambda, but we are generating the computation ourselves from the expression that the user wrote.

6.4.1 Purity and expression templates

One thing you might have noticed is that we are storing copies of original strings in the `lazy_string_concat_helper` class. It would be much more efficient if we just used references to them instead. We started all this with the idea to optimize string concatenation as much as possible. So we might optimize this as well.

```
template <typename LastString, typename... Strings>
```

```

class lazy_string_concat_helper<LastString,
                           Strings...> {
private:
    const LastString& data;
    lazy_string_concat_helper<Strings...> tail;

public:
    lazy_string_concat_helper(
        const LastString& data,
        lazy_string_concat_helper<Strings...> tail)
        : data(data)
        , tail(tail)
    {
    }

    // ...
};

```

There are two potential pitfalls here. The first one is that we can not use this expression outside of the scope in which the strings were defined. As soon as we exit the scope, the strings will be destroyed and the references we keep in the `lazy_string_concat_helper` object will become invalid.

Another potential problem is that the user will expect that the result of string concatenation is a string, and not some other intermediary type. This might lead to unexpected behaviour if our optimized concatenation is used with the automatic type deduction.

Consider the following example:

Listing 6.12. Problem with string references and lazy concatenation

```

std::string name = "Jane";
std::string surname = "Smith";

const auto fullname =
    lazy_concat + surname + ", " + name;      ①

name = "John";                                ②

std::cout << fullname << std::endl;

```

- ① We think that we have created a string "Smith, Jane"
- ② But the output is "Smith, John"

The problem here is that we are now storing references to the strings that we need to concatenate. If the strings change between the point where we created the expression (the declaration of the `fullname` variable) and the point where we are required to compute the result (writing out the `fullname` to the standard output) we will get an unexpected result. The changes to the strings made *after* we think we

concatenated them will be reflected on the result string.

This is an important thing to keep in mind, in order to work as expected, laziness requires purity. Pure functions give us the same result whenever we call them with the same arguments. And that is why we can delay executing them without any consequences. As soon as we allow side-effects like changing a value of a variable to influence the result of our operation, we will get undesired results when trying to make it execute lazily.

Expression templates allow us to generate structures that represent a computation instead of immediately calculating that expression. This way, we can choose when the computation will take place, we can change the normal order in which C++ evaluates expressions (see the `right-associative-addition` example),¹⁰ and transform the expressions in any way we want.

6.5 Summary

- Executing the code lazily and caching results can have significant impact on the speed of our programs
- It is not often easy (sometimes not even possible) to construct lazy variants of algorithms that we use in our programs, but if we manage to do it, we can make our programs much more responsive
- There is a big class of algorithms that have exponential complexity that can be optimized to be linear or quadratic just by caching intermediary results
- Levenshtein distance has many applications in sound processing, DNA analysis, spell checking, but can also find its place in regular programs — when we need to notify the UI of changes in a data model, it is useful to be able to minimize the number of operations that the UI needs to perform
- While we should write the caching mechanisms for each problem separately, it is sometimes useful to have functions like `make_memoized` to benchmark the speed gain that we could achieve by caching function results
- Expression templates are a powerful mechanism for delaying computation. They are often used in libraries that operate on matrices, and other places where we need to optimize the expressions before handing them out to the compiler

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch06

¹⁰ If you need to build and transform more complex expression trees, you can check out the Boost.Proto library www.boost.org/doc/libs/1_62_0/doc/html/proto.html



7 Ranges

This chapter covers:

- The problems of passing iterator pairs to algorithms
- What ranges are and how to use them
- Creating chained range transformations using the pipe syntax
- What range views and actions are
- Writing succulent code without for-loops

We have seen why we should avoid writing raw for-loops in chapter 2, and that we should instead rely on using generic algorithms provided to us by the STL.

Now, while this has some significant benefits, we have also seen its downsides. Namely, the algorithms in the standard library were not designed to be easily composed with each other. Instead, they are mostly focussed on providing a way to allow implementation of a more advanced version of an algorithm by applying one algorithm multiple times.

A perfect example of this is `std::partition` which moves all items in a collection that satisfy a predicate to the beginning of the collection, and it returns an iterator to the first element in the resulting collection that does not satisfy the predicate. This allows us to create a function that does multi-group partitioning — not to be limited only to predicates that return true or false — just by invoking `std::partition` multiple times.

As an example, we are going to implement a function that groups people in a

collection based on the team they belong to. It receives the collection of persons, a function that gets the team name for a person, and a list of teams. We can perform `std::partition` multiple times —once for each team name, and we will get a list of people grouped by the team they belong to.

Listing 7.1. Grouping people by the team they belong to

```
template <typename Persons, typename F>
void group_by_team(Persons& persons,
                  F team_for_person,
                  std::vector<std::string> teams)
{
    auto begin = std::begin(persons);
    const auto end = std::end(persons);

    for (const auto& team : teams) {
        begin = std::partition(begin, end,
                               [&](const auto& person) {
                                   return team == team_for_person(person);
                               });
    }
}
```

While this way to compose algorithms is useful, a more common use-case is to have a resulting collection of one operation passed to another. To recall our example from the chapter 2, we had a collection of people, and we wanted to extract the names of female employees only. Writing a for loop that does this is trivial.

```
std::vector<std::string> names;

for (const auto& person : people) {
    if (is_female(person)) {
        names.push_back(name(person));
    }
}
```

If we wanted to solve the same problem using the STL algorithms, we would need to create an intermediary collection to copy the persons that satisfy the predicate (`is_female`) and then to use `std::transform` to extract the names for all persons in that collection. This would be sub-optimal both performance and memory-wise.

The main issue behind this problem is that STL algorithms take iterators to the beginning and the end of a collection as separate arguments instead of taking the collection itself. There are a few implications of this:

- The algorithms can not return a collection as a result;
- Even if we had a function that returns a collection, we would not be able to pass it directly to the algorithm, but we would need to create a temporary variable so that

- we could call `begin` and `end` on it;
- Because of the previous reasons, most algorithms mutate their arguments instead of leaving them immutable and just returning the modified collection as the result.

This all makes it hard to implement our program logic without having at least local mutable variables.

7.1 *Introduction to ranges*

There have been a few attempts to fix these problems, but the concept that proved to be most versatile were ranges. For the time being, let's just think of ranges as a simple structure that contains two iterators — one pointing to the first element in a collection, and one pointing to the element after the last one.

Note **Libraries implementing ranges**

Ranges have not became a part of the standard library yet, but there is an ongoing effort for their inclusion into the standard, currently planned for C++20. The proposal to add ranges into C++ standard is based on the range-v3 library by Eric Niebler, which we will be using for the code examples in this chapter.

An older, but more battle-tested library is Boost.Range. It is not as featureful as range-v3, but it still is useful, and it supports older compilers. The syntax is mostly the same, and the concepts that we will cover apply to it as well.

What are the benefits of keeping two iterators in the same structure instead of having them as two separate values?

The main benefit is that we will be able to return a complete range as a result of a function, and to pass it directly to another function without creating local variables to hold the intermediary results.

Passing pairs of iterators is also error prone. It is possible to pass iterators belonging to two separate collections to an algorithm that operates on a single collection. Or to pass the iterators in a wrong order — to have the first iterator point to an element that comes after the second iterator. For both of these the algorithm would try to iterate through all elements from the starting iterator until it reaches the end iterator which would produce undefined behaviour.

Our previous example becomes a simple composition of `filter` and `transform` functions:

```
std::vector<std::string> names =
    transform(
        filter(people, is_female),
        name
    );
```

The `filter` function will return a range containing elements from the `people` collection that satisfy the `is_female` predicate. The `transform` function

will then take this result, and return the range of names of everybody in the filtered range.

We can nest as many range transformations like `filter` and `transform` as we want. The problem is that the syntax becomes quite cumbersome to reason about when we have more than a few composed transformations.

For this, the range libraries usually provide a special `pipe` syntax that overloads the `|` operator, inspired by the UNIX shell pipe operator. So, instead of nesting the function calls, we can simply pipe the original collection through a series of transformations like this:

```
std::vector<std::string> names = people | filter(is_female)
| transform(name);
```

Just like in the previous example, we are filtering the collection of persons on the `is_female` predicate, and then extracting just the names from the result. The main difference here is that, once you get accustomed to seeing the operator `|` as "*pass through a transformation*" instead of "*bitwise or*", this becomes easier to write and reason about than the original example.

7.2 ***Creating read-only views over data***

Now, the first question that comes to mind when seeing code like this is how efficient is it compared to writing a for loop that does the same. We have seen in chapter 2 that implementing this using the STL algorithms incurs performance penalties because we need to create a new vector of persons which will hold the copies of all females from the `people` collection in order to be able to call `std::transform` on it.

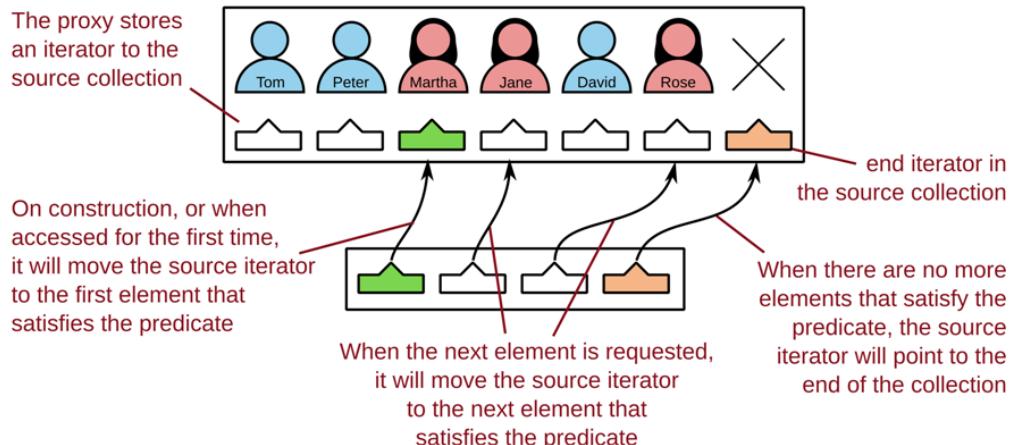
From just reading the solution which uses ranges, we might get the impression that nothing has changed but the syntax.

7.2.1 ***Filter function for ranges***

The `filter` transformation still needs to return a collection of people so that we can call `transform` on it. This is where the magic of ranges comes into play. Ranges are an abstraction that represents a collection of items, but nobody said that it actually is a collection — it just needs to behave like one. It needs to have a start, to know what is its end, and to allow us to get to each of its elements.

Instead of having `filter` return a collection like `std::vector`, it will just return a range structure whose begin iterator will be a smart proxy iterator that points to the first element in the source collection that satisfies the given predicate. And the end iterator will be a proxy for the original collection's end iterator. The only thing that the proxy iterator needs to do differently than the iterator from the original collection is that it needs to point only at the elements that satisfy the filtering predicate.

Figure 7.1. The view created by filter will store an iterator to the source collection. The iterator will only point to the elements that satisfy the filtering predicate. The user of this view will be able to use it as if it were a normal collection of people with only three elements in it—Martha, Jane and Rose.



In a nutshell, every time the proxy iterator is incremented, it needs to find the next element in the original collection that satisfies the predicate.

Listing 7.2. Increment operator for the filtering proxy iterator

```
auto& operator++()
{
    ++m_current_position;                                1
    m_current_position =
        std::find_if(m_current_position,      2
                     m_end,                  3
                     m_predicate);

    return *this;
}
```

- ➊ `m_current_position` is an iterator to the collection we are filtering. Whenever our proxy iterator is to be incremented, we need to find the first element after the current one that satisfies the predicate.
- ➋ We are starting our search from the next element.
- ➌ If there are no more elements that satisfy the predicate, we are returning an iterator pointing to the end of the source collection, which is also the end of the filtered range.

With a proxy iterator for filtering, we do not need to create a temporary collection containing copies of the values in the source collection that satisfy the predicate. We have just created a new *view* of already existing data.

We can just pass this view to the `transform` algorithm, and it will work just as well as it would on a real collection. Every time it requires a new value, it will request

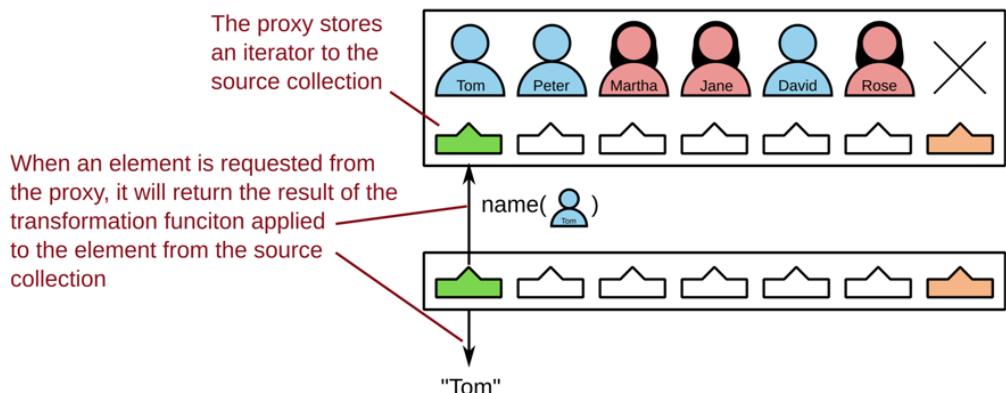
the proxy iterator to be moved one place to the right, and it will move to the next element that satisfies the predicate in the source collection. This means that `transform` will go through the original collection of people, it will just not be able to *see* any person who is not female.

7.2.2 Transform function for ranges

In a similar manner to `filter`, the `transform` function does not need to return a new collection. It also can return just a view over the existing data.

Unlike `filter` which returned a new view that contained the same items original collection had, just not all those items, `transform` needs to return the same number of elements found in the source collection, but it does not give the access to the elements directly. It returns each element from the source collection, but transformed.

Figure 7.2. The view created by `transform` will store an iterator to the source collection. The iterator will access all the elements in the source collection, but the view will not return them directly, it will first apply the transformation function to the element, and return its result.



This means that the increment operator does not need to be special, it just needs to simply increment the iterator to the source collection. This time, the operator that will be different is the operator to dereference the iterator. Instead of returning the value in the source collection, we are first applying the transformation function to it.

Listing 7.3. Dereference operator for the transformation proxy iterator

```
auto operator*() const
{
    return m_function(
        *m_current_position ①
    );
}
```

- 1 We are getting the value from the original collection, applying the transformation function to it, and returning it as the value our proxy iterator points to.

This way, just like with `filter`, we are avoiding creation of a new collection that holds the transformed elements — we are just creating a view that instead of showing original elements as they are, it shows them transformed.

Now, we can pass on the resulting range to another transformation, or we can assign it to a proper collection as in our example.

7.2.3 Lazy evaluation of range values

It is worth noting that even if we have two different range transformations in this example — one `filter` and one `transform` — the calculation of the resulting collection takes only a single pass through the source collection. Just like in the case of a hand-written for loop.

Range views are evaluated lazily, which means that, when we call `filter` or `transform` on a collection, it just defines a view, it does not evaluate a single element in that range.

Let's modify our example to fetch the names only of the first three females in the collection. We can use the `take(n)` range transformation which creates a new view over the source range that shows only the first `n` items in that range (or less if the source range has less than `n` elements).

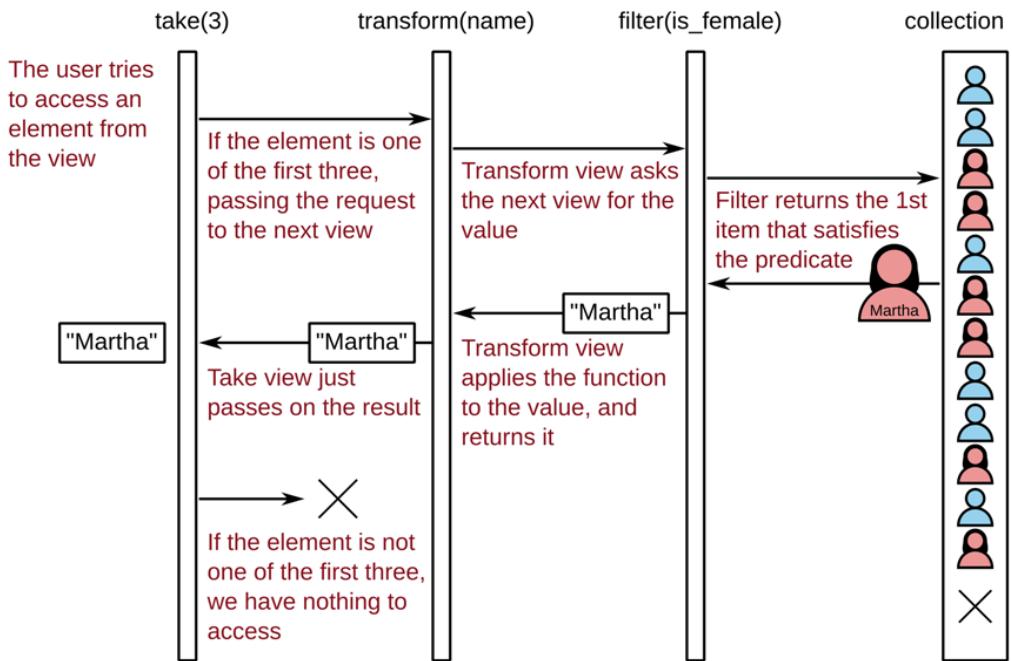
```
std::vector<std::string> names = people | filter(is_female)
                                         | transform(name)
                                         | take(3);
```

Let's analyze this snippet part by part:

- When `people | filter(is_female)` gets evaluated, nothing will happen apart from a new view being created. We haven't accessed a single person from the `people` collection, except potentially to initialize the iterator to the source collection to point to the first item that satisfies the `is_female` predicate;
- Then we pass that view to `| transform(name)`. The only thing that will happen is that a new view gets created. We still haven't accessed a single person, nor called the `name` function on any of them;
- Then we apply `| take(3)` to that result. It will, again, just create a new view and nothing else;
- The last step is that we need to construct a vector of strings from the view that we got as the result of `| take(3)` transformation.

In order to create a vector, we actually need to know the values that we want to put in it. So, this part will actually need to go through the view and access each of its elements.

Figure 7.3. When accessing an element from the view, it proxies the request to the next view in the composite transformation, or to the collection. Depending on the type of the view, it might transform the result, or skip elements, traverse them in a different order, and much more.



When we try to construct the vector of names from the range, all the values in the range will have to be evaluated. For each element that will be added to the vector, the following will be done:

- call the dereference operator on the proxy iterator that belongs to the range view returned by `take`;
- it will pass the request on to the proxy iterator created by `transform`. This iterator will pass on the request;
- now we are trying to dereference the proxy iterator defined by `filter` transformation. It will go through the source collection, find and return the first person that satisfies the `is_female` predicate. This is the first time we accessed any of the persons in the collection, and the first time the `is_female` function is called;
- the person retrieved by dereferencing the `filter` proxy iterator is passed to the `name` function, and the result is returned to the `take` proxy iterator which passes it on to be inserted into the `names` vector.

When an element is inserted, we will go to the next one, and next one, until we reach the end. Now, since we have limited our view to just three elements, this means that we do not need to access a single person in the `people` collection after

we find the third female.

This is lazy evaluation at work. Even if the code is more generic, and shorter than the equivalent hand-written for loop, it does exactly the same and has no performance penalties.

7.3 **Mutating values through ranges**

While many useful transformations can be implemented as simple views, some require changing the original collection. We will call these transformations *actions* as opposed to *views*.

One common example for the action transformation is sorting. In order to be able to sort a collection, we need to access all of its elements and to reorder them. This means that we need to change the original collection, or that we need to create and keep a sorted copy of the whole collection. The later is especially needed when the original collection is not randomly-accessible (a linked list for example) and can not be sorted efficiently — we need to copy its elements into a new collection that is randomly-accessible and sort that one instead.

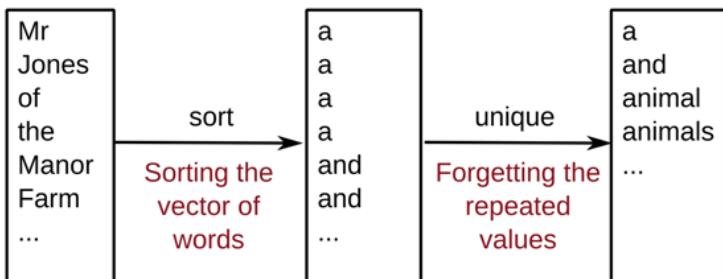
Views and actions in range-v3 library

As mentioned earlier, since the range-v3 library is used as the base for the proposal for extending the STL with ranges, we will be using it in the code examples, and we will use the same nomenclature it uses.

The range transformations that create views like `filter`, `transform` and `take` live in the `range::view` namespace, while the actions live in `range::action`. It is important to differentiate between these two, so we are going to specify the namespaces `view` and `action` from now on.

Imagine we have a function `read_text` that returns some text represented as a vector of words, and we want to collect all the different words in it. The easiest way to do this is to sort all the words, and then remove consecutive duplicates (we are going to consider all words to be lowercase in this example, for the sake of simplicity).

Figure 7.4. In order to get a list of words that appear in some text, it is sufficient to sort them, and then remove the consecutive repeated values



We can do it simply by piping it through `sort` and `unique` actions like this:

```
std::vector<std::string> words =
    read_text() | action::sort
        | action::unique;
```

Since we are passing a temporary to the `sort` action, it does not need to create a copy to work on, it can just reuse the vector returned by the `read_text` function, and do the sorting in-place. The same goes for `unique` — it can operate directly on the result of `sort` action. If we wanted to keep the intermediary result, we would use `view::unique` instead which does not operate on a real collection, but just creates a view that skips all repeated consecutive occurrences of a value.

This is an important distinction between views and actions. A view transformation creates a lazy view over the original data, while an action works on an existing collection, and performs its transformation eagerly.

Actions do not need to be performed only on temporaries, we can do it also on l-values using the operator `|=` like so:

```
std::vector<std::string> words = read_text();
words |= action::sort | action::unique;
```

This combination of views and actions gives us the power to choose when we want something to be done lazily and when to be done eagerly. A benefit of having this choice is that we can choose:

- to be lazy in the cases when we don't expect all items in the source collection to need processing, and when they do not need to be processed more than once;
- and to be eager to calculate all elements of the resulting collection if we know they will be accessed often.

7.4 Delimited and infinite ranges

We have started this chapter with the premise that a range is a structure that holds one iterator to the beginning and one to the end — exactly what the STL

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/functional-programming-in-cplusplus>

algorithms take, just in a single structure.

Now, the end iterator is a strange thing. We can never dereference it since it points to an element after the last element in the collection. We usually don't even move it. It is mainly used only to test whether we have reached the end of a collection:

```
auto i = std::begin(collection);
const auto end = std::end(collection);
for (; i != end; i++) {
    // ...
}
```

Therefore, it does not really need to be an iterator — it just needs to be something that we can use to test whether we are at the end. This special value is called a sentinel and allows us to have more freedom when implementing a test for whether we have reached the end of a range.

While this does not give us much when ordinary collections are in question, it allows us to create delimited and infinite ranges.

7.4.1 Delimited ranges optimize handling input ranges

Delimited ranges are ranges for which we don't know the end in advance, but for which we have a predicate function that can tell us when we have reached the end. Examples of this are null-terminated strings where we need to traverse the whole string until we reach the '\0' character, or the input streams where we read one token at a time until the stream becomes invalid — until we fail to extract a new token from it. In both these cases, we know the beginning of the range, but in order to know where the end is, we need to traverse the range item by item until the end test returns true.

Let's consider the input streams, and analyze the code which calculates the sum of the numbers it reads from the standard input:

```
std::accumulate(std::istream_iterator<double>(std::cin),
               std::istream_iterator<double>(),
               0);
```

We are creating two iterators in this snippet. One proper iterator — which represents the start of the collection of doubles read from `std::cin`, and one special iterator that does not belong to any input stream. This iterator is a special value that we will use to test whether we have reached the end of the collection — it is an iterator that behaves like a sentinel.

The `std::accumulate` algorithm will read values until its traversal iterator becomes equal to the end iterator. This means that we need to have the `operator==` and `operator!=` implemented for the `std::istream_iterator`. The equality operator need to work with both the proper iterators and special sentinel values. The implementation would have a form like this:

```

template <typename T>
bool operator==(const std::istream_iterator<T>& left,
                  const std::istream_iterator<T>& right)
{
    if (left.is_sentinel() && right.is_sentinel()) {
        return true;

    } else if (left.is_sentinel()) {
        // test whether sentinel predicate is
        // true for the **right** iterator

    } else if (right.is_sentinel()) {
        // test whether sentinel predicate is
        // true for the **left** iterator

    } else {
        // both iterators are normal iterators,
        // test whether they are pointing to the
        // same location in the collection
    }
}

```

We need to cover all the cases — whether the left iterator is a sentinel or not, and the same for the right one. And all these are checked on each step of an algorithm like `std::accumulate`.

This is inefficient. It would be much easier if the compiler was able to know that something is a sentinel at compile-time. This will be possible if we lift the requirement that the end of a collection has to be an iterator — if we allow it to be anything that can be equally-compared to a proper iterator. This way, the four cases we had in the previous code snippet will become separate functions and the compiler will know exactly which one to call based on the involved types. If it gets two iterators, it will call the `operator==` for two iterators, if it gets an iterator and a sentinel, it will call the `operator==` for an iterator and a sentinel, and so on.

Range-based for loop and sentinels

The range-based for loop, as defined in C++11 and C++14 requires both the begin and end to have the same type, that is, they need to be iterators, which means that the sentinel-based ranges can not be used with it.

This requirement was removed in C++17 where it is now allowed to have different types for the begin and end, which effectively means that the end can be a sentinel now.

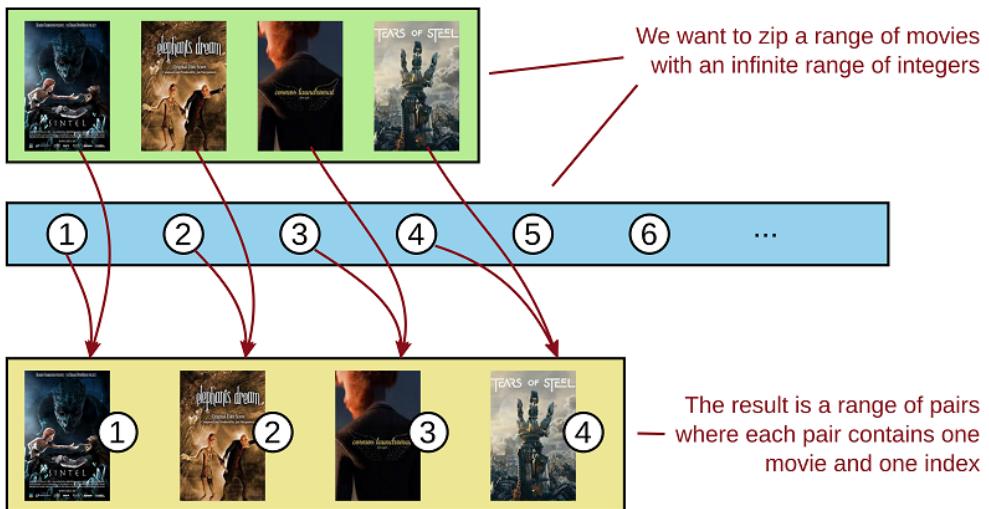
7.4.2 Sentinels also give us infinite ranges

The sentinel approach gave us some optimizations for the delimited ranges. But there is more, we are now able to easily create infinite ranges as well. Infinite ranges are the ranges that do not have an end like the range of all positive integers.

We have a start — the number 0, but there is no end.

While it is not obvious why we would need infinite data structures, they come in handy from time to time. One of the most common examples where we could use a range of integers is when we need to enumerate items in another range. Let's imagine we have a range of movies sorted by their scores, and we want to write out the first ten to the standard output along with their positions.

Figure 7.5. The range does not have a notion of an item index. If we want to have the indices for the elements in a range, we can zip it with the range of integers. We will get a range of pairs, where each pair contains an item from the original range along with its index



For this, we can use the `view::zip` function. It takes two ranges¹¹, and pairs up the items from those ranges. The first element in the resulting range will be a pair of items — the first item from the first range, and the first item from the second range. The second item will be a pair containing the second item from the first range, and the second item from the second range, and so on. The resulting range will end as soon as any of the source ranges ends.

Listing 7.4. Writing out the top 10 movies along with their positions (example:top-movies)

```
template <typename Range>
void write_top_10(const Range& xs)
{
    auto items =
```

¹¹ `view::zip` can also zip more than two ranges — the result will be a range of n-tuples instead of a range of pairs

```

view::zip(xs, view::ints(1))
    | view::transform([](const auto& pair) {
        return std::to_string(pair.second) + ①
            " " + pair.first;
    })
    | view::take(10); ②

for (const auto& item : items) {
    std::cout << item << std::endl;
}
} ③

```

- ① We are zipping the range of movies with the range of integers, starting with 1. This will give us a range of pairs where each pair will consist of a movie name and the index.
- ② The transform function takes a pair, and generates a string containing the rank of the movie and the movie name.
- ③ We are interested only in the first 10 movies

Instead of the infinite range of integers, we could have just used integers from 1 to `xs.length()` to enumerate the items in `xs`. But that would not work quite as well. As we have seen before, we could have a range for which we do not know its end, and therefore we probably can not tell its size without traversing it. This would mean we need to traverse it twice — once to get its size, and once for the `view::zip` and `view::transform` to do their magic. This is not only inefficient, but is also impossible to do with some of the range types. Ranges like the input stream range can not be traversed more than once. Once we read a value, we can not read it again.

Another benefit of having infinite ranges is not in actually using them, but in designing our code to be able to work on them. This makes our code more generic. Namely, if we write an algorithm that works on infinite ranges, it will work on a range of any size. It will also work on ranges that we don't know the size of.

7.5 Using ranges to calculate word frequencies

Let's move on to a more complicated example to see how our programs might become more elegant if we use ranges instead of writing the code in the "old style".

We will reimplement the example we talked about in chapter 4 — calculating the frequencies of the words in a text file. To recap, we are given a text, and we want to write out n most frequently occurring words in it.

We are going to decompose the problem into a composition of smaller transformations like before, we are just going to change a few things in order to better demonstrate how range views and actions interact with each other.

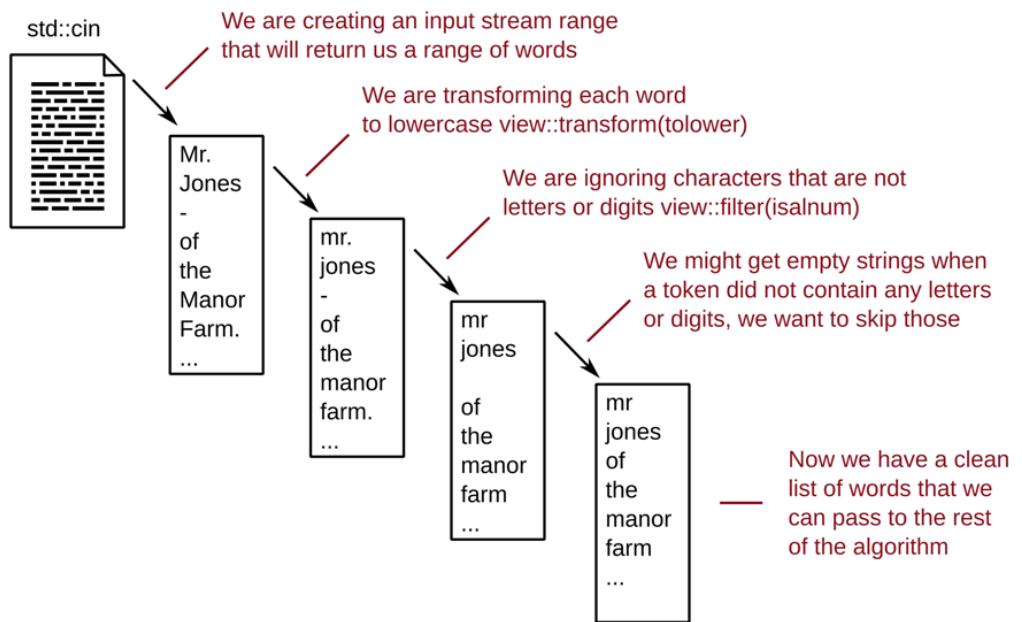
The first thing we need to do is to get a list of lowercase words without any special characters in them. Our data source is the input stream. We will use `std::cin` in this example.

The `range-v3` library provides us with a class template called `istream_range` which creates a stream of tokens from the input stream that we pass to it:

```
std::vector<std::string> words =
    istream_range<std::string>(std::cin);
```

In our case, since the tokens we want are of `std::string` type, it will read word by word from the standard input stream. Now, this is not enough, we want all the words to be lowercase and we do not want punctuation characters to be in them. So, we need to transform each word to lowercase, and to remove any non-alphanumeric characters from it.

Figure 7.6. We have an input stream we can read the words from. Before we can start calculating the word frequencies, we first need to have a list of words converted to lowercase with all punctuation removed



Listing 7.5. Getting a list of lowercase words that contain only letters or digits

```
std::vector<std::string> words =
    istream_range<std::string>(std::cin)
    | view::transform(string_to_lower)
    | view::transform(string_only_alnum)
    | view::remove_if(&std::string::empty);
```

- 1 All words need to be lowercase
- 2 We want to keep only letters and digits
- 3

- 1 All words need to be lowercase
- 2 We want to keep only letters and digits

- ③ We can sometimes get empty strings as the result (when a token did not contain a single letter or a digit) so we need to skip those

For the sake of completeness, we also need to implement `string_to_lower` and `string_only_alnum` functions. The former is a simple transformation where we convert each character in a string to lowercase, and the latter is a simple filter which skips characters that are not alpha-numeric. A `std::string` is a collection of characters, so we can manipulate it like any other range:

```
std::string string_to_lower(const std::string& s)
{
    return s | view::transform(tolower);
}

std::string string_only_alnum(const std::string& s)
{
    return s | view::filter(isalnum);
}
```

Now that we have all the words we need to process, we need to sort them. The `action::sort` needs a randomly-accessible collection, so it is quite lucky we declared `words` to be a `std::vector` of strings. We can simply request it to be sorted:

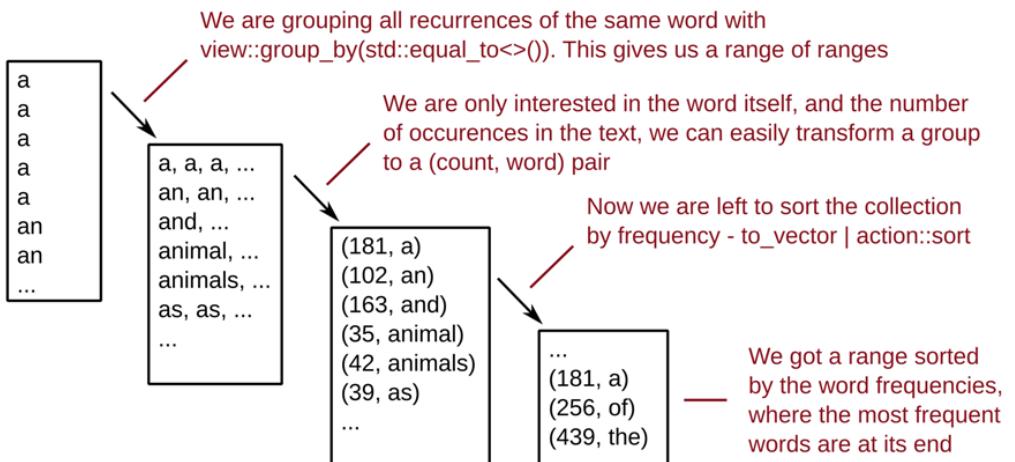
```
words |= action::sort;
```

Now that we have a sorted list, we can easily group the same ones using `view::group_by`. It will create a range of word groups (the groups are, incidentally, also ranges). Each group will contain the same word multiple times, as many times as it appeared in the original text.

We can transform this into pairs where the first item in the pair is the number of items in a group, and the second one is the word. This will give us a range containing all the words in the original text along with the number of occurrences for each of them.

Since we placed the frequency to be the first item of a pair, we can simply pass this range through `action::sort`. We can do it like in the previous code snippet by using the operator `|=`, but we can also do it inline by first converting the range to a vector. This will allow us to declare the `results` variable as `const`.

Figure 7.7. We got a range of sorted words. We just need to group the same words, count how many words we have in each group, and then sort them all by the number of occurrences



Listing 7.6. Getting a sorted list of frequency-word pairs from a sorted list of words (example:word-frequency)

```
const auto results =
    words | view::group_by(std::equal_to<>())
        | view::transform([](const auto& group) {
            const auto begin = std::begin(group);
            const auto end   = std::end(group);
            const auto count = distance(begin, end);
            const auto word  = *begin;

            return std::make_pair(count, word);
        })
    | to_vector | action::sort;
```

- ➊ We are grouping multiple occurrences of words from the words range
- ➋ We are getting the size of each group, and returning a pair consisting of the word frequency and the word itself
- ➌ We want to sort the words by their frequencies, so we first need to convert the range into a vector

The last step is to write the n most frequent words to the standard output. Since the results have been sorted in ascending order, and we need the most frequent words, not the least frequent ones, we first need to reverse the range, and then to take the first n elements.

```
for (auto value: results | view::reverse
        | view::take(n)) {
    std::cout << value.first << " " << value.second << std::endl;
}
```

That is it. We have implemented the same program that originally took a dozen pages in less than thirty lines. We have created a set of easily composable and highly reusable components, and not counting the output, we haven't used any for loops.

Range-based for loop and ranges

As previously mentioned, the range-based for loop started supporting sentinels in C++17.

This means that the above code will not compile on older compilers. If you are using an older compiler, the `range-v3` library provides a convenient `RANGES_FOR` macro which can be used as a replacement for the range-based for. Additionally, if we sorted the range of words in the same way we sorted the list of results (without the operator `|=`), we would have no mutable variables in our whole program.

```
RANGES_FOR (auto value, results | view::reverse
            | view::take(n)) {
    std::cout << value.first << " " << value.second << std::endl;
}
```

7.6 Summary

- One frequent source of errors when using STL algorithms is passing wrong iterators to them — sometimes even iterators belonging to separate collections
- Some collection-like structures don't know where they end. For those, it is customary to provide sentinel-like iterators which work, but have unnecessary performance overhead
- The ranges concept is an abstraction over any type of iterable data — it can model normal collections, input and output streams, database query result sets etc.
- The ranges proposal is planned for inclusion in C++20, but there are libraries that provide the same functionality today
- Range views do not own the data, and they are not able to change it. If you want to operate and change existing data, use actions instead
- Infinite ranges are a nice measure of algorithm generality — if something works for infinite ranges, it will work for the finite ones as well
- By using ranges and thinking of program logic in terms of range transformations, we can decompose the program into highly reusable components

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch07



Functional data structures

This chapter covers:

- Why are linked lists omnipresent in all FP languages
- Data sharing in functional data structures
- A trie structure that has efficient index-based access
- Comparison of a standard vector against its immutable counterpart

So far, we have talked mostly about the higher-lever functional programming concepts, and we spent quite some time talking about the benefits of programming without mutable state.

The problem is that our programs tend to have many moving parts. While talking about purity in chapter 5 we said that one of the options we have is a main component with mutable state where all other components are pure and just calculate what should be changed in the main component without actually changing anything. Only the main component would be able to perform the changes.

This approach creates a clear separation between the pure parts of the program and those that deal with the mutable state. The problem here is that it is often not easy to design software like this because we need to pay attention on the order in which the calculated changes to the state should be applied, and if we do not do it properly, we might encounter data-races similar to those we have when working with mutable state in a concurrent environment.

Because of this, it is sometimes necessary to avoid all mutations — not even to have the central mutable state. If we used the standard data structures, we would

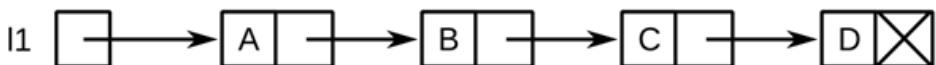
have to copy the data around each time we need a new version of it which is inefficient. Each time that we want to change an element in a collection we would need to create a new collection which is the same as the old one but with the desired element changed.

8.1 **Immutable linked lists**

Instead, we will have to think of alternative data structures that will be efficient to copy around, and to create modified copies.

One of the oldest data structures optimized for this type of usage are singly linked lists which were the basis for the oldest FP language—Lisp (short for List Processing). Now, we know that linked lists are inefficient for most tasks because they have bad memory locality which does not play well with the caching facilities of modern hardware, but they are the simplest example of how an immutable structure can be implemented, so they are the perfect structure for the introduction to the topic.

Figure 8.1. Singly linked list – each node contains a value and a pointer to the next one



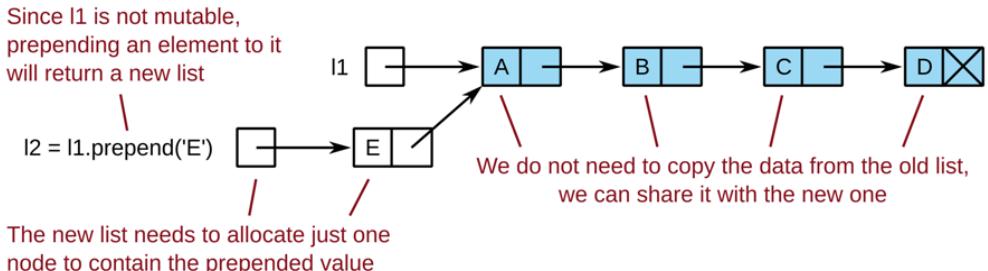
A list is simply a collection of nodes, where each node contains a single value, and a pointer to the next node in the list (`nullptr` if it is the last element of the list). The basic operations on a list are adding and removing elements to the start or the end of the list, inserting or removing elements from the middle, or changing the stored value in a specific node.

We will focus on modifying the start and the end since all other operations can be expressed through them. Mind that when we say "*modifying*", it actually means "*creating a new modified list from the old one, while the old one remains unchanged*".

8.1.1 **Adding and removing elements to/from the start of the list**

Let's first talk about modifying the start of the list. If we have a list, and we agreed that an existing list should never change, creating a new list with a specified value prepended is trivial — we can just create a new node with said value which will point to the head of the previously created list.

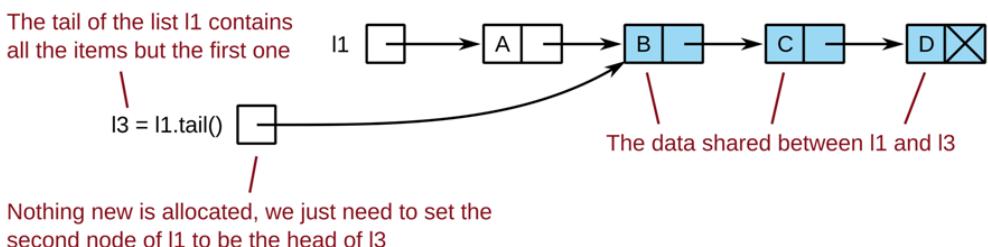
Figure 8.2. Prepending an element to an immutable list returns a new list. It does not need to copy any of the existing nodes from the original list, it can simply reuse them. This approach is not viable in the mutable version of the list because changing one list would have the side-effect of changing all other lists that share the data with it.



This will give us two lists — the old one which has not changed, and the new one that has literally the same elements as the old one, just with an element prepended.

Removing an element from the start is similar, we just need to create a new head for the list which points to the second element in the original list. Again, we got two lists — one with the old data, and one with the new data.

Figure 8.3. Getting the tail of a list creates a new list from the original one that will contain the same elements modulo the head of the original list. The resulting list can share all the nodes with the original one, and still nothing needs to be copied.



These operations are quite efficient ($O(1)$), both concerning the execution time and memory. We do not need to copy any data whatsoever as both lists share the same data.

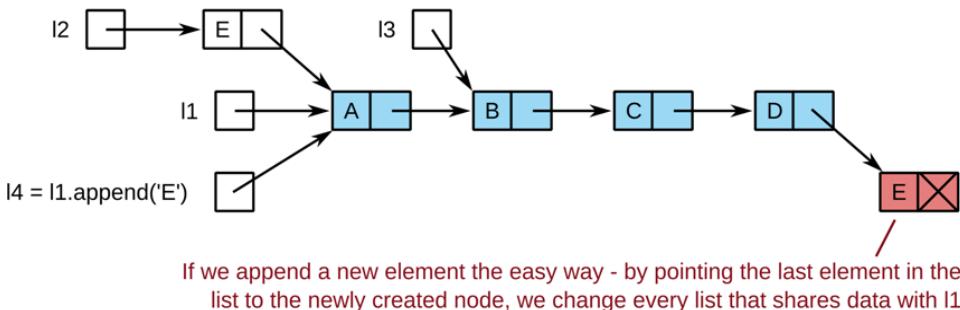
One important thing to note is that if we have the guarantee that the old list can never be changed, and we do not provide any function to change the newly added element, we can guarantee that the new list is also immutable.

8.1.2 Adding and removing elements to/from the end of the list

While changing the beginning of a list is trivial, the end is more problematic. The

usual approach of appending to the end of a list is to find the last element and make it to point to a new node. The problem is that this can not work for the immutable list.

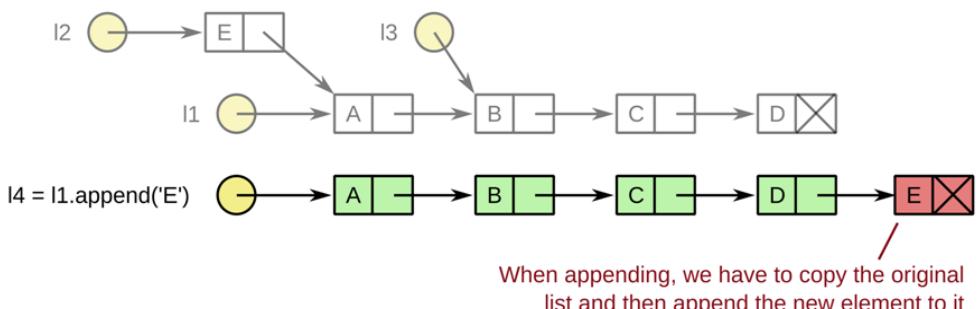
Figure 8.4. If we tried to do the same trick of avoiding copying the data when appending to a list, and just make the last element in the old list point to the newly created node, we will change every list that shares its data with the list we are modifying. It is interesting to note that if two lists share any nodes, then they share the end as well.



If we were just to create the new node, and make the last node in the old list point to it, we would change the old list. And not only the old list, but all lists that share data with it — all of them will get a new element appended.

This means that we can not efficiently append an element to an immutable list — we need to copy the original one, and append a new element to the copy.

Figure 8.5. If we want to create a list that is the same as the original one, but has an extra element at its end, we will need to copy all nodes from the original list and only then point the last node to the newly created node that contains the appended value.

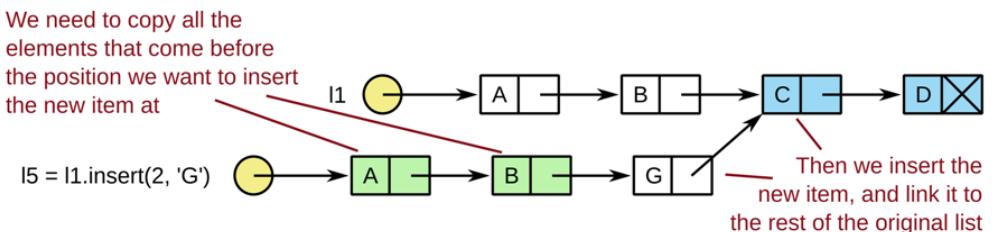


The same goes for removing the element from the end. If we removed it in-place, we would change all already existing lists that share the same data as the current one.

8.1.3 Adding and removing elements from/to the middle of the list

In order to add or remove an element from the middle of the list, we will need to create a copy of all elements that come before it in the original list — just like it was the case when changing the end of the list.

Figure 8.6. Inserting an element into the middle of the list is equivalent to removing all the items from the original list that should come before the element we are inserting, then prepend that element and prepend all the previously removed items. This means that the only nodes that we will share with the original list are nodes that come after the item we have inserted.



More specifically, we would need to remove all the elements that come before the location we want changed. It will give us the part of the original list that we can reuse in our resulting list. Then we need to insert the desired element, and re-add all the elements from the old list that should go in front of it.

Listing 8.1. Complexity of the linked list functions

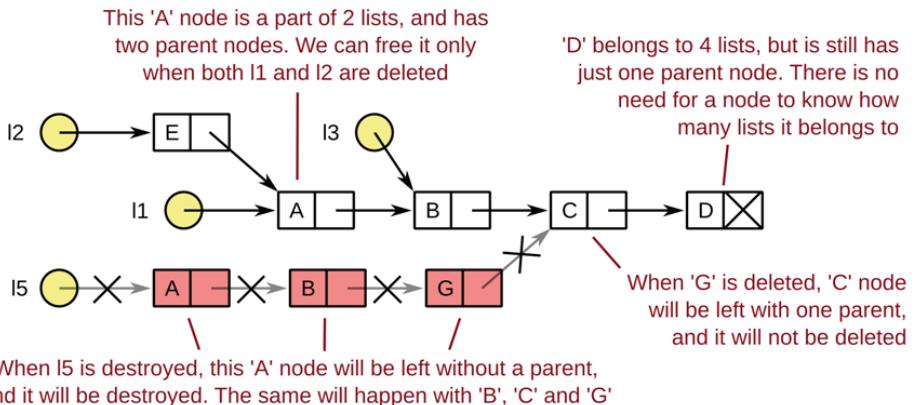
$O(1)$	$O(\log n)$	$O(n)$
<code>'head'</code>		appending
<code>'with_prepended'</code>		concatenation
<code>'tail'</code>		inserting at some position
		getting the last element
		getting the n -th element

These inefficiencies makes the singly linked list efficient only for implementing stack-like structures where we always deal only with the beginning of a list.

8.1.4 Memory management

One of the requirements for the implementation of the list is that the nodes need to be dynamically allocated. If we defined a list just to be a structure that holds a value and a pointer to another list, we might get into trouble when one of the lists goes out of the scope and gets deleted. All the lists that depend on that list will become invalid.

Figure 8.7. When a list is deleted, we need to destroy the nodes that no other list uses. The nodes do not know to how many lists they belong to, but they need to know how many immediate parents they have. When all parents of a node are destroyed, it means that we can destroy that node as well – there is no list it is the head of, nor any node that points to it.



Since we have to dynamically allocate the nodes, we need to have a way to get rid of them when they are not needed anymore. Namely, when a list goes out of the scope, we need to free all the nodes that belong only to this list, and none of the nodes that are shared.

For example, when I5 from the Figure 8.7 goes out of the scope, we need to delete all the nodes up to 'C', and nothing else.

Ideally, we would like to use the smart pointers to implement automatic memory clean-up. For regular linked lists, we might go for the `std::unique_ptr` since the head of the list owns its tail, and the tail should be destroyed if the head is.

But it is not the case here. As can be seen in the figure, each node can have multiple owners because it can belong to multiple lists. Therefore, we need to use the `std::shared_ptr`. It keeps a reference-count and automatically destroys the object instance it points to when the count reaches zero. In our case, that will happen when we want to destroy a node that does not belong to any other list.

Garbage collection

Most functional programming languages rely on garbage collection to free up the memory that is no longer used.

In C++, we can achieve the same by using smart pointers. In the case of most data structures, the ownership model is quite clear.

For example, for lists, the list head is the owner (not necessarily the only owner since multiple lists can share the same data) of the rest of the list.

The simplest structure for the `list` and its internal `node` class could look like this:

```
template <typename T>
class list {
public:
    // ...

private:
    struct node {
        T value;
        std::shared_ptr<node> tail;
    };

    std::shared_ptr<node> m_head;
};
```

This will correctly free all nodes that are no longer needed when an instance of `list` is destroyed. The `list` will check whether it is the sole owner of the node that represents its head, and destroy it if it is true. Then, the destructor of the node will check whether it is the only owner of the node representing its tail, and destroy it if it is. And this will continue until a node with another owner is found, or until we reach the end of the list.

The problem here is that all these destructors are invoked recursively, and if we had a large enough list, we might end up overflowing the stack. In order to avoid this, we will need to provide a custom destructor that will flatten-out the recursion into a single loop:

Listing 8.2. Flattening the recursive structure destruction

```
~node()
{
    auto next_node = std::move(tail);      ④
    while (next_node) {
        if (!next_node.unique()) break;    ①

        std::shared_ptr<node> tail;       ②
        swap(tail, next_node->tail);     ②
        next_node.reset();               ③

        next_node = std::move(tail);      ⑤
    }
}
```

- ① If we are not the only owner of the node, do not do anything
- ② We are stealing the tail of the node we are processing, to stop the node's destructor from destroying it recursively
- ③ We can now freely destroy the node — there will be no recursive calls as we have set its tail to be a `nullptr`
- ④ `std::move` is necessary here; otherwise `next_node.unique()` below will never return `true`

➅ `std::move` is not necessary in this case, but it could improve the performance

If we wanted to make this code thread-safe, we would need to do it manually. While the reference counting in `std::shared_ptr` is thread-safe, we have multiple separate calls that might modify it in our implementation.

8.2 Immutable vector-like data structure

Because of their inefficiencies, lists are unsuitable for most use-cases. We need a structure that will have efficient operations and fast look-up.

Now, if we look at `std::vector`, it has a few things going for it, mainly because it stores all its elements in one chunk of memory:

- Fast index-based access because given the index, it can directly calculate the location in memory of the corresponding element — simply by adding the index to the address of the first element in the vector;
- When CPU needs a value from memory, modern hardware architectures do not fetch only that value, but they fetch a whole chunk of surrounding memory and cache it for faster access. When iterating over values in a `std::vector`, this comes in handy because accessing the first element will fetch a few of them at the same time, which makes the consequent access faster;
- Even in the problematic cases, when the vector needs to reallocate memory, and to copy or move the previous data to the new location, it will benefit from the fact that everything is in the same block of memory because modern systems are optimized to move around blocks of memory.

The problem with `std::vector` is that if we wanted to use it as an immutable structure, we would need to copy all the data inside every time we need to create a modified version of it. So, we need an alternative that will behave as similarly as possible to it.

One of the popular alternatives is the Bitmapped Vector Trie, a data structure invented by Rick Hickey for the Clojure programming language (which is, in turn, heavily inspired by a paper on Ideal Hash Trees written by Phil Bagwell).

Copy on write

We often pass objects around to functions that just use the object without modifying it. It is superfluous to always copy the whole object when passing it just for case that the invoked function does change it.

Copy-on-write (COW) or *lazy copy* is an optimization which delays creating the copy of the data until somebody tries to modify it.

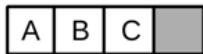
When the copy constructor or the assignment operator is called, the object just saves the reference to the original data and nothing else.

When the user invokes any member function that is supposed to change the object, we can not allow it to change the original data, so we need to create a proper copy and then change it. We can only change the original data in-place if no other object has access to it.

You can find more information about COW in Herb Sutter's book "More exceptional C++".

The structure starts with a copy-on-write vector, but with its maximum size limited to some predefined number m . In order for the operations on this structure to be as efficient as possible, the number m needs to be a power of 2 (we will see why later). Most of the implementations set the limit to 32, but in order to keep the figures simpler, we are going to draw the vectors as if the limit was 4.

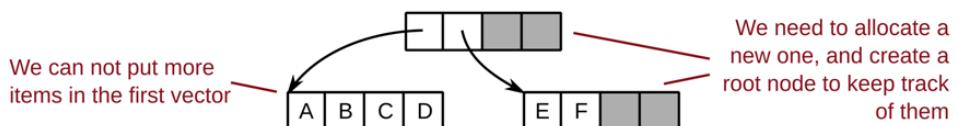
Figure 8.8. If we have at most m elements, the bitmapped vector trie will contain just one COW vector.



Until we reach that limit, the structure behaves like an ordinary COW vector. When we try to insert the next element a new vector with the same limit will be created, and that new element will be its first element. Now we have two vectors — one that is full, and the other one containing a single item. When a new element is appended, it will go into the second vector until it gets full, when a new vector will be created.

In order to keep them *together* (they are still, logically, a part of the same structure) we will create a vector of vectors — a vector that will contain pointers to at most m sub-vectors that contain the actual data.

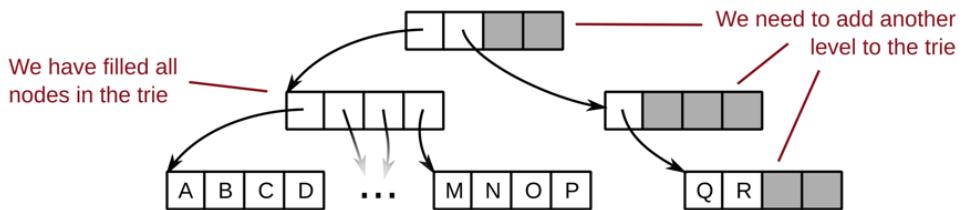
Figure 8.9. When we add an item that can not fit into the vector of capacity m , we will create another vector and place the new element inside it. In order to be able to track all the vectors we create, we will need an index — a new vector of capacity m which will hold the pointer to the vectors we created to hold the actual data. We get a trie with two levels.



When the top-level vector gets full, we create a new level. This means that if we have at most m items, we will have just one level, if we have at most $m \times m$ items, we will have two levels, if we have at most $m \times m \times m$ items, we will have three levels, and so on.

Figure 8.10. When we fill up the root vector and we can not add more items into it, we will

create another trie in which we can store more element, and in order to keep track of all these tries, we will create a new root node.



8.2.1 Element lookup in bitmapped vector tries

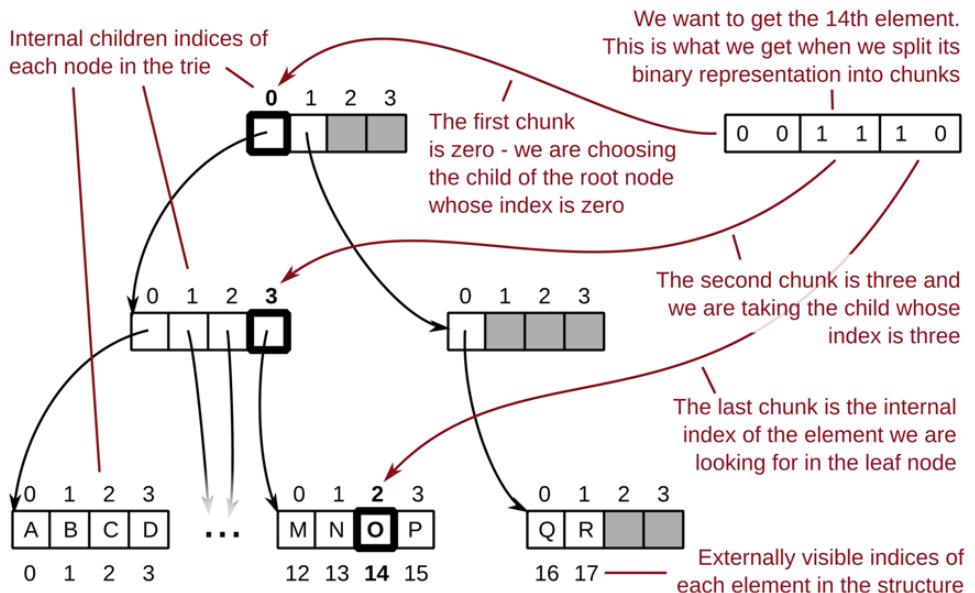
The structure we created is a special kind of *trie* (prefix tree) where the actual values are stored in the leaf nodes, and the non-leaf nodes hold no data — they just keep the pointers to the nodes in the lower level. All the leaf nodes are sorted by index — the leftmost leaf will always contain the values from 0 to $m-1$, the next leaf will contain the items from m to $2m - 1$, etc.

We said we want to have a fast index-based lookup. If we are given the index i of an element, how do we find the element?

The idea is quite simple, if we have only one level — just a single vector — then i will be index of the element in the vector itself. If we have two levels, the upper level will contain m elements for each of its indices (it will contain a pointer to a vector of m elements). The first leaf vector will contain items up to $m-1$, the second will contain items from m up to $2m-1$, the third from $2m$ to $3m-1$ and so on. This means that we can easily find in which leaf the i -th element is in — it will be the leaf whose index is i / m . And analogous for the higher levels.

So, when we are given an index of an element, we can look at it as if it is an array of bits, and not a number. We can split that array into chunks. This will give us the path from the root node to the element we are searching for — each chunk is the index of the child node we need to visit next, and the last chunk is the index of the element in the leaf vector (Figure 8.11).

Figure 8.11. It is easy to find the item for a given index. Each level has internal indices going from 0 to m . We said that m should be a power of 2 (in this figure, $m = 4$ although in reality it is usually 32). This means that the internal indices of each node, if looked as arrays of bits, will go from 00 to 11. We can look at the index of the requested element as an array of bits which we split into chunks of two-by-two. Each chunk will be the internal index in the corresponding trie node.

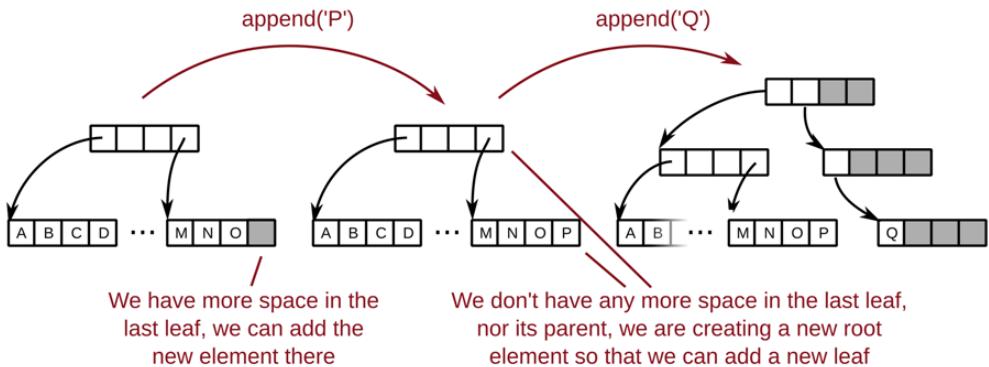


This gives us logarithmic lookup, with m being the logarithm base. We said that most of the time we will choose m to be 32, which will make our trie very shallow in most cases. For example, a trie of depth 5 can contain 33 million items. Because of this, and the fact that memory in the system is limited, it is considered that, in practice, the lookup complexity is done in constant-time ($O(1)$) for tries where m is 32 or greater — even if we fill the whole addressable memory with one collection, we will still not go above 13 levels (the exact number is not that important, the fact that it is a fixed number is).

8.2.2 Appending elements to bitmapmed vector tries

Now that we have seen how to perform lookup on the bitmapmed vector trie, let's move on to the next topic — how to update it. This would be trivial if we were implementing a mutable version of the structure — we would just need to find the first free space and place the new element there. If all the leaf nodes are filled, we would simply create a new leaf.

Figure 8.12. In a mutable binary vector trie, appending a new element is simple — if there is an empty space left in the last leaf node, we can just store the value there. If all leaves are full, we need to create a new one. We will need to create a pointer to the new leaf in the level above it. Recursively, if there is room in the parent node, we will store the pointer there. If not, we will create a new node, and then we will need to store a pointer to it in the level above it. This is repeated until we reach the root. If there is no room in the root node, we create a new one and make it a parent of the previous one.

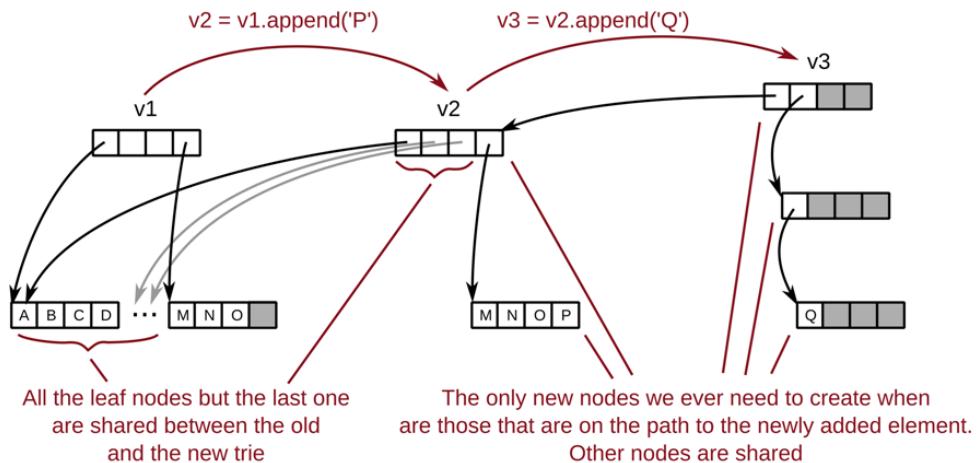


The problem is that we want to keep the old versions untouched. We can do it in the similar way we did for the linked lists — we will create a new trie that will share as much data as possible with the previous one.

If we compare the three tries in the previous figure, we can see that the differences are not big — the main difference is that the leaves differ. This means that we need to copy the leaf that is affected by the append. Then, since the leaf is changed, it means that the new trie also needs to have a new parent for that leaf. It will also need a new parent of that parent, and so on.

So, in order to append an element, we will need not only to copy the leaf node, but the whole path down to the inserted element.

Figure 8.13. In the immutable binary vector trie, the principle of adding a new element is the same. The difference here is that, since we want to have the previous collection untouched, we need to create a new trie that will hold the new data. Fortunately, like it was the case with the linked lists, we can share most of the data between the old and the new trie. The only nodes that we need to create are the nodes on the path to the element we are adding. All other nodes can be shared.

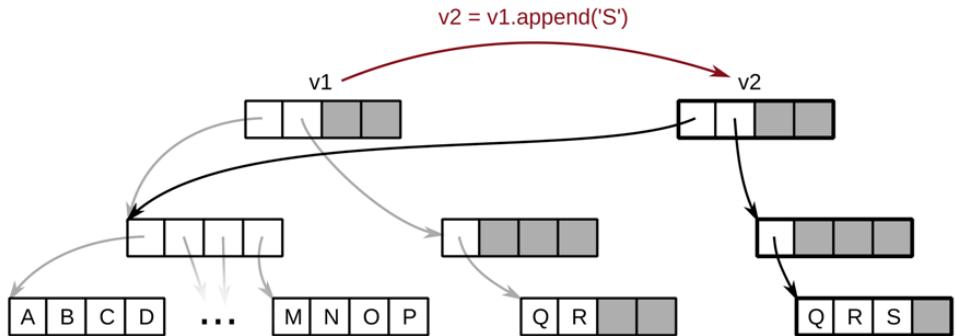


We have a few cases to cover here:

1. There is room for a new element in the last leaf
2. There is room for a new element in any of the non-leaf nodes
3. All nodes are full

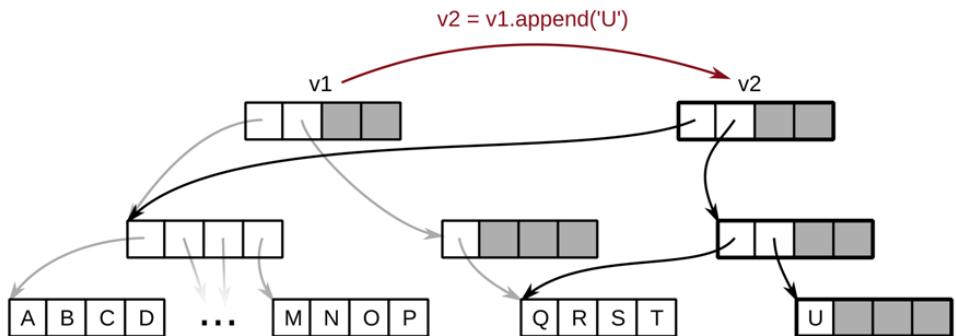
The first case is the simplest one — we just need to copy the path to the leaf node, duplicate the leaf node, and insert the new element into the newly created leaf.

Figure 8.14. When appending a new element to a trie that has a leaf with room for it, we have to duplicate that leaf and all its parents. We can then just add the new value into the newly copied leaf.



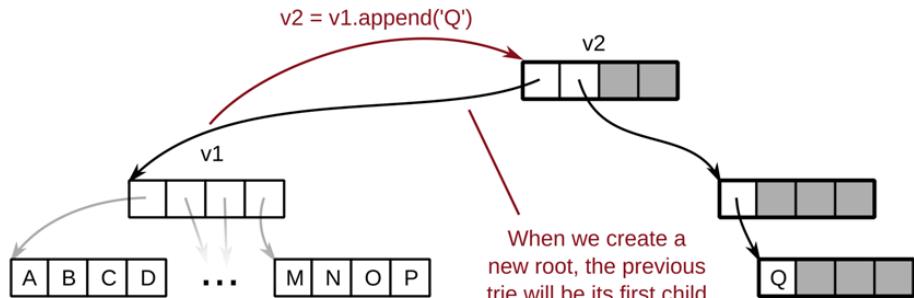
In the second case, we have to find the lowest node that is not full, and create all the levels below it until we create the leaf which we will add the new element to.

Figure 8.15. In the case when we do not have any room in the last leaf, we need to create a new one that contains just the element that we are appending to the trie. The newly created leaf should be inserted into the layer above it.



The third case is the most complex one. If we do not have a node that has any free space in it, it means that we can not store any more elements under the current root node. This means we need to create a new root node, and to make the previous root be its first element. After that, we come to the same situation as in the second case — we have the room on a non-leaf node.

Figure 8.16. We have a corner case when there is no space in any of the leaves, and there is also no space in any of the other nodes including the root. In this case, we are unable to create a new path inside this trie, so we need to create a new root item that will link to the previous trie, and a new trie with the same depth that contains just the newly added item.



What is the efficiency of appending? We are either copying the path nodes, or we are allocating new ones, and sometimes we need to create a new root node. All of these operations, by themselves, are constant-time (each node has at most a constant number of elements). If we use the same reasoning behind the claim that the lookup is constant-time, we can consider that we have a constant number of nodes to copy or allocate. This means that appending to this structure is also $O(1)$.

8.2.3 Updating elements in bitmapped vector tries

Now that we have seen how to append an element, let's analyze what we need to do when we want to update one. Again, let's think of what would be changed in the structure if we made it mutable and we changed an element at a predefined position.

We would need to find the leaf which contains the element, and modify the element. We do not need to create any new nodes — we get the same trie structure, just with a single value changed.

This means that updating is similar to the first case we had when appending — when the last leaf had the space to store the new element. The new trie will be able to share everything with the old one apart from the path which we used to reach the element we are updating.

8.2.4 Removing elements from the end of the bitmapped vector trie

Removing an element from the end is quite similar to appending, just reverse. We will have two separate cases:

1. The last leaf contains more than one element
2. The last leaf contains only one element
3. The root will contain one element after removing an element

In the first case, we just need to copy the path to the leaf node which contains the element we want removed, and remove the element from the newly created leaf node.

In the second case, we have a leaf that has only one element. This means that in the newly created trie, that leaf node should not exist. Again, this means that we will copy the whole path, and trim the empty nodes from its end. We will remove the leaf, but we will also remove its parent if it no longer has any children, and its parent, and so on.

If we get a root node with just a single child after trimming the empty nodes, we need to set that child to be the new root to lower the number of levels our trie has.

8.2.5 Other operations and the overall efficiency of bitmapped vector tries

We have seen how we can update elements, or how to append or remove them from the end, and we have seen that these operations on the bitmapped vector trie are quite efficient.

What about other operations like prepending or inserting elements at a specified position? What about concatenation?

Unfortunately, with these operations we can not do anything smart. Prepending and inserting needs to shift all other elements one step to the right. And concatenation needs to allocate enough leaf nodes and copy them from the collection we are concatenating.

Listing 8.3. Complexity of the bitmapped vector trie functions

<code>`0(1)`</code>	<code>`0(log n)`</code>	<code>O(n)</code>
<code>`operator[]`</code>		prepending
<code>`with_appended`</code>		concatenation
<code>`at(int i)`</code>		inserting at some position

These are the same complexities that we have with `std::vector`. Changing the existing elements, adding and removing elements to and from the end of the collection are constant-time, and inserting or removing elements from the beginning or the middle of the collection is linear.

Now, even if the algorithm complexity is the same, the bitmapped vector trie can not be exactly as fast as `std::vector`. When we access an element, it has to go through a several layers of indirection, and it will have cache misses slightly more often because it does not keep all of its elements in a contiguous block of memory, but in several contiguous chunks.

One case where it beats the ordinary vector is copying which is what we needed in the first place. We got a structure that has a comparable speed to `std::vector` while

being optimized for immutable usage — where instead of modifying existing values, we are always creating slightly modified copies of the original data.

We have talked often about immutability in the previous chapters, and how to leverage the features of C++ like `const` to implement safer and more concise code. The standard collection classes which can be found in the STL are not optimized for this use-case. They are meant to be used in the mutable manner. They can not be safely shared amongst different components unless we copy them which is inefficient.

The structures we have seen in this chapter, especially the bitmapped vector trie, live on the other side of the spectrum. They are designed to be as efficient as possible when we want to use them in the pure functional style — to never change any data we set once. While they do have somewhat lower performance, they make copying blazingly fast.

8.3 Summary

- The main optimization in all immutable data structures, which are also called persistent, is the data sharing. It allows us to keep many slightly modified versions of the same data without the memory overhead we would have if we used the classes from the standard library;
- Having an efficient way to store past values of a variable allows us to travel through the history of our program — if we store all the previous states of our program, we can jump back to any of them whenever we want;
- Similar modifications exist for other structures as well. For example, if we needed an immutable associative container, we could use the red-black trees, just modified to share the data between different instances similar to what we had done for the bitmapped vector trie;
- Immutable data structures have always been an active area of research. A lot of contributions came from academia (as it is often the case with functional programming), but also from normal developers;
- It is always worth checking which structures would be suited best for any particular case. Unlike the go-to collection in C++ — the `std::vector`, all immutable data structures have their downsides (even if they are as magical as bitmapped vector tries are) so some of them might not be well suited for all cases you might have;
- For more information on immutable data structures, the book that is usually recommended is Okasaki's "Purely Functional Data Structures".

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch08

Algebraic data types and pattern matching



This chapter covers:

- Removing invalid states from programs
- Algebraic data types
- Handling errors with optional values and variants
- Creating overloaded function objects
- Handling algebraic data types through pattern matching

During the course of this book, we have tackled a few of the problems that the program state introduces. We have seen how to design software without mutable state, and how we can implement data structures that are efficient to copy.

But we still haven't removed all the problems of program state. We still haven't covered the unexpected or invalid states. Consider the following situation — we have an application that counts the total number of words in a web page. The program can start counting as soon as it retrieves a part of the web page. It will have three basic states:

- the initial state, when the counting process haven't started yet;
- the counting state, while the web page is being retrieved and the counting is in progress
- the final state where the web page is fully retrieved, and all the words have been counted.

When we need to implement something like this, we usually create a class that will

contain all the data we need. We end up with a class that contains the handler through which the web page is accessed (a socket or a data stream), the counter to contain the number of words, and flags which indicate whether the process has started and whether it has finished.

We could have a structure like this:

```
struct state_t {
    bool started = false;
    bool finished = false;
    unsigned count = 0;
    socket_t web_page;
};
```

The first thing that comes to mind is that `started` and `finished` don't need to be separate Boolean flags, that we could replace them with an enum which could contain three different values `init`, `running` and `finished`. If we keep them as separate values, we open ourselves to having invalid states like `started` being false while `finished` is true.

We will have the same problems with other variables as well. The `count` should never be greater than zero if `started` is false, and it should never be updated after `finished` becomes true, the `web_page` socket should be open only if `started` is true, and `finished` is false, etc.

When we replaced the Boolean values with an enum that can have three different values, we have lowered the number of states our program can be in. Which means that we have removed some of the invalid states. It would be useful if we could do the same for other variables as well.

9.1 Algebraic data types

In the functional world, building new types from the old ones is usually done through two operations — sum and product (these new types are called *algebraic* because of this).

A product of two types `A` and `B` is a new type that contains an instance of `A` and an instance of `B` (it will be a Cartesian product of the set of all values of type `A` and the set of all values of `B`). Similarly, the product of multiple types is a new type that contains an instance of each of the types involved in the product.

In the example of counting words in a web page, the `state_t` is a product of two Booleans, one `unsigned` and one `socket_t`.

This is something we are accustomed to in C++ — every time we want to combine multiple types into one, we either create a new class, or use `std::pair` or `std::tuple` when we don't need the member values to be named.

Pairs and tuples

The `std::pair` and `std::tuple` are useful generic types for creating quick-and-dirty product types. The `std::pair` is a product of two types, while a `std::tuple` can have as many types as we want.

One of the most useful features of pairs and tuples is that when we create product types with them, we get lexicographical comparison operators for free.

It isn't rare to see people implement comparison operators for their classes through tuples. For example, if we had a class that holds a person's name and surname, and we wanted to implement a less-than operator for it which will compare the surname first, and then the name, we could do it like this:

```
bool operator<(const person_t& left, const person_t& right)
{
    return std::tie(left.m_surname, left.m_name) <
           std::tie(right.m_surname, right.m_name);
}
```

The `std::tie` function creates a tuple of references to the values pass to it. This means that no copies of the original data will be created when creating the tuple, that the original strings will be compared.

The problem with pairs and tuples is that the values stored in them aren't named. If we see a function that returns a `std::pair<std::string, int>` we aren't able to tell what those values mean just from the type itself. Whereas if the return type was a structure with member variables called `full_name` and `age`, we wouldn't have that problem.

Because of this, pairs and tuples should be used rarely, and their usage should always be localized. It is a bad idea to ever make them a part of a public API (even if some parts of STL do it).

The sum types aren't as prominent in C++ as product types. The sum type of types A and B is a type that can hold an instance of A or an instance of B, but not both at the same time.

Enums as sum types

Enums are a special kind of sum types. We define an enum by specifying the different values it can hold. And an instance of that enum type can hold exactly one of those values.

If we treat these values as one-element sets, the enum is a sum type of those sets.

You can think of sum types as a generalization of an enum — instead of providing one-element sets when defining the sum type, we specify sets with arbitrary number of elements.

In our scenario, we have three main states — the initial state, the counting state, and the finished state. The initial state doesn't need to contain any additional

information, the counting state contains a counter and the handler for accessing the web page, and the finished state needs only to contain the final word count.

Since these states are mutually exclusive, when we write the program for this scenario, we can model the state by creating a sum type of three different types — one type for each state.

9.1.1 Sum types through inheritance

There are multiple ways to implement sum types in C++. One way is to create a class hierarchy where we will have a super-class representing the sum type, and derived classes to represent summed types.

If we wanted to represent the three states for our program, we could create a super-class `state_t` and three sub-classes — one for each of the main states — `init_t`, `running_t` and `finished_t`.

The `state_t` class can be empty since we only need it as a placeholder — we will only have a pointer to its sub-classes. When we need to check which state we are in, we can simply use the `dynamic_cast`. Alternatively, since dynamic casting is slow, we can add an integer tag to the super class which we will use to differentiate between the sub-classes.

Listing 9.1. Tagged super-class for creating sum types through inheritance

```
class state_t {
protected:
    state_t(int type)                                ①
        : type(type)                                ②
    {
    }

public:
    virtual ~state_t() {};
    int type;
};


```

- ① It shouldn't be possible to create instances of this class, so we are making the constructor protected. It can only be called by classes that inherit from `state_t`
- ② Each subclass should pass a different value for the `type` argument. We will be able to use it as an efficient replacement for `dynamic_cast`.

We have created a class that we can't instantiate directly. This means that we can only use it as a handler to instances of its sub-classes. We will create a sub-class for each state that we want to represent.

Listing 9.2. Types to denote different states

```

class init_t : public state_t {
public:
    enum { id = 0 };          ①
    init_t()                 ①
        : state_t(id)        ①
    {                        ①
    }                        ①
};

class running_t : public state_t {
public:
    enum { id = 1 };
    running_t()
        : state_t(id)
    {
    }

    unsigned count() const
    {
        return m_count;
    }

    // ...

private:
    unsigned m_count = 0;      ②
    socket_t m_web_page;     ②
};

class finished_t : public state_t {
public:
    enum { id = 2 };
    finished_t(unsigned count) ③
        : state_t(id)         ③
        , m_count(count)      ③
    {
    }

    unsigned count() const
    {
        return m_count;
    }

private:
    unsigned m_count;          ④
};

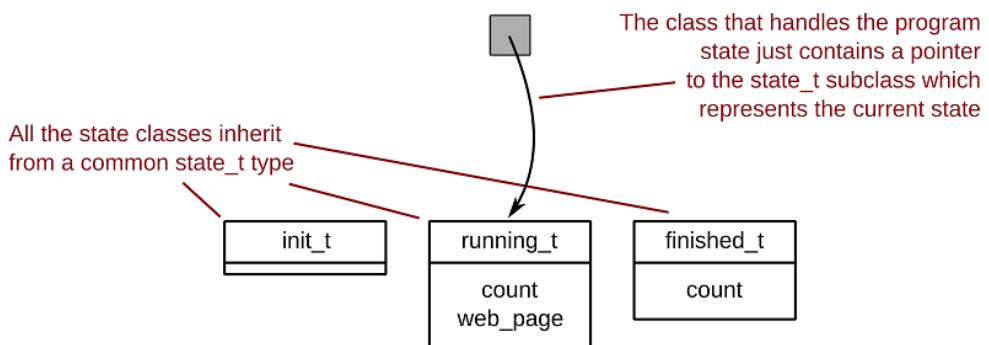
```

- ① The class which represents the initial state doesn't need to hold any data — we still don't have the handler to the web page nor the counter. It only needs to set the type to its ID (zero)

- ② For the running state, we need a counter and the handler to the web page whose words we want to count
- ③ When the counting is finished, we no longer need the handler to the web page. We just need the calculated value.

The main program now just needs to have a pointer to `state_t` (an ordinary pointer, or a `unique_ptr`). Initially, it should point to an instance of `init_t`, and when the state changes, that instance should be destroyed, and replaced with an instance of another state sub-type.

Figure 9.1. Implementing sum types with inheritance is simple—we can create multiple classes that inherit a common `state_t` class and the current state is denoted by a pointer pointing to an instance of one of these classes.



Listing 9.3. The main program

```
class program_t {
public:
    program_t()
        : m_state(std::make_unique<init_t>())           ①
    {
    }

    // ...

    void counting_finished()
    {
        assert(m_state->type == running_t::id);          ②

        auto state = static_cast<running_t*>(            ③
            m_state.get());

        m_state = std::make_unique<finished_t>(
            state->count());                           ④
    }
}

private:
```

```
    std::unique_ptr<state_t> m_state;
};
```

- ➊ The initial state should be an instance of `init_t`. We shouldn't allow `m_state` to be null at any point
- ➋ If the counting is finished, it should mean that we were in the counting state. If we can't guarantee that this assumption holds, we can use `if-else` instead of `assert`
- ➌ We know the exact type of the class `m_state` points to, we can statically cast to it
- ➍ We are switching to the new state that holds the end result. The previous state is destroyed

With this approach, we can no longer have invalid states. The count can't be greater than zero if we haven't started counting yet (the count doesn't even exist in that case). We can't have the count accidentally change after the counting process is finished, and we know exactly which state we are in at all times.

What's more, we don't need to pay attention to lifetimes of resources we acquire for specific states. Just focus on the `web_page` socket for a moment. In our original approach of putting all variables that we will need in the `state_t` structure, we could forget to close the socket after we finish reading from it. The socket instance continues to live for as long as the instance of `state_t` lives. By using the sum type, all the resources needed for a specific state will automatically be released as soon as we switch to another state. In this case, the destructor of `running_t` will close the `web_page` socket.

When we use inheritance to implement sum types, we get open sum types. The state can be any class that inherits from `state_t`, which makes it easily extendable. While this is sometimes useful, we usually know exactly which possible states our program should have, and we don't need to allow other components to extend the set of states dynamically.

Also, the inheritance approach has a few downsides. If we want to keep its openness, we need to use virtual dispatch (at least for the destructors), we need to have the type tags in order not to rely on slow dynamic casts, and we need to allocate the state objects dynamically on the heap. We also need to take care of the `m_state` pointer to always be valid.

9.1.2 Sum types through unions and `std::variant`

The alternative approach to inheritance-based sum types is `std::variant` which provides us a type-safe implementation of unions. With `std::variant`, we can define closed sum types — sum types that can hold exactly the types we specify and nothing else.

When using inheritance to implement sum types, the type of the `m_state` member variable is a (smart) pointer to `state_t`. The type itself doesn't communicate anything about the possible states — `m_state` can point to any object that inherits from `state_t`.

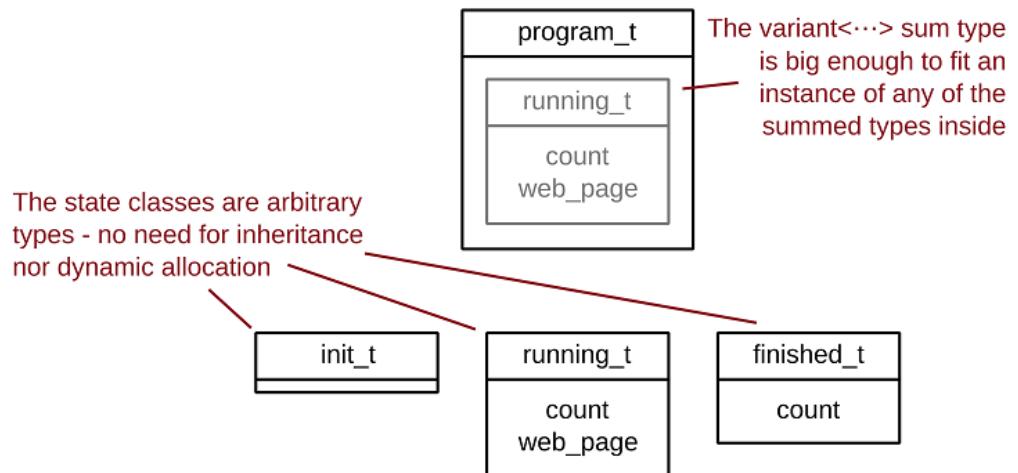
With `std::variant`, all the types that we want to use to handle the program state

need to be explicitly specified when defining the `m_state` variable — they will be encoded in the type of `m_state`. This means that in order to extend the sum type, we will need to change the type definition, which wasn't the case with the solution based on inheritance.

An alternatives to `std::variant` with older compilers

The `std::variant` has been introduced in C++17. If you have an older compiler and an older STL implementation that doesn't support C++17 features, you can use the `boost::variant` instead.

Figure 9.2. Variants can be used to define proper sum types. We get a value type that can contain a value of any of the summed types inside of its own memory space. This means that the variant instance will have at least the size of the biggest type we are summing, regardless of the value it currently holds



In order to implement our program state using `std::variant`, we can reuse the definitions of `init_t`, `running_t` and `finished_t` classes, with the exception that we don't need them to be subclasses of some common type, and we don't need to create integer tags for them::

```
class init_t {
};

class running_t {
public:
    unsigned count() const
    {
        return m_count;
    }
}
```

```
// ...

private:
    unsigned m_count = 0;
    socket_t m_web_page;
};

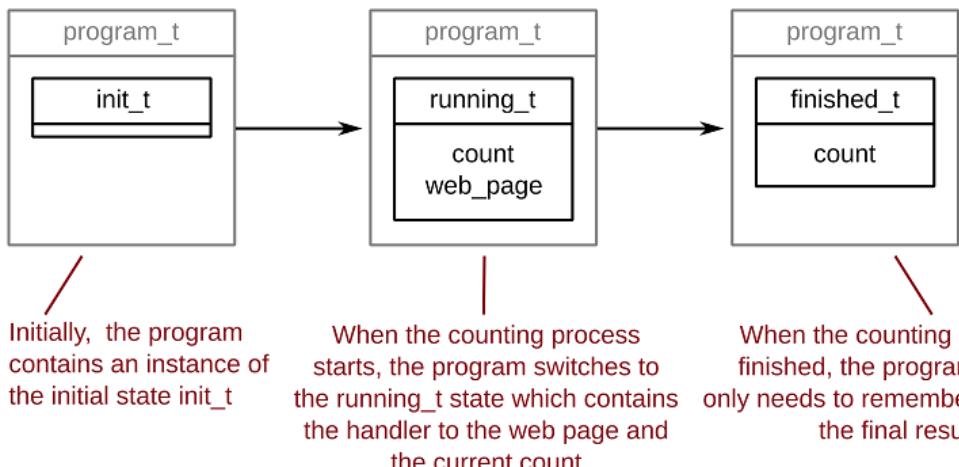
class finished_t {
public:
    finished_t(unsigned count)
        : m_count(count)
    {
    }

    unsigned count() const
    {
        return m_count;
    }

private:
    unsigned m_count;
};
```

All the boiler-plate we had to add in order for inheritance (dynamic polymorphism, to be exact) to work properly has gone away. The `init_t` class is now empty because it doesn't have any state to remember. The `running_t` and `finished_t` classes only define their state and nothing else.

Figure 9.3. When counting the number of words in a web page, we have three main states. Before we have the web page, we can not do anything. When we get the web page, we can start counting. While counting, we need to store the current count and the handler to the web page. When the counting is finished, we no longer need the handle to the web page, we can just store the final count.



Now the main program can just have a `std::variant` value that can hold any of these three types:

Listing 9.4. The main program using `std::variant`

```
class program_t {
public:
    program_t()
        : m_state(init_t())           ①
    {
    }

    // ...

    void counting_finished()
    {
        auto* state = std::get_if<running_t>(&m_state);  ②

        assert(state != nullptr);                         ②

        m_state = finished_t(state->count());            ③
    }

private:
    std::variant<init_t, running_t, state_t> m_state;
};
```

- ① Initially, the state will be an instance of `init_t`
- ② If we want to check whether we have a value of specified type inside of a `std::variant`, we can use the `std::get_if` function. It will return a `nullptr` if the variant doesn't hold the value of specified type.
- ③ Changing the current state is as easy as assigning the new value to the variable

One thing worth noting is that we initialized the `m_state` from a value of type `init_t`. We did not pass it a pointer (`new init_t()`) like we did in the implementation that used inheritance. The `std::variant` is not based on dynamic polymorphism — it doesn't store pointers to objects allocated on the heap. It stores the actual object in its own memory space just like ordinary unions. The difference is that it automatically handles construction and destruction of the objects stored within, and it knows exactly the type of the object it contains at any given time.

When accessing the value stored inside of a `std::variant`, we can use `std::get` or `std::get_if`. Both of them can access the elements of the variant either by type or by index of the type in the variant's declaration:

```
std::get<running_t>(m_state);
std::get_if<1>(&m_state);
```

The difference is that `std::get` either returns a value, or throws an exception if the

variant doesn't hold a value of the specified type, while `std::get_if` returns a pointer to the contained value, or a null pointer on error.

The benefits of `std::variant` compared to our first solution are numerous — it requires us to have almost no boiler-plate, the type tags are handled by `std::variant` itself. We also don't need to create an inheritance hierarchy where all the summed types have to inherit from some super-type — we can sum already existing types like strings, vectors etc. Also, `std::variant` doesn't perform any dynamic memory allocations — the variant instance will, like a regular union, have the size of the largest type we want to store within (plus a few bytes for bookkeeping).

The only drawback is that it can't be easily extended — if you want to add a new type into the sum, you need to change the type of the variant. In the rare cases where extensibility is necessary, we would need to go back to the inheritance-based sum types.

Note

An alternative to the inheritance-based open sum types is the `std::any` class. It is a type-safe container for values of any type. While it is useful in some circumstances, it isn't as efficient as `std::variant`, and shouldn't be used as a simple-to-type replacement for `std::variant`.

9.1.3 Implementing specific states

Now that we have the `program_t` class and classes to represent the potential states, we can proceed on to implement the logic.

The question is where should the logic be implemented. We can make the obvious choice and implement everything in `program_t`, but that would mean that in each function we make, we need to check whether we are in the correct state.

For example, if we wanted to implement the function that starts the counting process inside of the `program_t` class, we would first need to check whether the current state is `init_t`, and then proceed to changing the state to `running_t`. We can check this using the `index` member function of `std::variant`. It returns the index of the current type that occupies the variant instance. Since `init_t` is the first type we specified when defining the type of the `m_state` member variable, its index will be zero.

```
void count_words(const std::string& web_page)
{
    assert(m_state.index() == 0);

    m_state = running_t(web_page);

    ... // count the number of words

    counting_finished();
}
```

After we check for the current state, we switch to `running_t`, and start the counting process. The process is synchronous (we will deal with concurrency and asynchronous execution in chapters 10 and 12), and when it finishes, we can process the result.

As can be seen from the snippet above, we will call the `counting_finished` function which should switch to the `finished_t` state. We need to check for the current state in this function as well. The `counting_finished` function should be only called when we are in the `running_t` process. We will need the result of the process, so instead of using `index`, we will use `std::get_if` and assert on its result.

```
void counting_finished()
{
    const auto* state = std::get_if<running_t>(&m_state);

    assert(state != nullptr);

    m_state = finished_t(state->count());
}
```

The state we end up in is `finished_t` which only needs to remember the calculated word count.

Both of these functions deal with state changes, so it might not be a bad idea to have them in `program_t` even if we need to perform the checks that we are in the correct state. What if a function just needs to handle a part of program logic that is only relevant to a particular state?

For example, the `running_t` state needs to open a stream through which to fetch the contents of a web page. It needs to read that stream word by word, and count the numbers of words in it. It does not change the main program states. It does not even care what are other states in the program — it can do its work only with that data it contains.

For this reason, there is no point in having any part of this logic in the `program_t` class — it can be in `running_t` itself.

```
class running_t {
public:
    running_t(const std::string& url)
        : m_web_page(url)
    {}

    void count_words()
    {
        m_count = std::distance(
            std::istream_iterator<std::string>(m_web_page),
            std::istream_iterator<std::string>());
    }
}
```

```

}

unsigned count() const
{
    return m_count;
}

private:
    unsigned m_count = 0;
    std::istream m_web_page;
};

```

This is the recommended approach to designing programs with state based on sum types — put the logic that deals with one state inside the object that defines that state, and put the logic that performs state transitions into the main program. A full implementation of this program that works with files instead of web pages is available in the `word-counting-states` example.

An alternative approach is to keep the logic out of the `program_t` class, and put everything into the state classes. This approach removes the need for checks with `get_if` and `index`, since a member function of a class representing a particular state will be called only if we are in that state. The downside is that now the state classes need to perform program state changes, which implies that they need to know about each other thus breaking encapsulation.

9.1.4 Special sum type – optional values

We have already mentioned `std::get_if` which returns a pointer to a value if the value exists, or a null pointer otherwise. The reason why a pointer is used is to allow this special case — denoting the missing value.

Pointers can have a lot of different meanings in different situations and they are bad at communicating what they are used for. When we see a function that returns a pointer, we have a few different options:

- It is a factory function that returns us an object that we will become the owner of;
- It is a function that returns us a pointer to an already existing object that we don't own;
- It is a function that can fail, and communicates the failure by returning the null pointer as the result.

In order to be able to tell which of these is the case with a function like `std::get_if` that we want to call, we need to check its documentation.

Instead, it is often better to use types that clearly communicate what the function result is. In the first case, it would be better to replace the pointer with `std::unique_ptr`. In the second case, a `std::shared_ptr` (or `std::weak_ptr`) will do.

The third case is special. There is no reason why it needs to be a pointer. The only reason why it is a pointer instead of an ordinary value is to *extend* the original type with a "no value present" state. That is, to denote that the value is optional — it can exist, but it can be undefined.

In other words, we can say that an optional value of some type T is either a value of type T , or it is empty. So, it is a sum of all values that T can have, and a single value to denote that the value doesn't exist. This sum type could be easily implemented with `std::variant`:

```
struct nothing_t {};  
  
template <typename T>  
using optional = std::variant<nothing_t, T>;
```

Now when we see a function that returns `optional<T>`, we immediately know that we are getting either a value of type T , or *nothing*. We don't have to think about the lifetime of the result, whether we should destroy it or not. We can't forget to check whether it is null or not like it is the case with pointers.

Optional values are useful enough that the standard library provides us with `std::optional` — a more convenient implementation than the one we created by aliasing `std::variant`.

Optional values in the standard library

`std::optional`, just like `std::variant`, has been introduced with C++17. If you have an older compiler, you can use `boost::optional` instead.

Since `std::optional` is a more specific sum type than `std::variant`, it also comes with a more convenient API. It provides member functions to check whether it contains a value, `operator→` to access the value if it exists etc.

So, we could implement our own `get_if` function. We will use `std::get_if` and check whether the result is valid or if it is a null pointer. If it points to a value, we will return that value wrapped inside of an `std::optional`, and we will return an empty optional object if it is null:

```
template <typename T, template Variant>
std::optional<T> get_if(const Variant& variant)
{
    T* ptr = std::get_if<T>(&variant);

    if (ptr) {
        return *ptr;
    } else {
        return std::optional<T>();
    }
}
```

We can use the same approach to wrap any function that uses pointers to denote the missing values. Now that we have a variant of `get_if` that returns an optional, we can reimplement the `counting_finished` function easily:

```
void counting_finished()
{
    auto state = get_if<running_t>(&m_state);

    assert(state.has_value());

    m_state = finished_t(state->count());
}
```

The code looks mostly the same as the one that used pointers. The most obvious difference is that we had `state != nullptr` in the assert statement, and now we have `state.has_value()`. We could make both just `assert` on `state` since both pointers and optionals can be converted to `bool` — `true` if they have a valid value, `false` if they are empty.

The main difference between these two approaches is more subtle. The `std::optional` instance is a proper value and it owns its data. In the example that used a pointer, we would get an undefined behaviour if `m_state` was destroyed or changed to contain another `state` type between the call to `std::get_if` and `state->count()`. In this example, the `optional` object contains the copy of the value, so we can't have that problem. In this example, we don't need to think about the lifetime of any variable, we can just rely on the default behaviour that the C++ language provides.

9.1.5 Sum types for error handling

The optional values can be useful to denote whether an error has occurred or not. For example, we implemented `get_if` function which returns either a value if all is well, or returns nothing — an empty optional — if we tried to get the instance of a type that is not currently stored in the `variant`. The problem in this use-case is that optional values on the value, and give us no information about the error when the value is not present.

If we want to track the errors as well, we can create a sum type which will either contain a value, or an error. The error can be an integer error code, or some more intricate structure, even a pointer to an exception (`std::exception_ptr`).

So, we need a sum type of `T` and `E`, where `T` is the type of the value we want to return, and `E` is the error type. As with `optional`, we can simply define it as `std::variant<T, E>`, but that will not give us a nice API. We would need to use functions like `index`, `std::get` or `std::get_if` to reach the value or the error which is not really convenient. It would be nicer if we had a class dedicated exactly to this use-case, which will have better named functions like `value` and `error`.

For this, we will roll out our own implementation called `expected<T, E>`. When a function returns a value of this type, it will clearly communicate that we expect a value of type `T`, but that we can also get an error of type `E`.

Internally, the class will be a simple tagged union. We will have a flag to denote whether we currently hold a value or an error, and we will have a union of `T` and `E` types:

Listing 9.5. Internal structure of `expected<T, E>`

```
template<typename T, typename E>
class expected {
private:
    union {
        T m_value;
        E m_error;
    };

    bool m_valid;
};
```

The easy part of the implementation are the getter functions. If we need to return a value, but we have an error, we can simply throw an exception, and vice-versa.

Listing 9.6. Getter functions for `expected<T, E>`

```
template<typename T, typename E>
class expected {
// ...

T& get()
{
    if (!m_valid) {
        throw std::logic_error("Missing a value");
    }

    return m_value;
}

E& error()
{
    if (m_valid) {
        throw std::logic_error("There is no error");
    }

    return m_error;
};
};
```

The `const` variants of the getter functions would be the same. They would just

return values instead of references.

The complex part is handling the values inside the union. Since we have union of potentially complex types, we will need to manually call constructors and destructors when we want to initialize or deinitialize them.

Constructing a new value of `expected<T, E>` can be done from a value of type `T`, or from an error of type `E`. Since these two types can be the same, we will make dedicated functions for each of them instead of using ordinary constructors. This isn't mandatory to do, but it will make the code cleaner.

Listing 9.7. Constructing values of `expected<T, E>`

```
template<typename T, typename E>
class expected {
    // ...

    template <typename... Args>
    static expected success(Args&&... params)
    {
        expected result;                                1
        result.m_valid = true;                          2
        new (&result.m_value)                           3
            T(std::forward<Args>(params)...);          3
        return result;
    }

    template <typename... Args>
    static expected error(Args&&... params)
    {
        expected result;                                4
        result.m_valid = false;                         4
        new (&result.m_error)                           4
            E(std::forward<Args>(params)...);          4
        return result;
    }
};
```

- 1 The default constructor creates an uninitialized union
- 2 We are initializing the union tag — we will have a valid value inside
- 3 Calling the placement `new` to initialize the value of type `T` in the memory location of `m_value`
- 4 Creating the error instance is the same, apart from calling the constructor for the type `E` instead of the constructor of `T`

Now if we have a function that can fail, we can simply call `success` or `error` when returning the result.

Placement new

Unlike the regular `new` syntax which allocates memory for a value and initializes that value (calls the constructor), the *placement new* syntax allows us to use already allocated memory and construct the object inside it.

In the case of `expected<T, E>`, the memory is preallocated by the union member variable we defined.

While this is not something that should be often used, it is necessary if we want to implement sum types that don't perform any dynamic memory allocation at runtime.

In order to properly handle the lifetime of an `expected<T, E>` instance and the values stored in it, we need to create the destructor and the copy and move constructors, along with the assignment operator.

The destructor just needs to call the destructor for `m_value` or for the `m_error` depending on which of those we are currently holding:

```
~expected() {
    if (m_valid) {
        m_value.~T();
    } else {
        m_error.~E();
    }
}
```

The copy and move constructors are similar. We need to check whether the instance that we are copying or moving from is valid or not, and initialize the proper member of the union.

Listing 9.8. The copy and move constructors for `expected<T, E>`

```
expected(const expected& other)           1
    : m_valid(other.m_valid)               1
{
    if (m_valid) {
        new (&m_value) T(other.m_value);   2
    } else {
        new (&m_error) E(other.m_error);   3
    }
}

expected(expected&& other)                4
    : m_valid(other.m_valid)               4
{
    if (m_valid) {
        new (&m_value) T(std::move(other.m_value)); 5
    } else {
        new (&m_error) E(std::move(other.m_error)); 5
    }
}
```

```
    }
}
```

- ➊ The copy constructor first initializes the flag which denotes whether we have a value or an error
- ➋ If the instance we are copying from contains a value, we will call the copy constructor on that value to initialize our `m_value` member variable using the placement new syntax
- ➌ If the original instance contained an error, we are copying said error into our instance
- ➍ The move constructor behaves just like the copy constructor, but we are allowed to *steal* the data from the original instance
- ➎ Instead of calling the copy constructor for `m_value` or for `m_error`, we can call the move constructor

These were quite straight-forward. The biggest problem to implement is the assignment operator. It needs to work for four different cases:

- Both the instance we are copying to and the instance that we are copying from contain valid values
- Both instances contain errors
- `this` contains an error, while `other` contains a value, and
- `this` contains a value, and `other` contains an error.

As it is customary, we are going to implement the assignment operator using the copy-and-swap idiom which means we will need to create a `swap` function for `expected<T, E>` class.

Copy-and-swap idiom

In order to have a strongly exception-safe class which can't leak any resources when an exception occurs, and which can guarantee that an operation is either successfully completed, or it isn't completed at all, we usually use the copy-and-swap idiom when implementing the assignment operator.

In a nutshell, we create a temporary copy of the original object, and then swap our internal data with it. If all went well, when the temporary object gets destroyed, it will destroy our previous data with it.

If an exception is thrown, we will never take the data from the temporary object, which will leave our instance unchanged.

For more information, check out www.gotw.ca/gotw/059.htm and stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom

Listing 9.9. Swap function for expected<T, E>

```
void swap(expected& other)
{
    using std::swap;
    if (m_valid) {
        if (other.m_valid) {
            swap(m_value, other.m_value);
            swap(m_error, other.m_error);
        } else {
            m_value = other.m_value;
            m_error = other.m_error;
        }
    } else {
        if (other.m_valid) {
            m_value = other.m_value;
            m_error = other.m_error;
        }
    }
}
```

```

        swap(m_value, other.m_value);           ①

    } else {
        auto temp = std::move(other.m_error);
        other.m_error.^E();
        new (&other.m_value) T(std::move(m_value));
        m_value.^T();
        new (&m_error) E(std::move(temp));
        std::swap(m_valid, other.m_valid);
    }
} else {
    if (other.m_valid) {
        other.swap(*this);                  ③

    } else {
        swap(m_error, other.m_error);      ④
    }
}
}

expected& operator=(expected other)          ⑤
{
    swap(other);
    return *this;
}

```

- ① If other contains a value and so do we, then just swap them
- ② If we are valid, but other contains an error, we need to store the error in a temporary variable, so that we can then move our value into other. Then we can safely set the error for our instance
- ③ In the case that this contains an error, while other is valid, we can base the implementation on the previous case
- ④ If both instances contain an error, just swap the error values
- ⑤ The assignment operator is trivial — the other argument will contain a copy of the value we want assigned to our instance (note that we are getting other by-value, not as a const-reference as usual), and then we just swap it with our instance

Having implemented this, the rest is easy. We can create a casting operator to `bool` which will return true if we have a valid value, and false otherwise. Also, we can create a casting operator to convert the instance to `std::optional` so that we can use `expected<T, E>` with the code that uses `std::optional`.

```

operator bool() const
{
    return m_valid;
}

operator std::optional<T>() const
{
    if (m_valid)
        return m_value;
    } else {

```

```

        return std::optional<T>();
    }
}

```

Now we can use the `expected` type as a drop-in replacement of `std::optional`. We can redefine the `get_if` function yet again. For the sake of simplicity, we will use `std::string` as the error type (we will see how to use `std::exception_ptr` in the next chapter).

```

template <typename T, template Variant,
          template Expected = expected<T, std::string>>
Expected get_if(const Variant& variant)
{
    T* ptr = std::get_if<T>(variant);

    if (ptr) {
        return Expected::success(*ptr);
    } else {
        return Expected::error("Variant doesn't contain the desired type");
    }
}

```

Now we get a function that either returns us a value, or a detailed error message. This is quite useful to have — either for debugging purposes, or when we need to show the error to the user. Embedding errors in the type is also useful in asynchronous systems because we can easily transfer the errors between different asynchronous processes.

9.2 Domain modelling with the algebraic data types

The main idea when designing data types is to make illegal states not possible to represent. That is why the `size` member function of `std::vector` returns an unsigned value (even if some people don't like unsigned types¹²) — so that the type itself forces our intuition that the size can't be negative, and why algorithms like `std::reduce` (which we talked about in chapter 2) take proper types to denote the execution policy instead of ordinary integer flags.

We should do the same with the types and functions that we create. Instead of thinking about which data we need in order to cover all the possible states we can be in, and place them in a class, we need to think about how to define the data to cover **only** the states we can be in.

We are going to demonstrate this using the new scenario - the *Tennis kata* (codingdojo.org/kata/Tennis/). The aim is to implement a simple tennis game. In tennis, we have two players (pretending that doubles do not exist) that play against each other. Whenever a player is not able to return the ball to the other

¹² Danger — unsigned types used here — critical.eschertech.com/2010/04/07/danger-unsigned-types-used-here/

player's court, he loses the ball and the score gets updated.

The scoring system in tennis is a bit unique, but rather simple:

- Allowed scores are 0, 15, 30 and 40;
- If a player has 40 points, and wins the ball, he wins the game;
- If both players have 40, then the rules become a bit different — the game is in deuce;
- When in deuce, the player that wins the ball has *advantage*;
- If the player wins the ball while in advantage, he wins the game, if he loses, the game returns to deuce.

9.2.1 The naive approach, and where it falls short

The naive approach to solving this problem is to create two integer scores so that we can track how many points each player has. We can use a special value to denote when a player has advantage.

```
class tennis_t {
private:
    int player_1_points;
    int player_2_points;
};
```

It does cover all the possible states, but the problem is that it allows us to set an invalid number of points for a player. Usually, we solve this by verifying the data that we get in the setter, but it would be more useful if we didn't need to verify anything — if the types forced us to have the correct code.

The next step would be to replace the number of points with an enum value which would allow only to set the valid number of points:

```
class tennis_t {
    enum class Points {
        love, // zero points
        fifteen,
        thirty,
        forty
    };

    Points player_1_points;
    Points player_2_points;
};
```

This significantly lowers the number of states we can be in, but it still has problems. The first problem is that it allows us to have both players with 40 points (which is technically not allowed — that state has a special name), and we have no way to represent advantage. We could add deuce and advantage to the enum, but that would just create new invalid states (one player could be in deuce while the

other one has zero points).

This isn't a good approach for solving this problem. Instead, we should go top-down and try to split the original game into disjunctive states, and then define those states.

9.2.2 A more sophisticated approach: top-down design

From the rules, we see that there are two main states of the game — the state where the scoring is numeric, and the state where we are in the deuce or advantage.

The state where we have normal scoring keeps the scores of both players at the same time. Unfortunately, the things aren't that simple. If we used the previously defined `points` enum, we would have the possibility of both players having forty points which isn't allowed, and should be represented as the deuce state.

We can try to solve this by just removing the `forty` enum value, but we will then lose the ability to have a forty-something score. Back to the drawing board. The *normal scoring state* isn't just a single state. We can have two players with scores up to thirty, or we can have one player with forty points, and another player up to thirty.

```
class tennis_t {
    enum class points {
        love,
        fifteen,
        thirty
    };

    enum class player {
        player_1,
        player_2
    };

    struct normal_scoring {
        points player_1_points;
        points player_2_points;
    };

    struct forty_scoring {
        player leading_player;
        points other_player_scores;
    };
};
```

We have now covered all the *regular* scoring states. We are now left with the deuce and advantage states. Again, we can consider these as clearly different states, and not as a single one. The deuce state doesn't need to hold any values, while the advantage state should know which player has the advantage.

```
class tennis_t {
```

```
// ...

    struct deuce {};

    struct advantage {
        player player_with_advantage;
    };
};
```

Now that we have covered all the possible states, we just need to define the sum type of all of them:

```
class tennis_t {
    // ...

    std::variant
        < normal_scoring
        , forty_scoring
        , deuce
        , advantage
    > m_state;
};
```

We have covered all the possible states that the tennis game can be in and we can't have a single illegal state.

Note

Title

There is one state potentially missing — the *game over* state which should denote which player won.

If we just want to print out the winner, and terminate the program, then we do not need to add that state to it.

Otherwise, if we wanted the program to continue running, we would need to implement that state as well. It would be trivial to do — we would just need to create another structure with a single member variable to store the winner, and expand the `m_state` variant.

As usual, it isn't always worth it to go this far in removing the invalid states (we might have decided just to deal with the forty-forty case manually), but the main point of this example is to demonstrate the process of designing algebraic data types that match the domain we want to model. We start by splitting the original state space into smaller independent sub-parts, then describe those independent parts individually.

9.3 Better handling of algebraic data types with pattern matching

The main issue when implementing programs with optional values, variants and other is that every time we need a value, we need to check whether it is present or not, and extract it from its wrapper type. We would need the equivalent checks

even if we created the state class without sum types, but only when we set the values, not every time we want to access them.

This can be tedious, so many functional programming languages provide us with a special syntax which makes this easier. Usually, this syntax looks like a switch-case statement on steroids which isn't only able to match against specific values, but also on types, and more complex patterns.

Now, imagine that we created an enum in our tennis scenario to denote the state we are in. It would be a common thing to see a `switch` statement like this somewhere in the code:

```
switch (state) {
    case normal_score_state:
        ...
        break;

    case forty_scoring_state:
        ...
        break;

    ...
};
```

Depending the value of the `state` variable, it will execute a specific case label. But sadly, it works only on integer-based types.

Now imagine a world in which this could also test strings, if it could work with variants and execute different cases based on the type of the variable contained in the variant. And even more, what it would be like if each case could be a combination of a type check, value check, and a custom predicate that we could define. This is what pattern matching in most functional programming languages look like.

C++ provides ways to do pattern matching for template meta-programming (as we will see in chapter 11), but not for normal programs, so we will need to find another way to do it.

The standard library provides us with a `std::visit` function that takes an instance of a `std::variant` and a function that should be executed on the value stored inside. For example, if we wanted to print out the current state in our tennis game (given we implemented the operator `<<` to write the state types to the standard output), we could just do:

```
std::visit([] (const auto& value) {
    std::cout << value << std::endl;
},
m_state);
```

We are passing a generic lambda (a lambda with an argument type specified

as `auto`) so that it can work on values of different types, since our variant can have four completely different types, and all of this is still statically typed.

Using a generic lambda with `std::visit` is often useful, but it isn't sufficient in most cases. We want to be able to execute different code based on which value we have stored in the variant — just like we can do with the `case` statement.

One way that we can do this is by creating an overloaded function object which will separate implementations for different types, and the correct one will be called based on the type of the value stored inside the variant instance. In order to make this as short as possible, we will use the language features available in C++17. The implementation compatible with older compilers is available in the accompanying code examples.

```
template <typename... Ts>
struct overloaded : Ts... { using Ts::operator()...; };

template <typename... Ts> overloaded(Ts...) -> overloaded<Ts...>;
```

The `overloaded` template takes a list of function objects, and creates a new function object that presents the call operators of all provided function objects as its own (the `using Ts::operator()...` part).

Note

The code snippet that implements the `overloaded` structure uses the template argument deduction for classes that has been introduced in C++17.

The template argument deduction relies on the constructor of the class to figure out what are the template parameters.

We can either provide a constructor, or we can provide a template deduction guideline like we did above.

For more information, check out the appendix A.

We can now use this in our tennis example. Every time a player wins the ball, the `point_for` member function will be called, and it will update the game state accordingly.

```
void point_for(player which_player)
{
    std::visit(
        overloaded {
            [&](const normal_scoring& state) {
                // Increment the score, or switch the state
            },
            [&](const forty_scoring& state) {
                // Player wins, or we switch to deuce
            },
            [&](const deuce& state) {
                // We switch to advantage state
            },
        },
        state
    );
}
```

```

        [&](const advantage& state) {
            // Player wins, or we go back to deuce
        }
    },
    m_state);
}

```

The `std::visit` will call the overloaded function object, and the object will match the given type against all its overloads and execute the one that is the best match (type-wise). While the syntax is not as pretty, this provides us with an efficient equivalent of the `switch` statement that works on the type of the object stored in a variant.

We can easily create a `visit` function for `std::optional`, for our `expected` class, and even for inheritance-based sum types which would give us a unified syntax to handle all the sum types we created.

9.4 Powerful pattern matching with the Mach7 library

So far, we have had a simple pattern matching on types, and we could easily create matching on specific values by hiding the `if-else` chains behind a structure similar to `overloaded`.

But it would be much more useful to be able to match on more advanced patterns. For example, we might want to have separate handlers for `normal_scoring` when the player has less than thirty points, and when it has thirty points because in that case we need to change the game state to `forty_scoring`.

Unfortunately, C++ doesn't have a syntax that allows this. But there is a library called `Mach7` that allows us to write more advanced patterns, though with a bit awkward syntax.

The Mach7 library for efficient pattern matching

The `Mach7` library (github.com/solodon4/Mach7) is created by Yuriy Solodkyy, Gabriel Dos Reis and Bjarne Stroustrup, and serves as an experiment that will eventually be used as the base for extending C++ to support pattern matching.

While the library started as an experiment, it is considered stable for general use. And it is generally more efficient than the visitor pattern (not to be confused with `std::visit` for variants). The main downside of `Mach7` is its syntax.

The `Mach7` library allows us to specify the object we want to match against, and then list all the patterns to use for matching, and actions to perform if the pattern is matched. In our tennis game scenario, the implementation of the `point_for` member function would look like this:

```
void point_for(player which_player)
```

```

{
    Match(m_state)
    {
        Case(C<normal_scoring>())
            // Increment the score, or switch the state

        Case(C<forty_scoring>())
            // Player wins, or we switch to deuce

        Case(C<deuce>())
            // We switch to advantage state

        Case(C<advantage>())
            // Player wins, or we go back to deuce
    }
    EndMatch
}

```

Now we might want to split the second pattern into several separate ones. If the player who won the ball had 40 before, it means that he has won the game. Otherwise, we need to see whether we can increase the points of the second player, or we need to switch to the deuce state.

So, if we are currently in `forty_scoring` state, and the player which has the 40 points has won the ball, he wins the game regardless of how many points the other player has. We can denote this with `C<forty_scoring>(which_player, _)` pattern. The underscore means that we wo not care about a value when matching — in this case, we do not care about the number of points that the other player has.

If the ball was not won by the player who had 40 points, we want to check whether the other player had 30 points so that we could switch to the deuce state. We can match this with the `C<forty_scoring>(_, 30)` pattern. We do not need to match agains a particular player because we know that if the player who previously had 40 points had won the ball, we would match it with the previous pattern.

If neither of these two patterns matched, we just need to increase the number of points for the second player. We can just check that we are in `forty_scoring` state.

Listing 9.10. Matching on deconstructed types

```

void point_for(player which_player)
{
    Match(m_state)
    {
        ...
        Case(C<forty_scoring>(which_player, _))      ①
            // Player wins

        Case(C<forty_scoring>(_, 30))                ②
    }
}

```

```

    // Switch to deuce

    Case(C<forty_scoring>())
        // Increment the points for the player
        ...
    }
EndMatch
}

```

- ➊ If the player who already had 40 points has won the ball, he wins the game, we don't even need to consider the number of points the other player has
- ➋ If the ball was won by the player which didn't have 40 points (the previous case wasn't a match), and the current number of points for the other player is 30, the game is in deuce
- ➌ If none of the previous cases was a match, this means that we just need to increase the number of points for the player

No matter which path you take when handling programs with algebraic types — `std::visit` with overloaded function objects, or with full pattern matching with a library like Mach7, you will have to write more correct programs. The compiler will force you to write exhaustive patterns or the program will not compile. And the space of possible states will be kept as minimal as possible.

9.5 Summary

- Implementing program state through algebraic data types requires some thinking, and it produces longer code, but it allows us to minimize the number of states our program can be in and removes the possibility to have invalid states;
- Inheritance, dynamic dispatch, and the visitor pattern are often used in OO programming languages to implement sum types. The main problem with using inheritance to implement sum types is that it incurs runtime performance penalties;
- If we know exactly which types we want to sum, variants are a better choice for implementing sum types than inheritance. The main downside of `std::variant` is that it incurs a lot of boiler-plate because of the `std::visit` function;
- Unlike exceptions which should denote, as the name implies, exceptional situations that our program can find itself in, optional values and the `expected` class are perfect for explicitly stating the possibility of failures. They also travel across multiple threads, multiple processes more easily;
- The idea of having a type that contains either a value, or an error has been popular in the functional programming world for a long time. It became popular in the C++ community after the talk by Andrei Alexandrescu at the C++ and Beyond conference called "Systematic Error Handling in C++" where he presented his version of the `expected` type. That version of `expected` was similar to ours, but it only supported `std::exception_ptr` as the error type.

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch09

10

Monads

This chapter covers:

- What is a functor?
- Going one step further from `transform` with monads
- Composing functions that return wrapper types
- How to handle asynchronous operations in FP style

The functional programming world is not big on design patterns, but there are some common abstractions that pop up everywhere. These abstractions allow us to handle different types or problems from significantly different domains in the same way.

In C++, we already have one abstraction like this — the iterators. With ordinary arrays, we can use pointers to move around and access the data — we can just use the operators `++` and `--` to go to the next and previous elements, and we can dereference them with the `*` operator. The problem is that this works only for arrays and structures that keep their data in a contiguous chunk of memory. It does not work for data structures like linked lists, sets and maps implemented using trees etc.

For this, the iterators were created. They leverage operator overloading to create an abstraction that has the same API as pointers, which can work not only for arrays but for various different data structures. It also works for the things like input and output streams which are not traditionally considered to be *data structures*.

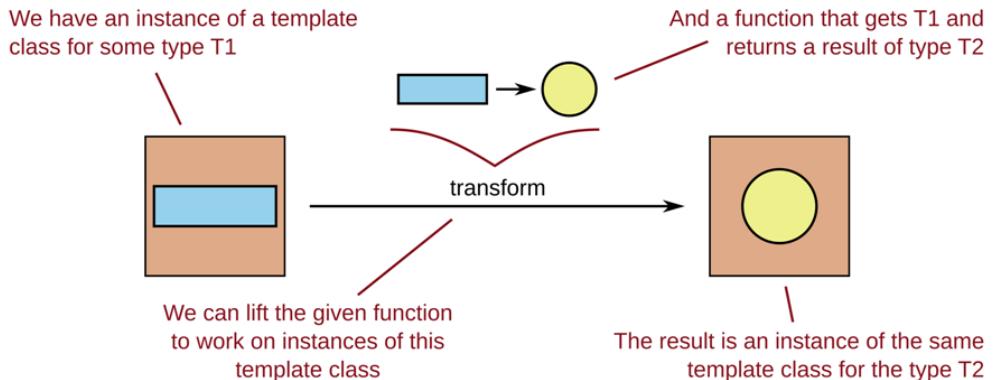
Now, in chapter 7 we have seen another abstraction that builds on top of iterators

— the ranges — which go one step further than iterators by abstracting over different data structures instead of only abstracting the data access in those structures. We will see ranges pop up in this chapter as well.

10.1 Not your father's functors

We are going to break a bit from the usual form of this book, and instead of going example-first, we will first define a concept, and then see the examples.

Figure 10.1. Functor is a template class over some type T that we can lift any function operating on T to be able work on instances of that template class.

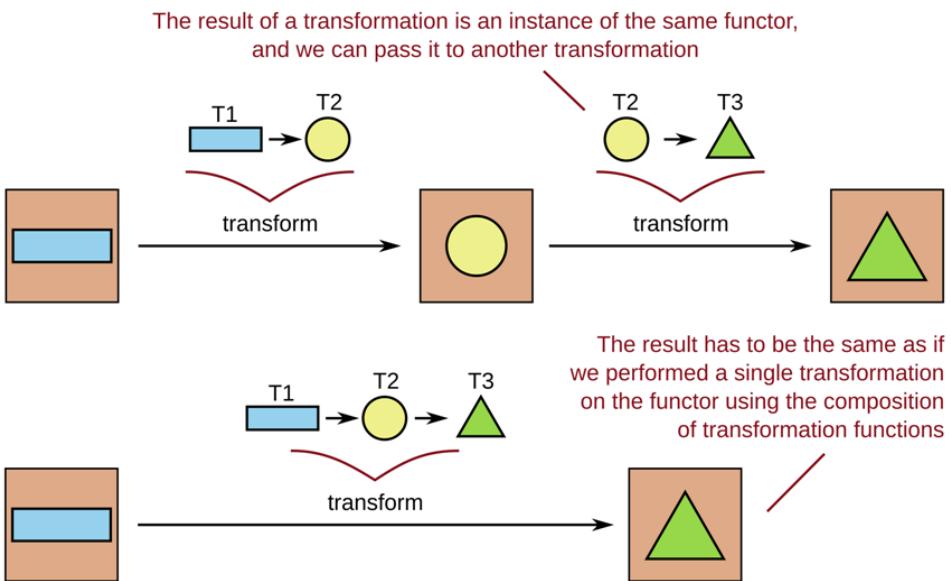


We are going to start with a *functor*. As we've mentioned in chapter 3, a lot of C++ developers use this word to mean "*a class with the call operator*", but this is a misnomer, and when we talk about functors in FP, we are talking about something quite different.

Functor is a concept coming from a really abstract branch of mathematics called *category theory*, and its formal definition sounds as abstract as the theory it is coming from. We will define it in a way that should be more intuitive to C++ developers.

A template class F is a functor if it has a `transform` (or `map`) function defined on it. The `transform` function takes an instance f of a type $F<T_1>$, and a function $t: T_1 \rightarrow T_2$, and it returns a value of type $F<T_2>$. This function can have multiple different forms, so we will use the pipe notation we have seen in the chapter 7 for clarity.

Figure 10.2. The main rule the transform function needs to obey is that composing two transforms — one that lifts the function f and another that lifts the function g needs to have the same effect as having a single transformation with the composition of f and g



The transform function needs to obey the following two rules:

- Transforming a functor instance with an identity function returns the same (as in *equal to*) functor instance:

```
f | transform([](auto value) { return value; }) == f
```

- Transforming a functor with one function, and then with another is the same as transforming the functor with the composition of those two functions:

```
f | transform(t1) | transform(t2) ==
f | transform([]=)(auto value) { return t2(t1(value)); }
```

This looks very much like what the `std::transform` algorithm and the `view::transform` from the ranges library do. That is not accidental — generic collections from the STL and ranges *are* functors. They are all wrapper types that have a well-behaved `transform` function defined on them.

It is important to note that the other direction does not hold — not all functors are collections or ranges.

10.1.1 Handling optional values

One of the basic functors is the `std::optional` type from the previous chapter. It just needs to have a `transform` function defined:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/functional-programming-in-cplusplus>

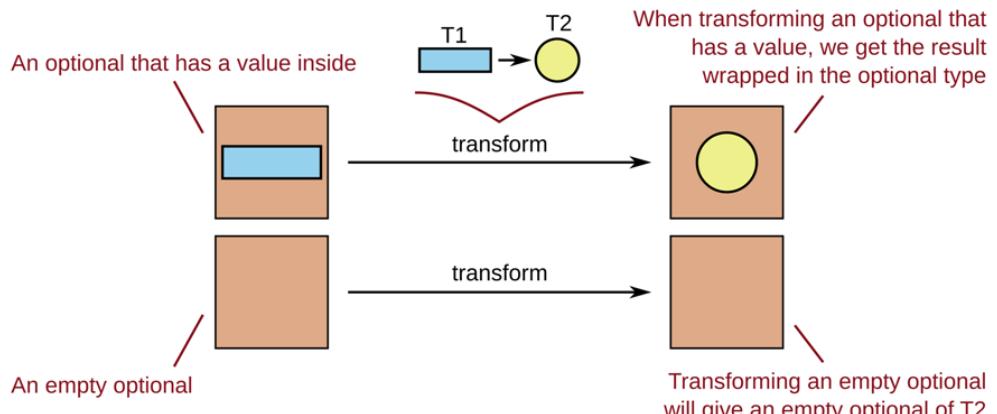
Listing 10.1. Defining the transform function for std::optional

```
template <typename T1, typename F>
auto transform(const std::optional<T1>& opt, F f)
    -> decltype(std::make_optional(f(opt.value())))) ①
{
    if (opt) {
        return std::make_optional(f(opt.value())); ②
    } else {
        return {}; ③
    }
}
```

- ① We need to specify the return type since we are returning just {} in the case when we have no value
- ② If opt contains a value, transform it using the function f and return the transformed value inside a new instance of std::optional
- ③ If we have no value, return an empty instance of std::optional

Alternatively, we can create a range view that will give us a range of one element when the std::optional contains a value, and an empty range otherwise. This will allow us to use the pipe syntax (you should check out the `functors-optional` code example where we define the `as_range` function which converts a std::optional to a range of at most one element).

Figure 10.3. Optional is a wrapper type that can contain a single value, or it can contain nothing. If we transform an optional that contains a value, we will get an optional containing the transformed value. If the optional did not contain anything, we will get an empty optional as the result.



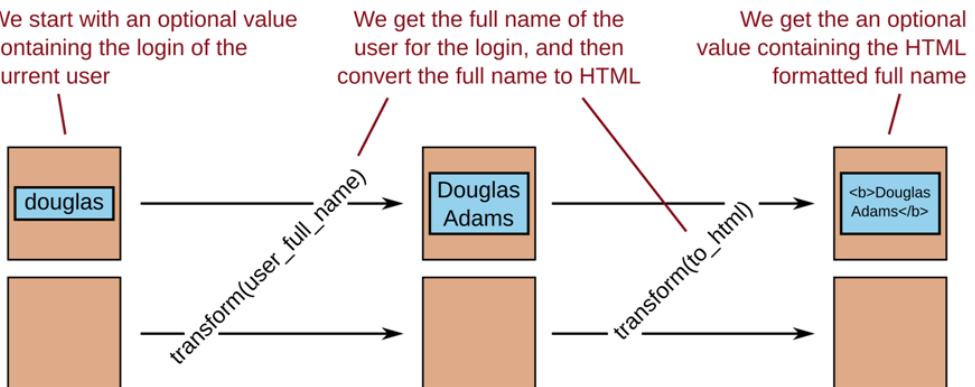
What is the benefit of using the `transform` function compared to handling the missing values manually with an if-else statement?

Consider the following scenario — we have a system that manages user logins. We can have two states — user is either logged in or not. It is natural to represent this with a `current_login` variable of `std::optional<std::string>` type — it will be empty if the user is not logged in, whereas it will contain the username otherwise. We will make the `current_login` variable a global one to simplify the code examples.

Now imagine we have a function that retrieves the full name of the user, and a function that creates a HTML formatted string of anything we pass to it.

```
std::string user_full_name(const std::string& login);
std::string to_html(const std::string& text);
```

Figure 10.4. We can apply a chain of lifted functions to an optional value. In the end, we will get an optional object containing the result of all transformations composed.



If we wanted to get the HTML formatted string of the current user, we would either always need to check whether there is a current user, or we could create a function that returns a `std::optional<std::string>` — it would return an empty value if no user is logged in, and it would return the formatted full name in the case there is. This function is trivial to implement now that we have a `transform` function that works on optional values:

```
transform(
    transform(
        current_login,
        user_full_name),
    to_html);
```

Alternatively, if we wanted to return a range, we could just perform the transformations using the pipe syntax:

```
auto login_as_range = as_range(current_login);
login_as_range | view::transform(user_full_name)
    | view::transform(to_html);
```

If we look at these two implementations, one thing pops out — there is nothing that says that this code works on optional values. It can work for arrays, vectors, lists or anything else that has a `transform` function defined for it. This means that we will not need to change the code if we decided to replace `std::optional` with any other functor.

Peculiarity of ranges

It is important to note that there is no automatic conversion from a `std::optional` to a range and vice-versa, so we need to do this manually.

This means that, strictly speaking, the `view::transform` function is not properly defined for it to make something a functor. It always returns a range, and not the same type we passed to it.

This can be problematic since it will force us to convert the types manually, but it can be considered a minor nuisance when we consider the benefits that ranges bring us.

Imagine we wanted to create a function that takes a list of usernames, and gives us a list of formatted full names — the implementation of said function would be identical to the one that works on optional values. The same goes for a function that uses `expected<T, E>` instead of `std::optional<T>`.

This is the main power that widely applicable abstractions like functors bring us — we can write generic code that works unchanged in various scenarios.

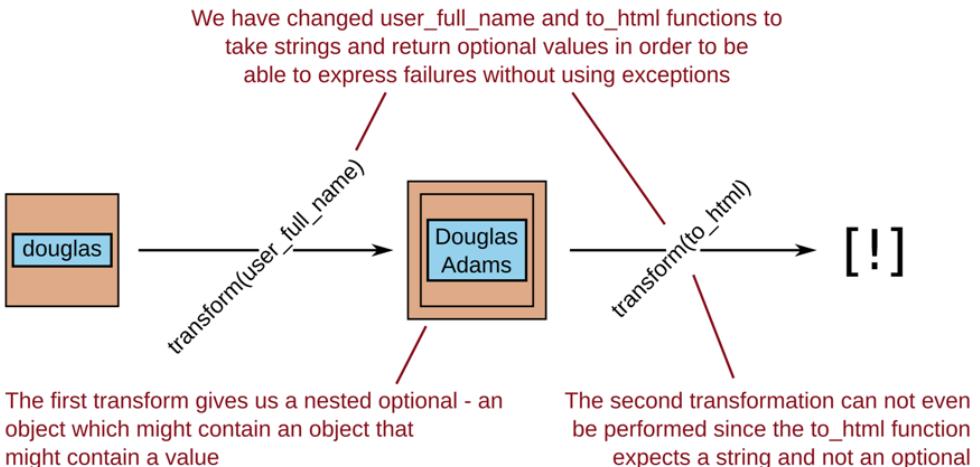
10.2 Monads – more power to the functors

Functors allow us to easily handle transformations of wrapped values, but they have a serious limitation. Imagine that our functions `user_full_name` and `to_html` can fail. And instead of returning strings, they returned `std::optional<std::string>`:

```
std::optional<std::string> user_full_name(const std::string& login);
std::optional<std::string> to_html(const std::string& text);
```

The `transform` function will not help us much in this case. If we tried to use it and write the same code we wrote in the previous example, we would get quite a complicated result. As a reminder, `transform` received an instance of a functor `F<T1>`, and a function from `T1` to `T2`, and it returned us an instance of `F<T2>`.

Figure 10.5. If we try to compose multiple functions that take a value and return an instance of a functor, we will start getting nested functors – in this case, we will get an optional of optional of some type which is mostly useless. What's more, in order to be able to chain two transformations, we would need to lift the second twice.

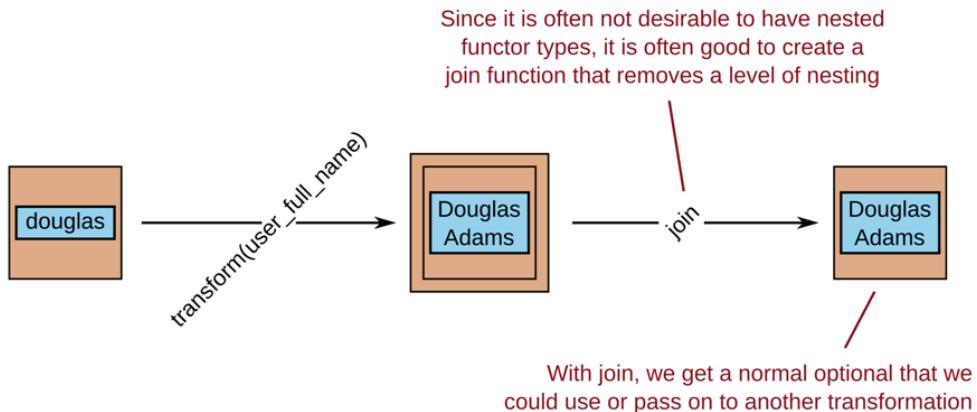


Let's look at the following code snippet:

```
transform(current_login, user_full_name);
```

What is its return type? It is not a `std::optional<std::string>`. The `user_full_name` function takes a string and return an optional value which makes `T2 = std::optional<std::string>`. And that, in turn, makes the result of the `transform` be a nested optional value `std::optional<std::optional<std::string>>`. The more transformations we perform, the more nesting we get, and this is very unpleasant to work with.

Figure 10.6. Transforming a functor with a function that does not return a value, but a new instance of that functor will give us nested functor types. We can create a function that will remove nesting.

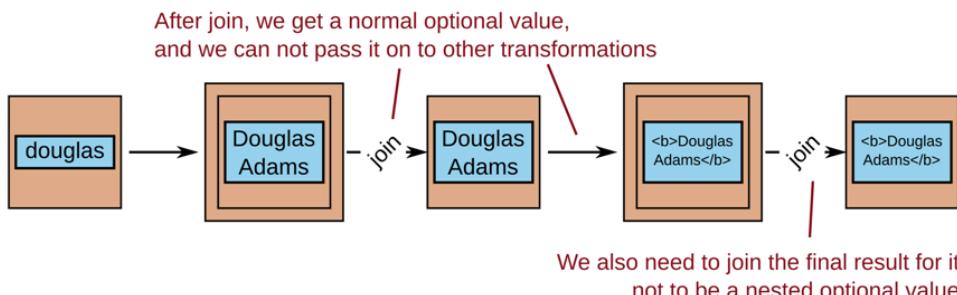


This is where monads come into play. A monad $M<T>$ is a functor that has an additional function defined on it — a function that removes one level of nesting:

```
join: M<M<T>> → M<T>
```

With `join`, we no longer have a problem of using functions that do not return ordinary values, but return monad (functor) instances.

Figure 10.7. We can use optional values to denote errors in computations. With join, we can easily chain multiple transformations that can fail.



We can now write our code like this:

```
join(transform(
    join(transform(
        current_login,
        user_full_name)),
    to_html));
```

Or if you prefer the range notation:

```
auto login_as_range = as_range(current_login);
login_as_range | view::transform(user_full_name)
    | view::join
    | view::transform(to_html)
    | view::join;
```

When we changed the return type of our functions, we made quite an intrusive change. If we implemented everything using if-else checks, we would have had to make significant changes to our code. Here, we just needed to avoid wrapping a value multiple times.

Now, it is obvious that we can simplify this even more. In all of the above transformations we perform `join` on the result. Can we just merge those into a single function?

We can, and in fact, it is a more common way to define monads. We can say that a monad `M` is a wrapper type that has a constructor (a function that constructs an instance of `M<T>` from a value of type `T`), and a `mbind` function (it is usually called just `bind`, but we will use this name so that it does not get confused with `std::bind`) which is a composition of `transform` and `join`.

```
construct : T → M<T>
mbind     : (M<T1>, T1 → M<T2>) → M<T2>
```

It is easy to show that all monads are functors — it is trivial to implement `transform` using `mbind` and `construct`.

As it was the case with functors, we have a few rules here as well. We are going to write them just for the sake of completeness, they are not necessary in order for us to be able to leverage monads in our programs:

- The first rule says that if we have a function `f: T1 → M<T2>`, and a value `a` of type `T1`, wrapping said value into the monad `M` and then binding it with function `f` is the same as just calling the function `f` on it:

```
mbind(construct(a), f) == f(a)
```

- The second rule is the same, just turned around — if we bind a wrapped value to the construction function, we will get the same wrapped value:

```
mbind(m, construct) == m
```

- The third rule is a bit less intuitive. It defines the associativity of the `mbind` operation.

```
mbind(mbind(m, f), g) == mbind(m, [] (auto x) { return mbind(f(x), g) })
```

While these rules might look off-putting, they are just there to precisely define

what is a well-behaving monad. We will just rely on our intuition from now on — a monad is something that we can construct and `mbind` it to a function.

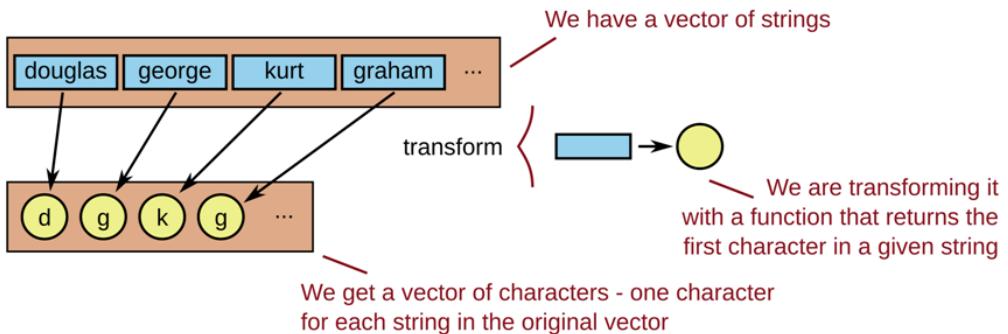
10.3 Basic examples

Let's start with a few simple examples. When learning C++ the right way, we first learn about some basic types, and the first *wrapper type* we see is `std::vector`.

So, let's see how we can create a functor out of it. We have two things to check:

- We said that a functor is a template class,
- And that we need a `transform` function that will be able to take one vector, and a function that can transform its elements, and it will return a vector of transformed elements.

Figure 10.8. Transforming a vector will give us a new vector with the same number of elements. For each element in the original collection, we get an element in the result.



The `std::vector` is a template class, so we are good there, and with ranges, implementing the proper `transform` function is a breeze:

```
template <typename T, typename F>
auto transform(const std::vector<T>& xs, F f)
{
    return xs | view::transform(f) | to_vector;
}
```

We are treating the given vector as a range, and transforming each of its elements with `f`. In order for our function to return a vector, as required by the functor definition, we need to convert the result back to the vector. If we wanted to be a bit more lenient here, we could have just returned a range as the result.

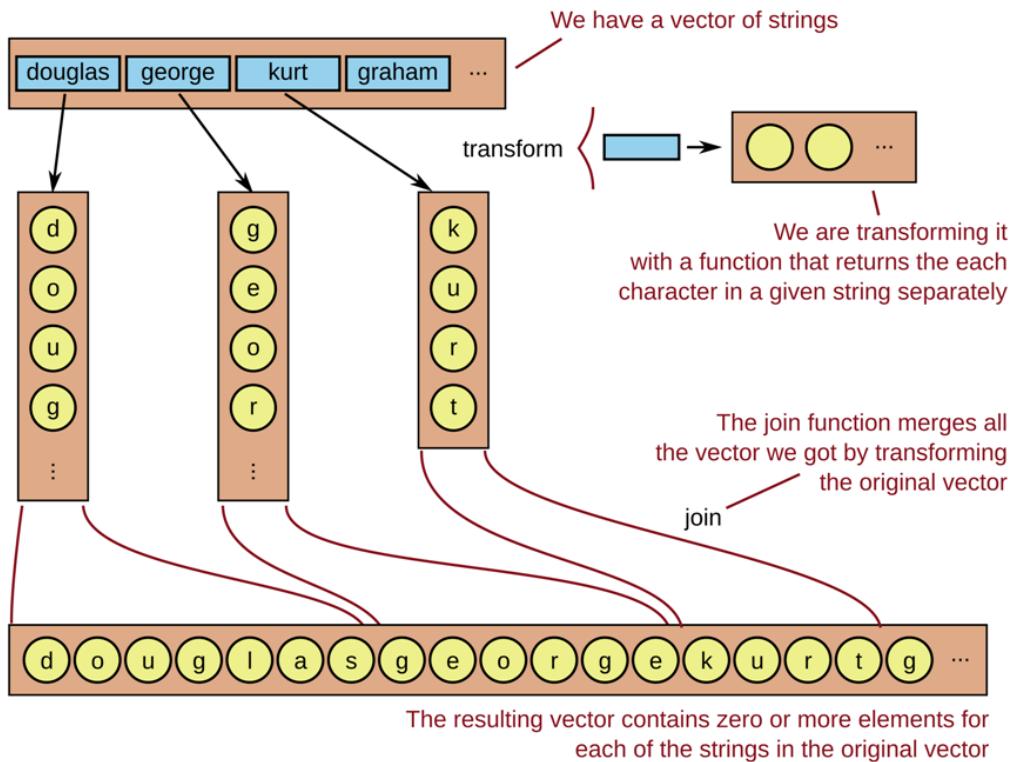
Now that we have a functor, let's turn it into a monad. Here we need the `construct` and `mbind` functions.

The `construct` should take a value and create a vector out of it. For this, the natural thing is to use the actual constructor for the vector. If we wanted to write a proper

function that does it, we could easily whip up something like this:

```
template <typename T>
std::vector<T> make_vector(T&& value)
{
    return {std::forward<T>(value)};
}
```

Figure 10.9. The `mbind` function for vectors applies the transformation function to each element in the original vector. The result of each of those transformations is a vector of elements. All elements from those vectors will be collected into a single resulting vector. This means that unlike `transform`, `mbind` allows us not only to have a single resulting element per input element, but we can have as many as we want.



Now, we are left with `mbind`. In order to implement it, it is always useful to think of `transform` plus `join`. The `mbind` function (contrary to `transform`) wants a function that maps values to instances of the monad — in the case of `std::vector`. This means that for each element in the original vector, it will return a whole new vector instead of just a single element.

Listing 10.2. mbind function for vectors

```
template <typename T, typename F>
auto mbind(const std::vector<T>& xs, F f) ①
{
    auto transformed =
        xs | view::transform(f)          ②
        | to_vector;                  ②

    return transformed
        | view::join                 ③
        | to_vector;                ③
}
```

- ① `f` takes a value of type `T` and returns a vector of `T` or some other type
- ② `transform` will call `f` and give us a range of vectors, which we are converting to a vector of vectors
- ③ We do not want a vector of vectors, we just want all the values in a single vector

We have implemented the `mbind` function for vectors. It is not as efficient as it can be because it saves all intermediary results to temporary vectors, but the main point is just to show that `std::vector` is a monad.

Note Pipe syntax for mbind

We have defined `mbind` to be a function that takes two arguments and returns the result.

For convenience, in the rest of the chapter, we will use the `mbind` function with the pipe syntax because it is more readable. We will write `xs | mbind(f)` instead of `mbind(xs, f)`.

This is not something we can do out-of-the-box — it needs a bit of boiler-plate code as can be seen in the 10-monad-vector and 10-monad-range examples.

10.4 Range and monad comprehensions

The same approach we used for vectors would work for other similar collections like arrays and lists. In essence, all these collections are quite similar on the level of abstraction we are working at — they are flat collections of items of the same type.

We have already seen an abstraction that works over all these types — ranges. As a matter of fact, we did use ranges to implement all previous functions for `std::vector`.

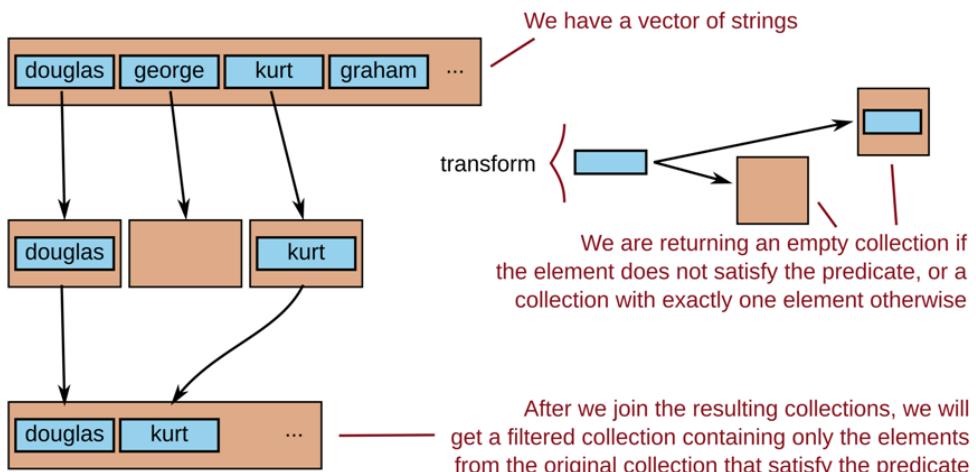
We have already seen multiple times that the `transform` function can be quite useful. And now we got a new function called `mbind` that is similar, but a bit more powerful. The question is whether this additional power is something that is needed at all.

We will see how it benefits us for other monads later, but let's first investigate how useful it can be for normal collections and ranges.

Let's paint a different picture of what `mbind` does that can be more fitting to collection-like structures. And let's start with `transform` since we know how it works. The `transform` function takes a collection, and generates a new one. It does so by traversing the original collection, transforms each element in it, and puts those transformed elements into the new collection.

With `mbind`, the story is similar, but slightly different. We said that it can be seen as a composition of `transform` and `join`. The `transform` function to create a whole range of new elements for each element in the original collection, and `join` to concatenate all those generated ranges. In other words, `mbind` allows us not to generate only a single new element for each element in the original collection, but to generate as many as we want — zero or more.

Figure 10.10. We can easily express filtering with `mbind` by passing it a transformation function that returns an empty collection if the original element does not satisfy the filtering predicate, while it returns a collection containing one element otherwise



When can this be useful? One of the functions that we have also seen a few times before is `filter`. It can easily be implemented in the terms of `mbind`. We just need to give `mbind` a function that will return an empty range if the current element should be filtered out, or a single element range otherwise like so:

Listing 10.3. Filtering in terms of `mbind`

```
template <typename C, typename P>
auto filter(const C& collection, P predicate)
{
    return collection
        | mbind([=](auto element) {
            return view::single(element)
        })
}
```

1

```

        | view::take(predicate(element)
                  ? 1 : 0);
    });
}

```

- ➊ We are creating a range with a single element (constructing the monad instance out of the value) and taking 1 or 0 elements depending on whether the current element satisfies the predicate

There are a few different range transformations that can be implemented in a similar manner. The list of those grows if we allow ourselves to use mutable function objects. While the mutable state is discouraged, it should be safe in this case since the mutable data is not shared (chapter 5). Obviously, the fact that we can do something does not mean we should — the point is just to see that we can be quite expressive with `mbind`.

Now, the fact that ranges are monads is not something that only allows us to reimplement range transformations in a *cool* way. It is also the reason why we can have range comprehensions.

Imagine that we need to generate a list of Pythagorean triples (sum of squares of two numbers to be equal to the square of the third). If we wanted to write it with `for` loops, we would need to nest three of them. The `mbind` function allows us to perform a similar nesting with ranges:

Listing 10.4. Generating the Pythagorean triples

```

view::ints(1)                                ①
    | mbind([])(int z) {
        return view::ints(1, z)
            | mbind([z])(int y) {          ②
                return view::ints(y, z) |
                    view::transform([y,z](int x) {
                        return std::make_tuple(x, y, z); ③
                    });
            });
    }
    | filter([] (auto triple) {           ④
        ...
    });
}

```

- ➊ Generate an infinite list of integers
➋ For each of those integers, which we will call `z` generate a list of integers from 1 to `z` and call them `y`
➌ And for each of those, generate integers between `y` and `z`
➍ We now have a list of triples, we just need to filter out those that are not Pythagorean

Having flattened out ranges is quite useful, so the ranges library provides us with a few special functions which allow us to write the same code in a bit more readable way using the combination of `for_each` and `yield_if`.

Listing 10.5. Generating the Pythagorean triple with range comprehensions

```

view::for_each(view::ints(1), [](int z) {
    return view::for_each(view::ints(1, z), [z](int y) {
        return view::for_each(view::ints(y, z), [y,z](int x) {
            return yield_if(
                x * x + y * y == z * z,
                std::make_tuple(x, y, z)
            );
        });
    });
});

```

- ➊ Generate the (x, y, z) triples like in the previous example
- ➋ If the (x, y, z) is a Pythagorean triple, put it into the resulting range

We have two components in a range comprehension. The first one is the `for_each` which will traverse any collection you give to it, and collect all the values that you yield from the function you pass to it. If we have multiple nested range comprehensions, all the yealded values will be placed consecutively in the resulting range. It will not generate a range of ranges, it will flatten out the result. The second part is `yeald_if`. It puts a value in the resulting range if the predicate specified as the first argument holds.

In a nutshell, a range comprehension is nothing more than a `transform` or `mbind` coupled with `filter`. And since all these functions exist for any monad, and not only for ranges, we could also call them monad comprehensions.

10.5 Failure handling

One thing that we have touched a bit at the beginning of this chapter are the functions that communicate errors through the return type, and not by throwing exceptions.

The main job a function has in FP, in fact, the only function, is to calculate the result and return it. If the function can fail, we can make it return a value if it has calculated one, or return nothing if there was an error. As we have seen in the previous chapter, we can do it by making the return value optional.

10.5.1 `std::optional<T>` as a monad

Optionals allow us to express the possibility that a value can be missing. While this is good by itself, it has a downside that we need to have constant checks whether the value is present or not if we want to use it.

For the `user_full_name` and `to_html` functions we had defined earlier that return `std::optional<std::string>`, we would get a code ridden with checks:

```
std::optional<std::string> current_user_html()
{
    if (!current_login) {
        return {};
    }

    const auto full_name = user_full_name(current_login.value());

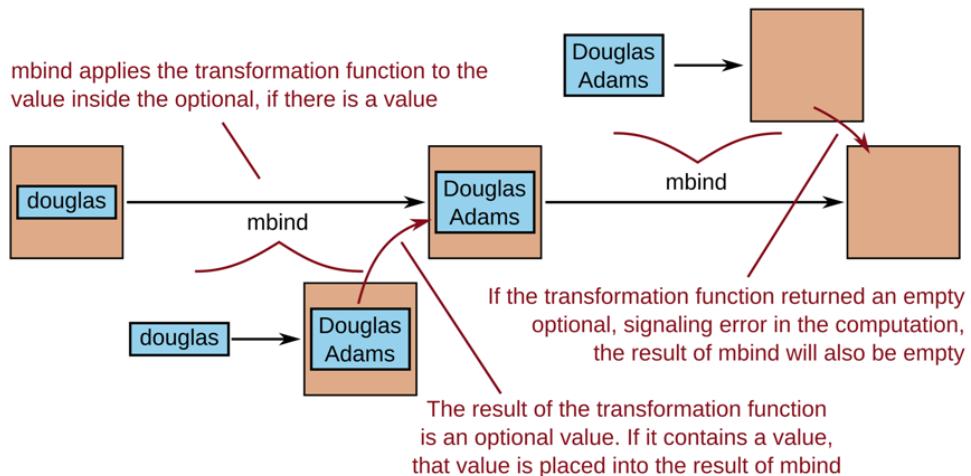
    if (!full_name) {
        return {};
    }

    return to_html(full_name.value());
}
```

Now imagine we had more functions that we need to chain like this. The code starts to look like old C code where we need to check `errno` after almost each function call we make.

Instead, we will do something smarter. As soon as we see a value with some kind of context that should be stripped out when calling another function, we should think of monads. With optional values, the context is the information whether the value is present or not. Since the other functions take normal values, this context needs to be stripped out when calling them.

Figure 10.11. If we want to perform error handling with optional values instead of exceptions, we can chain functions that return optional values with `mbind`. The chain will be broken as soon as any of the transformations fails and returns an empty optional. If all the transformations succeed, we will get an optional containing the resulting value.



And this is exactly what monads allow us to do — to compose functions while not having to do any extra work in order to handle the contextual information.

The `std::make_optional` is the constructor function for the monad, and `mbind` is easily defined:

```
template <typename T, typename F>
auto mbind(const std::optional<T>& opt, F f)
    -> decltype(f(opt.value())) ①
{
    if (opt) {
        return f(opt.value()); ②
    } else {
        return {}; ③
    }
}
```

- ① We need to specify the return type since we are returning just {} in the case when we have no value
- ② If `opt` contains a value, transform it using the function `f` and return its result since it already returns an optional value
- ③ If we have no value, return an empty instance of `std::optional`

This will give us an empty result if the original value was missing, or if the function `f` fails and it returns an empty optional. The valid result is returned otherwise. If we chain multiple functions with this, we get automatic error handling — the functions will be executed until the first one that fails. If no function fails, we will get the result.

Now our function becomes much simpler:

```
std::optional<std::string> current_user_html()
{
    return mbind(
        mbind(current_login, user_full_name),
        to_html);
}
```

Alternatively, we could also create a `mbind` transformation that will have the same pipe syntax as `do` and get the code much more readable:

```
std::optional<std::string> current_user_html()
{
    return current_login | mbind(user_full_name)
        | mbind(to_html);
}
```

Note that this looks exactly like the example we had for functors, where in that case we had ordinary functions and we have used `transform`, and here we have functions that return optional values, and we are using `mbind` instead.

10.5.2 `expected<T, E>` as a monad

Now, `std::optional` allows us to handle errors, but it does not tell us what the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/functional-programming-in-cplusplus>

Licensed to Alexey Fadeev <alexey.s.fadeev@gmail.com>

actual error is. With `expected<T, E>` we can have both.

Like with `std::optional<T>`, if we haven't encountered an error, `expected<T, E>` will contain a value. Otherwise, it will contain the information about the error.

Listing 10.6. Composing the expected monad

```
template <
    typename T, typename E, typename F,
    typename Ret = typename std::result_of<F(T)>::type ①
    >
Ret mbind(const expected<T, E>& exp, F f)
{
    if (!exp) {
        return Ret::error(exp.error()); ②
    }

    return f(exp.value()); ③
}
```

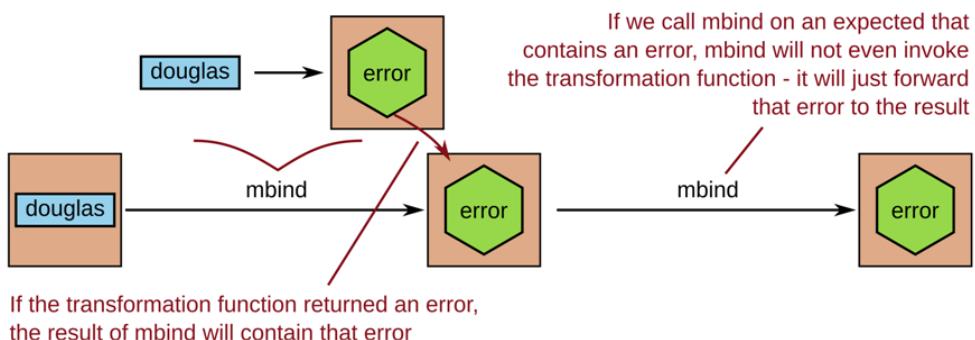
① `f` can return a different type, so we need to deduce it so that we can return it

② If `exp` contains an error, we are passing it on

③ Otherwise, we will return the result `f` returned us

We can now easily convert our functions not only to tell us whether or not we have the value, but also to contain an error. For the sake of simplicity, we will use integers to denote errors.

Figure 10.12. When using expected for error handling, we can chain the multiple transformations that can fail just like with optional values. As soon as we encounter an error, the transformations will stop and we will get that error in the resulting expected object. If there was no error, we will get the transformed value.



```
expected<std::string, int> user_full_name(const std::string& login);
expected<std::string, int> to_html(const std::string& text);
```

The implementation of `current_user_html` function does not need to change in any way.

```
expected<std::string, int> current_user_html()
{
    return current_login | mbind(user_full_name)
           | mbind(to_html);
}
```

As before, the function will return a value if no error has occurred. Otherwise, as soon as any of the functions we are binding to return an error, the execution will stop and return that error to the caller.

It is important to note here that for monads we need to have one template parameter, and here we have two. We could have easily decided to do `mbind` on the error if we needed to perform the transformations on it instead of on the value.

10.5.3 Try monad

The `expected` type allow us to use anything as the error type. We could use integers as error codes, strings to convey the error messages, or some combination of the two. We could also use the same exception class hierarchy we would use with normal exception handling by specifying the error type to be `std::exception_ptr`.

Listing 10.7. Wrapping the functions that use exceptions into the expected monad

```
template <typename F,
         typename Ret = typename std::result_of<F()>::type,
         typename Exp = expected<Ret, std::exception_ptr>
Exp mtry(F f)
{
    try {
        return Exp::success(f());                                ①
    }
    catch (...) {
        return Exp::error(std::current_exception());          ②
    }
}
```

- ① `f` is a function without arguments. If we need to call it with arguments, we can always pass a lambda
- ② If no exception was thrown, we will return an instance of `expected` that contains the result of calling `f`
- ③ If any exception gets thrown, we will return an instance of `expected` that contains a pointer to it

This will allow us to easily integrate the already existing code that uses exceptions with the error handling based on the `expected` monad. For example, we might want to get the first user in the system. The function that retrieves the list of users can throw, and we also want to throw if there are no users at all:

```
auto result = mtry([]= {
    auto users = system.users();
```

```

    if (users.empty()) {
        throw std::runtime_error("No users");
    }

    return users[0];
});
```

The result will be either a value, or a pointer to the exception that was thrown.

We can also do it the other way round — if we have a function that returns an instance of `expected` with a pointer to the exception, we can easily integrate it into the code that uses exceptions. We can simply create a function that will either return the value stored in the `expected` object, or throw the exception it holds:

```

template <typename T>
T get_or_throw(const expected<T, std::exception_ptr>& exp)
{
    if (exp) {
        return exp.value();
    } else {
        std::rethrow_exception(exp.error());
    }
}
```

These two functions allow us to have both monadic error handling, and exception based error handling in the same program, and to integrate them in a nice way.

10.6 Handling state with monads

One of the reasons why monads are popular in the functional world is that they allow us to implement stateful programs in a pure way. For us, this is not necessary — mutable state is something that we've always had in C++.

On the other hand, if we want to implement our programs using monads and different monad transformation chains, it could be useful to be able to track the state of each of those chains.

As we have said numerous times, if we want to have pure functions, we can not have any side effects in them — we can not change anything from the outside world. How can we change the state then?

Impure functions can make implicit changes to the state — we don't see what happens and what is changed just by calling the function. If we want to do it in a pure way, we need to make every change explicit.

The simplest way we can do it is to pass each function the current state along with its regular arguments, and the function should return the new state. We have already talked about this idea in the chapter 5 where we entertained the idea of handling mutable state by creating new worlds instead of changing the current one.

Let's see this on an example. We will reuse the `user_full_name` and `to_html` functions, but this time we don't want to handle failures, we want to keep some kind of debugging log of the operations we performed. Said log is the state that we want to change. Instead of using `optional` or `expected` which are union types allowing us to have either a value inside them, or something denoting an error, we want a product type that will contain both the value and additional information (the debugging log) at the same time¹³.

The easiest way to do this is to create a class template:

```
template <typename T>
class with_log {
public:
    with_log(T value, std::string log = std::string())
        : m_value(value)
        , m_log(log)
    {}

    T value() const { return m_value; }
    std::string log() const { return m_log; }

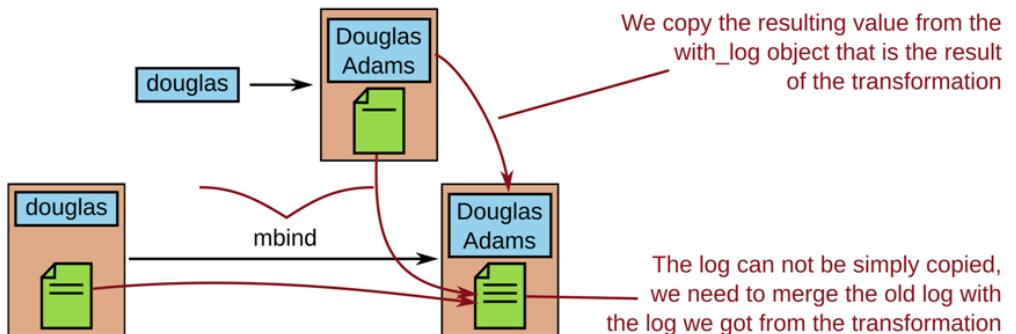
private:
    T m_value;
    std::string m_log;
};
```

Now we can redefine the `user_full_name` and `to_html` functions to return the values along with the log. Both of them will return the result along with their personal log of performed actions.

```
with_log<std::string> user_full_name(const std::string& login);
with_log<std::string> to_html(const std::string& text);
```

¹³ In literature, this monad is usually called *Writer monad* because we are only writing the contextual information, we are not using the context in the `user_full_name` and `to_html` functions

Figure 10.13. Unlike the previous monads like optional and expected where the result depended only on the last transformation, with_log takes us a bit further — we are collecting the log from all the transformations we have performed.



As before, if we want to be able to easily compose these two functions, we need to make a monad out of `with_log`. Creating the monad construction function is trivial — we can either use the `with_log` constructor, or we can create a `make_with_log` function the same way we wrote `make_vector`.

The `mbind` function is where the main magic happens. It should take an instance of `with_log<T>` which contains the value and the current log (state), a function that transforms the value and returns the transformed value along with the new logging information. The `mbind` function will need to return the new result along with new logging information appended to the old log.

Listing 10.8. Maintaining the log with `mbind`

```
template <typename T,
          typename F>
          typename Ret = typename std::result_of<F(T)>::type
Ret mbind(const with_log<T1>& val, F f)
{
    const auto result_with_log = f(val.value());           ①
    return Ret(result_with_log.value(),
               val.log() + result_with_log.log());           ②
}
```

- ① Transforming the given value with `f` which will give us the resulting value and the log string that `f` generated
- ② We need to return the value we got, but we can not just return the log that `f` returned, we need to concatenate it with the previous log

This approach to logging has multiple advantages to logging to the standard output. We can have multiple parallel logs — one for each monad transformation chain we create — without the need for any special logging facilities. We have the ability

for one function to write to various different logs depending on who called it, and we do not have the need to specify "this log goes here" and "that log goes there".

Even more, this approach to logging will allow us to keep logs in chains of asynchronous operations without having debugging output of different chains interleaved.

10.7 Concurrency and the continuation monad

So far, we have seen a few different monads. All of them contained zero or more values, and some contextual information.

This might induce a mental picture that a monad is some kind of container that knows how to operate on its values. And if we have an instance of said container, we might access those elements when we need them.

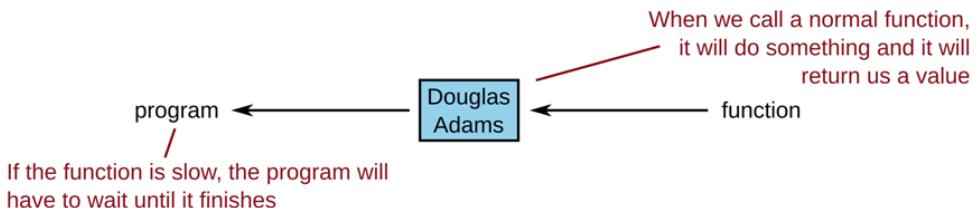
This analogy works for more than a few different monads, but not for all. If we recall the monad definition, the only thing that it allows us is to create a monad instance out of a normal value, or to perform some transformation on the value already inside the monad. We haven't been given a function that can extract that value from inside a monad.

This probably sounds like an oversight — to have a container that holds some data, but which doesn't allow us to get said data. After all, we are used only to containers for which this is not the case — we can access all the elements in a vector, a list, an optional, etc. Right?

Not really. While we might not call the input streams like `std::cin` *containers*, they are. They contain elements of type `char`. We also had the `istream_range<T>` which is essentially a container of zero or more values of type `T`. The difference compared to the normal containers is that we don't know their sizes in advance, and we can not access the elements until the user enters them.

From the point of view of the person who writes the code, there is no much difference — we can easily write a generic function that will perform operations like `filter` and `transform` which would work for both vector-like containers, and the input stream-like containers.

Figure 10.14. When we call a function from the main program, the program is blocked until the function finishes. If the function is slow, the program can be blocked for a long time, and it could use that time to perform other tasks.

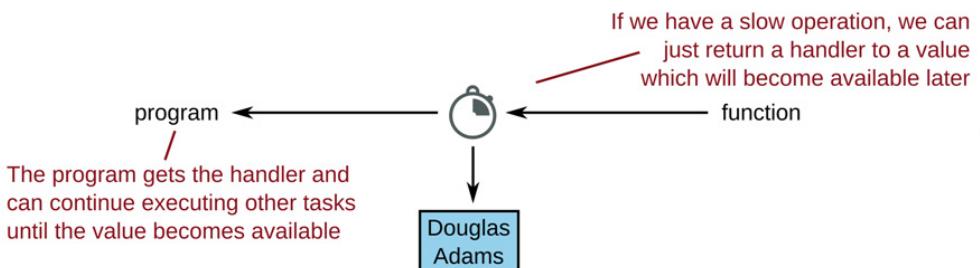


But there is a huge difference in executing said code. If we are executing the code on input stream-like containers, the execution of our program will be blocked until the user enters all the data we need.

In interactive systems, we should never be allowed to block the program. Instead of requesting data and processing it, it is much better just to tell the program what to do with it when it becomes available.

Imagine we want to extract the title of a web page. We would need to connect to the server on which that page is located, we would need to wait for the response, and then parse it in order to find the title. The operation of connecting to the server and retrieving the page can be slow, and we can not allow ourselves to block the program while it finishes.

Figure 10.15. Instead of waiting for the slow function to finish, it is better to have that function return some kind of handler which we can use to access the value once it is calculated.



We need to perform the request, and then continue with all the other tasks our program needs to perform. When the request is finished, and we have the data, then we return to it, and process the data.

This means that we need some kind of handler which will give us the access to the data once it becomes available. We will call it a *future* because the data is not available immediately, but will become available sometime in the future. Since this idea of future values can be quite useful for other things as well, we will not limit it

to be able to contain only strings (the source of a web page), but it will be a generic handler `future<T>`.

To summarize, we are going to have a handler called `future<T>` which might not yet contain a value, but the value of type `T` will be in it at some point in time. With it we will be able to design programs that have different components executed concurrently or asynchronously — any time we have a slow operation, or an operation that we don't know its execution time, we will make it return a `future` instead of an ordinary value.

The first thing to notice is that it looks like a container type, but a container whose element we can not get directly. That is, unless the asynchronous operation has finished, and the value is in the container.

10.7.1 Futures as monads

The future object, as we defined it just screams "Monad!". It is a container-like thing, it can hold zero or one result depending on whether the asynchronous operation is finished or not.

Let's first check whether a future can be a functor. We need to be able to create a `transform` function that takes a `future<T1>` and a function `f: T1 → T2`, and it should give us an instance of `future<T2>`.

Conceptually, this should not be a problem. A `future` is a handler to some future value. If we can get the value when the future arrives, we will be able to pass it to the function `f` and get the result. This means that at some point in the future, we will have the transformed value. If we know how to create a handler for it, we can create the `transform` function for futures, and the `future` is a functor.

This could be useful when we do not care about the whole result of some asynchronous operation, but we just need some part of it. Like in the example we had earlier — where we wanted to get the title of a web page. If we had a future and the `transform` function defined for it, we could do it easily:

```
get_page(url) | transform(extract_title)
```

This will give us a future of string which, when it arrives, will give us only the title of the web page.

Now, if the `future` is a functor, it is time to move to the next step, and check whether it is a monad.

Constructing a handler that already contains a value should be easy. The more interesting part is `mbind`.

Let's change the result type of `user_full_name` and `to_html` yet again. This time, in order to get the full name of the user, we need to connect to a server and fetch the data. So, the operation should be performed asynchronously. Also, let's

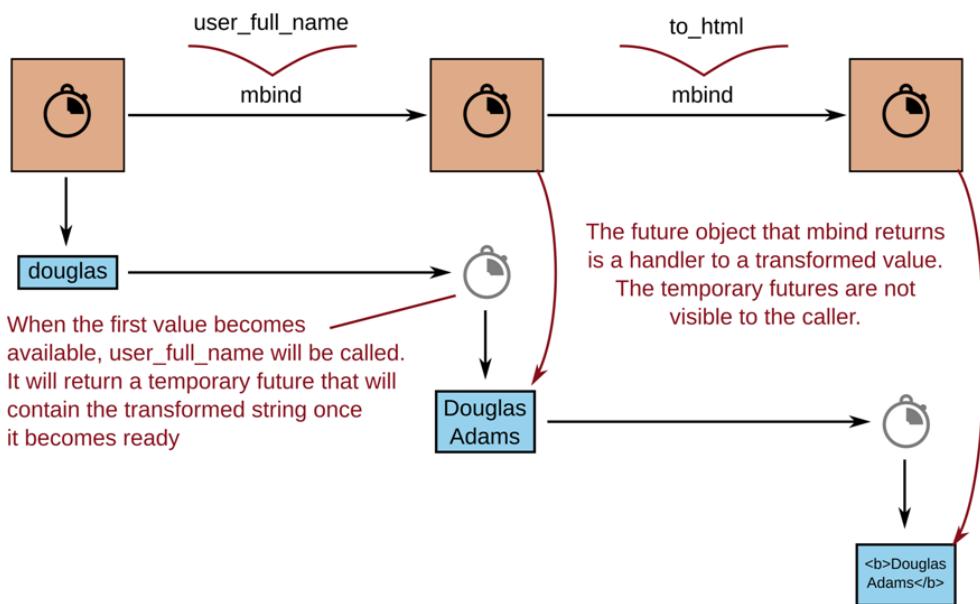
imagine `to_html` is a slow operation that also needs to be asynchronous. This means that both of them should return futures instead of normal values:

```
future<std::string> user_full_name(const std::string& login);
future<std::string> to_html(const std::string& text);
```

If we used `transform` to compose these two functions, we would get a future of a future of a value which sounds quite strange. Having a handler that will some time in the future give us another handler which, in turn, will even later give us the value. This becomes even more incomprehensible when we compose more than two asynchronous operations.

This is where we will truly benefit from the `mbind` function. As in the previous cases, it will allow us to avoid nesting — we will always get a handler to the value, and not a handler to a handler to a handler...

Figure 10.16. Monadic binding allows us to chain multiple asynchronous operations. The result will be the future object handling the result of the last asynchronous operation.



The `mbind` function will have to do all the dirty work for us. It needs to be able to tell when the future arrives, then call the transformation function and get the handler to the final result. And the most important part, it will have to give us the handler to the final result immediately.

With `mbind`, we will be able to chain as many asynchronous operations as we want. Using the range notation, we would get the code like this:

```
future<std::string> current_user_html()
{
    return current_user() | mbind(user_full_name)
        | mbind(to_html);
}
```

In this snippet, we have three asynchronous operations chained completely naturally. Each function continues the work done by the previous one. Because of this, the functions passed to `mbind` are usually called continuations, and the future value monad we defined is called the *continuation monad*.

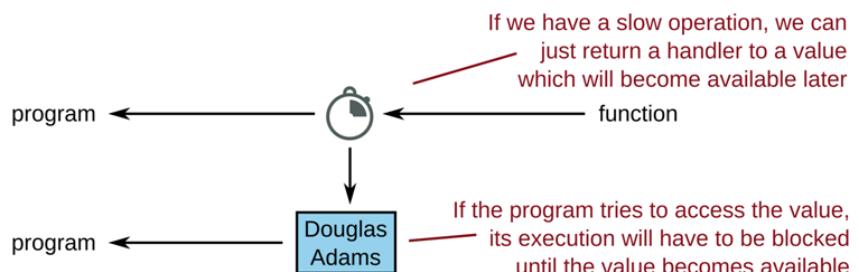
The code is localised, readable and easy to understand. If we wanted to implement the same using the common approaches like callback functions or signals and slots, this single function would have to be split into a few separate ones — every time we would call an asynchronous operation, we would need to create a new function that handles the result.

10.7.2 Implementations of the future

Now that we know what the concept of futures should be, let's analyze what is available to us in C++ world.

Since C++11, we have had `std::future<T>` which does provide a handler for some future value. Apart from a value, `std::future` can contain an exception if the asynchronous operation failed. In a sense, it is close to a `future<expected<T, std::exception_ptr>>` value.

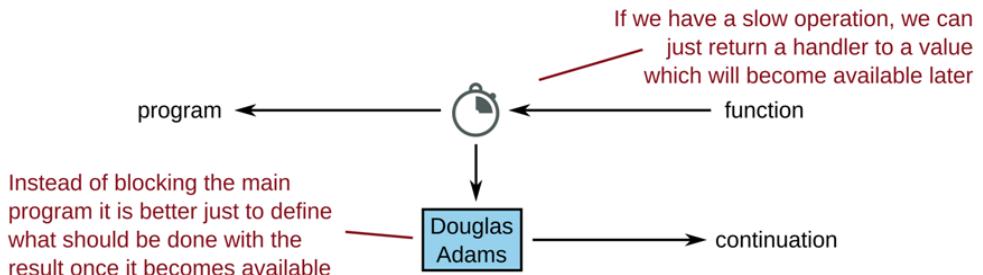
Figure 10.17. One way to access the value from the `std::future` is to use the `.get` member function. Unfortunately, this will block the caller if the future is not yet finished. This is useful when we want to perform parallel computations and want to collect the results before continuing with the program, but is a big problem in interactive systems.



The bad part about it is that it does not have a smart mechanism to attach a continuation to it. The only way to get the value is through its `.get` member function, which will block the program execution if the future is not ready. This means that we either need to block the main program, or spin off a thread that will wait for the future to arrive, or poll the future every once in a while to check

whether it has finished.

Figure 10.18. Instead of blocking the execution of the program, it would be better if we could attach a continuation function to the future object. When the value becomes available, the continuation function will be called to process it



All these options are bad. There is a proposal to extend the `std::future` with a `.then` member function which we can pass the continuation function to. Currently, the proposal is published in the Concurrency TS along with C++17 which means that most standard library vendors will support it, and the extended future class will be accessible as `std::experimental::future`.

If you don't have the access to a compiler that supports C++17, you can use `boost::future` class which already supports continuations.

The `.then` member function behaves similarly to `mbind` with a slight difference. The monadic bind takes a function whose argument is an ordinary value, and the result is a `future` object, while the `.then` wants a function whose argument is an already completed `future` object, and returns a new `future`.

This means that `.then` is not exactly the function that makes futures a monad, but it makes it trivial to implement the proper `mbind` function trivial:

Listing 10.9. Implementing `mbind` using the `.then` member function

```
template <typename T, typename F>
auto mbind(const future<T>& future, F f)           ①
{
    return future.then(                                ②
        [](future<T> finished) {                      ③
            return f(finished.get());                  ④
        });
}
```

- ① `mbind` takes a function from `T` to some `future` instance `future<T2>`
- ② but `.then` takes a function from `future<T>` and returns `future<T2>`, so we need to pass a lambda that will extract the value from the `future` object before passing it to `f`

- ③ The `.get` function will not block anything here because the continuation will be called only when the result is ready (or if an exception has occurred)

Outside of the usual C++ ecosystem, there are a few other libraries that implement their own futures. Most of them model the basic concept with the addition of being able also to handle and report errors which can occur during the asynchronous operations.

The most notable examples outside of the standard library and `boost` are the Folly library's `Future` class and Qt's `QFuture`. Folly provides a clean implementation of the future concept which can never block (`.get` will throw an exception if the future is not ready instead of blocking). While Qt provides its own `QFuture` which has a way of connecting continuations to it using signals and slots, but will also block on `.get` like the future from the standard library. The `QFuture` class also has some additional features that go beyond the basic concept by being able to collect multiple values over time instead of just a single result.

Despite these few differences, all the classes can be used to chain multiple asynchronous operations using the monadic bind operation.

10.8 Monad composition

So far, we have had an instance of a monadic object, and we used the `mbind` function to pass it through a monadic function. This would give us an instance of the same monadic type, which we could `mbind` to another function and so on.

This is analogous to normal function application where we passed a value to some function, and we got a result which we could pass to another function, and so on. If we removed the original value from the equation, we would just get a list of functions that we composed with one another, and that composition would yield a new function.

This is also possible with monads. We can express the binding behavior without the original instance of the monad, and just focus on which monadic functions we want to compose.

During the course of this chapter, we have seen a few variations of the `user_full_name` and `to_html` functions. Most of them received a string and returned a string wrapped inside some monadic type. They looked like this (with `M` replaced with `optional`, `expected` or some other wrapper type):

```
user_full_name : std::string → M<std::string>
to_html       : std::string → M<std::string>
```

If we wanted to create a function that composes these two, we would need to create a function that receives an instance of `M<std::string>` representing the user we

need the name for, and inside it, we need to pass that through two `mbind` calls:

```
M<std::string> user_html(const M<std::string>& login)
{
    return mbind(
        mbind(login, user_full_name),
        to_html);
}
```

This works, but is unnecessarily verbose. It would be easier if we could just say that `user_html` should be a composition of `user_full_name` and `to_html`.

We can create the generic composition function easily. When composing normal functions, we are given two functions $f: T_1 \rightarrow T_2$ and $g: T_2 \rightarrow T_3$, and, as the result, we get a function from T_1 to T_3 . With monad composition, the things change slightly. The functions do not return normal values, but values wrapped in a monad. Therefore, we will have $f: T_1 \rightarrow M<T_2>$ and $g: T_2 \rightarrow M<T_3>$.

Listing 10.10. Composing two monad functions

```
template <typename F, typename G>
mcompose(F f, G g) {
    return [=](auto value) {
        mbind(f(value), g);
    };
}
```

We can now define `user_html` simply as:

```
auto user_html = mcompose(user_full_name, to_html);
```

It can also be used for *simpler* monads like ranges (and vectors, lists, arrays). Imagine we had a function `children` that gives us a range containing all the children of a specified person. It has the right signature for a monad function — it takes a single `person_t` value, and gives us a range of `person_t`. We could easily create a function that retrieves all grandchildren:

```
auto grandchildren = mcompose(children, children);
```

The `mcompose` function allows us to have short and highly generic code, but there is also one theoretical benefit. If you recall, we had three monad rules which weren't really intuitive. With this composition function, we can express them in a much nicer way.

If we compose any monadic function with the constructor function for corresponding monad, we get the same function:

```
mcompose(f, construct) == f
mcompose(construct, f) == f
```

And the associativity law tells us that if we have three functions f , g and h that we want to compose, it is completely irrelevant whether we first compose f and g , and compose the result with h , or we compose g and h , and compose f with the result:

```
mcompose(f, mcompose(g, h)) == mcompose(mcompose(f, g), h)
```

This composition is also called Kleisli composition, and it generally has the same attributes like the normal function composition.

10.9 Summary

- Programming patterns are usually connected to the object-oriented programming, but the functional programming world is also filled with often used idioms and abstractions like the functor and monad;
- Functors are collection-like structures that know how to apply a transformation function on their contents;
- Monads know everything that functors do, but they have two additional operations — they allow creating monadic values from normal values, and they know how to flatten out nested monadic values;
- Functors allow us to easily handle and transform wrapper types, while monads allow us to compose functions that return wrapper types;
- It is often useful to think about monads as boxes, where we need to take the term *box* loosely to cover the cases like the continuation monad — a box that will eventually get the data in it;
- While we can open a box in the real world to see what is inside, this is not the case with monads. In the general case, we can only tell the box what to do with the value(s) it has — we can not always access the value directly.

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch10

Template meta-programming

This chapter covers:

- How to use transform types during compilation
- How to extract the type of elements contained in given a collection
- How to use **constexpr-if** to perform branching at compile-time
- How to perform static introspection — to check for type properties at compile-time
- How to implement a generic currying function
- Using **std::invoke** and **std::apply**
- How to create a DSL to define transactions for data record updates

The normal way to think of programming is — we write some code, we compile it, and then the user executes the compiled binary. This is the way programming works for the most part.

With C++, we can also write a different kind of programs — programs that the compiler executes **while** it compiles our code. This might seem like a strange thing to be able to do — how useful can it be to execute code when we still have no input from the user and no data to process? But the main point of compile-time code execution is not in processing data at run-time, as most of the data will only become available when we execute the compiled program, but rather in manipulating the things that are available during compilation — types and the generated code.

This is a necessity when writing optimized generic code. We might want to implement an algorithm differently depending on the features of the type we are

given. For example, when working with collections, it is often important to know whether the collection is randomly accessible or not. Depending on this, we might want to choose a completely different implementation of our algorithm.

The main mechanism for compile-time programming (or meta programming) in C++ are templates. Let's take a look at the definition of the `expected` template class we introduced in chapter 9:

```
template<typename T, typename E = std::exception_ptr>
class expected
{
    ...
};
```

This is a template parametrized on two types `T` and `E`. When we specify these two parameters, we get a concrete type that we can create instances of. For example, if we set `T` to be `std::string` and `E` to be `int`, we will get a type called `expected<std::string, int>`. If we set both `T` and `E` to be `std::string`, we will get a different type `expected<std::string, std::string>`.

These two resulting types are similar, and they will be implemented in the same way, but they are still two distinct types. We can not convert an instance of one type to the other. Furthermore, these two types will have completely separate implementations in the compiled program binary.

So, we've got a thing called `expected` which accepts two types, and gives us a type as the result. It is like a function that does not operate on values, but on the types themselves. We will call these **meta-functions** in order to differentiate them from ordinary functions.

Meta-programming with templates (or TMP — template meta-programming) is a huge topic that deserves a whole book for itself. In this chapter, we will cover only some of the parts relevant to this book. We are going to focus on C++17 mostly because it introduced a few features that make writing TMP code much easier.

11.1 Manipulating types at compile-time

Imagine we want to create a generic algorithm that sums all items in a given collection. It should just take the collection as its argument, and it should return the sum of all elements inside the collection. The question is what will be the return type of this function?

This is easy with `std::accumulate` — the type of the result is the same as the type of the initial value used for accumulation. But here we have a function that only takes the collection, and no initial value.

```
template <typename C>
??? sum(const C& collection)
{
```

```
...  
}
```

The most obvious answer is to have the return type be the type of the items contained in the given collection. If we have been given a vector of integers, the result should be an integer. If we got a linked list of doubles, the result should be a double, and for any collection of strings, the result should be a string.

The problem is that we know the type of the collection, not the type of the items contained in it. We need to somehow write a meta-function that will get a collection type, and return the type of the contained item.

One thing common to most collections is that we can use iterators to traverse over them. If we dereference an iterator, we get a value of the contained type.

If we want to create a variable to store the first element in a collection, we can do it like this:

```
auto value = *begin(collection);
```

The compiler will be able to deduce its type automatically. If we have a collection of integers, `value` will be an `int` just like we wanted. This means that the compiler is able to properly deduce the type at the time of compilation. Now we just need to leverage this fact somehow to create a meta-function that does the same.

It is important to note that the compiler does not know whether the given collection contains any items at the time of compilation. It will be able to deduce the type of `value` even if the collection can be empty at runtime. Of course, we will get a runtime error if the collection is empty and we try to dereference the iterator returned by `begin`, but we are not interested in that at this point.

If we have an expression, and we want to get its type, we can use the `decltype` specifier. We will create a meta-function that takes a collection, and returns the type of the contained item. We can implement this meta-function as a generic type alias¹⁴ like this:

```
template <typename T>  
using contained_type_t = decltype(*begin(T()));
```

Let's dissect this into smaller pieces to see what is happening. The template specification itself tells us that we have a meta-function called `contained_type_t` that takes one argument — the type `T`. This meta-function will return the type of the expression contained in the `decltype` specifier.

When we declared the `value` variable, we had a concrete collection to call `begin` on. Here, we do not have a collection, we just have its type. Because of that, we are

¹⁴ Type aliases — en.cppreference.com/w/cpp/language/type_alias

creating a default-constructed instance `T()` and passing it to `begin`. The resulting iterator gets dereferenced, and the `contained_type_t` meta-function returns the type of the value that the iterator points to.

Unlike the previous code snippet, this will not produce an error at runtime because we are just playing around with types at compile-time. The `decltype` will never actually execute the code we pass to it, it just returns the type the expression has without evaluating it.

While this sounds great, the meta-function above has two significant problems.

The first is that it relies on `T` being default-constructible. While all collections in the standard library have default constructors, it is not difficult to imagine a collection-like structure that does not. For example, we can see the previously mentioned `expected<T, E>` as a collection that contains zero or one value of type `T`. And it does not have a default constructor —if we want to have an empty `expected<T, E>`, we will need to specify the error explaining why it is empty.

This means that we can not use `contained_type_t` with it — calling `T()` will yield a compiler error. In order to fix this, we need to replace the constructor call with the `std::declval<T>()` utility meta-function. It takes any type `T`, be it a collection, and integral type or any custom type like `expected<T, E>`, and it **pretends** to create an instance of that type so that we can use it in our own meta-functions when we need values instead of types which is often the case when we use `decltype`.

In our original scenario of summing items in a collection, we knew exactly how we can implement the summation, the only problem we had was that we do not know the return type. The `contained_type_t` meta-function gives us the type of the elements contained in a collection, so we can use it to deduce the return type of the function that sums all items in a collection.

We can use it in the following manner:

```
template <typename C,
          typename R = contained_type_t<C>>
R sum(const C& collection)
{
    ...
}
```

While we are calling these **meta-functions**, they are nothing more than templates that define something. The meta-functions are **invoked** by instantiating this template. In this case, we instantiated the `contained_type_t` template for the type of the collection `C`.

11.1.1 Debugging deduced types

The second problem with our implementation of `contained_type_t` is that it does

not really do what we want. If we try to use it, we will soon encounter problems. If we try to compile the previous code, we will get compiler messages hinting that the result of `contained_type_t` for a `std::vector<T>` is not `T`, but something else.

In cases like these — when we expect one type, but the compiler claims to have another, it is useful to be able to check exactly which type we have. We can either rely on the IDE we are using to show us the type, which can be imprecise, or we can force the compiler to tell us.

One of the neat tricks that we can use for this is to declare a class template, but not to implement it. Whenever we want to check for a specific type, we can just try to instantiate that template, and the compiler will report an error specifying exactly which type we passed.

Listing 11.1. Checking which exact type is deduced by `contained_type_t` meta-function

```
template <typename T>
class error;

error<contained_type_t<std::vector<std::string>>>();
```

This will produce a compilation error similar to this (depending on the compiler you are using):

```
error: invalid use of incomplete type
'class error<const std::string&>'
```

This means that `contained_type_t` deduced the type to be a constant reference to a string, instead of deducing it to be a string like we wanted — and like `auto` value would deduce.

This is to be expected because `auto` does not follow the same deduction rules as `decltype`. When using `decltype`, we get the exact type of a given expression, while `auto` tries to be smart and behaves much like the template argument type deduction.

Since we got a constant reference to the type, and we want to get just the type, we need to remove the reference part of the type and the `const` qualifier. In order to remove the `const` and `volatile` qualifiers, we can use the `std::remove_cv_t` meta-function, and we can use `std::remove_reference_t` to remove the reference.

Listing 11.2. Full implementation of the `contained_type_t` meta-function

```
template <typename T>
using contained_type_t =
    std::remove_cv_t<
        std::remove_reference_t<
```

```

        decltype(*begin(std::declval<T>())))
>
>;

```

If we were to check the resulting of `contained_type_t<std::vector<std::string>>` now, we would get just `std::string`.

The `<type_traits>` header

Most of these standard meta-functions are defined in the `<type_traits>` header. It contains a dozen useful meta-functions for manipulating the types, and also for simulating `if` statements and logical operations in meta-programs.

The meta-functions that end with `_t` were introduced in C++14. In order to perform similar type manipulations on older compilers that only support C++11 features, you will need to use more verbose constructs. You can check out en.cppreference.com/w/cpp/types/remove_cv for examples of using the regular `remove_cv` instead of the one with the `_t` suffix which we used.

Another useful utility when writing and debugging meta-functions is `static_assert`. Static assertions can ensure that some rule holds during compilation time. For example, we could write a series of tests to verify the implementation of `contained_type_t`:

```

static_assert(
    std::is_same<int, contained_type_t<std::vector<int>>>(),
    "std::vector<int> should contain integers");
static_assert(
    std::is_same<std::string, contained_type_t<std::list<std::string>>>(),
    "std::list<std::string> should contain strings");
static_assert(
    std::is_same<person_t*, contained_type_t<std::vector<person_t*>>(),
    "std::vector<person_t> should contain people");

```

The `static_assert` expects a Boolean value that can be calculated at compile-time and it stops the compilation if that value turns out to be `false`. In the above example, we are checking that the type that the `contained_type_t` meta-function is exactly the type that we expected it to be.

We can not compare types by using the operator `==`, we need to use its **meta** equivalent — `std::is_same`. The `std::is_same` meta-function takes two types, and returns `true` if the types are identical, and `false` otherwise.

11.1.2 Pattern matching during compilation

Let's see how the previously used meta-functions are implemented. We will start by implementing our own version of the `is_same` meta-function. It should take two arguments and return `true` if the types are the same and `false` otherwise. Since we

are manipulating types here, our meta-function will not return a value—true or false—but a type—`std::true_type` or `std::false_type`. You can think of these two as Boolean constants for meta-functions.

When defining meta-functions, it is often useful to think about what the result is in the general case, and then cover the specific cases and calculate the results for them. For `is_same`, we have two cases—the case that we have been given two different types and that we need to return `std::false_type` and the case that we have been given the same type for both parameters where we need to return `std::true_type`. The first case is more general, so we will cover it first:

```
template <typename T1, typename T2>
struct is_same : std::false_type {};
```

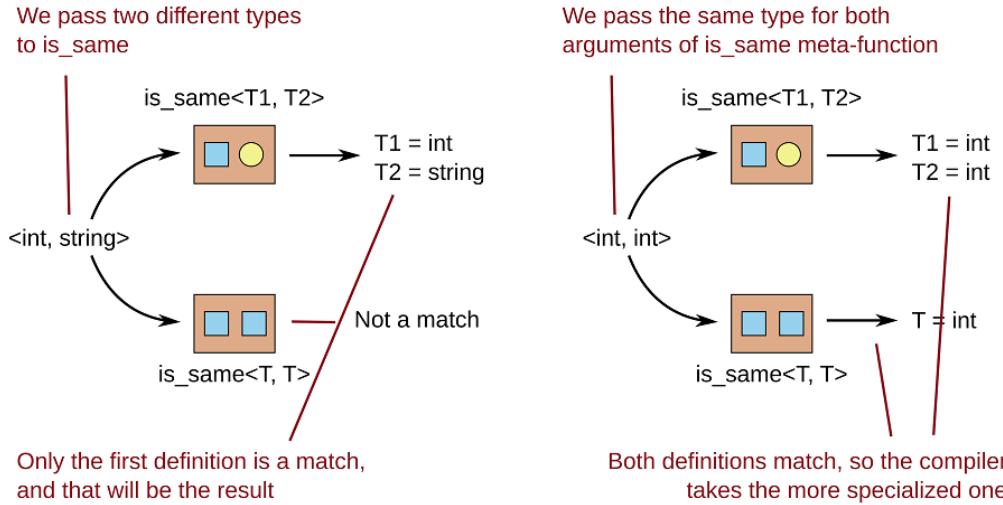
With this definition, we have created a meta-function of two arguments that always returns `std::false_type` regardless of what `T1` and `T2` are.

Now we need to cover the second case:

```
template <typename T>
struct is_same<T, T> : std::true_type {};
```

We have created a specialization of the previous template that will be used only if `T1` and `T2` are the same. When the compiler sees `is_same<int, contained_type_t<std::vector<int>>>`, it will first calculate the result of `contained_type_t` meta-function and that result will be `int`. Then, it will find all the definitions of `is_same` that can be applied to `<int, int>` and it will pick the most specific one.

Figure 11.1. When we provide different arguments to `is_same`, only the first definition which returns `false_type` will be a match. If we pass the same type as both parameters, both definitions will be a match and the more specialized match will win — the one returning `true_type`



With the above implementation, it will find both implementations — the one that inherits `std::false_type` and the one that inherits `std::true_type`. Since the second one is more specialized, it will be chosen.

What would happen if we wrote `is_same<int, std::string>`? The compiler would generate the list of definitions that can be applied to an `int` and a `std::string`. In this case, the specialized definition will not be applicable because there is nothing we can substitute `T` for in `<T, T>` to get `<int, std::string>`. The compiler will pick the only definition it can, and that is the first one that inherits from `std::false_type`.

The `is_same` is a meta-function that returns a compile-time Boolean constant. We can implement a function that returns a modified type in a similar way. We will implement the `remove_reference_t` function equivalent to the one from the standard library.

This time we have three different cases:

- The general case is when we are given a non-reference type,
- We have been given an l-value reference,
- We have been given an r-value reference.

In the first case, we need to return the type unmodified, while we need to strip out the references in the second two cases.

In this case, we can not simply make the function `remove_reference` inherit from

the result like it was with `is_same`. We need the exact type as the result, not some custom type that inherits from the result.

To do this, we will have to create a structure template that will contain a nested type definition which will hold the exact result.

Listing 11.3. Implementation of `remove_reference_t` meta-function

```
template <typename T>          1
struct remove_reference {        1
    using type = T;            1
};                                1

template <typename T>          2
struct remove_reference<T&> {    2
    using type = T;            2
};                                2

template <typename T>          3
struct remove_reference<T&&> {   3
    using type = T;            3
};                                3
```

- ➊ In the general case, `remove_reference<T>::type` will be the type `T` — we can say that in the general case, `remove_reference` returns the same type it gets
- ➋ If we got an l-value reference `T&`, we need to strip the reference and return just `T`
- ➌ If we got an r-value reference `T&&`, we need to strip the reference part and return `T`

When we implemented the `contained_type_t` meta-function, we created a template type alias. Here, we had a bit different approach. We have a template structure that defines a nested alias called `type`. In order to call the `remove_reference` meta-function and get the resulting type, we need to use the more verbose syntax than it was the case with the `contained_type_t`. We need to instantiate the `remove_reference` template, and then reach into it to get its nested type definition. For this, we need to write `typename remove_reference<T>::type` whenever we want to use it.

Since this is overly verbose, we can create a convenience meta-function `remove_reference_t` to avoid writing `typename ...::type` all the time in a similar way that the C++ does for meta-functions in the `type_traits` header.

```
template <typename T>
using remove_reference_t<T> =
    typename remove_reference<T>::type;
```

When we use the `remove_reference` template for specific template argument, the compiler will try to find all the definitions that match that argument, and it will pick up the most specific one.

If we call `remove_reference_t<int>`, the compiler will check which of the above definitions can be applied. The first one will be a match, and it will deduce that τ is `int`. The second and third definitions will not be matches since there is no possible type τ for which a reference to τ will be an `int` — `int` is not a reference type. Since there is only one matching definition, it will be used and the result will be `int`.

If we call `remove_reference_t<int&>` the compiler will again search for all matching definitions. This time it will find two. The first definition, as the most general one, will be a match, and it will match τ to be `int&`. The second definition will also be a match, and it will match $\tau&$ to be `int&`, that is, it will match τ to be `int`. The third definition will not be a match because it expects an r-value. Out of the two matches, the second one is more specific which means that τ and therefore the result will be `int`.

This process would be similar for `remove_reference_t<int&&>` with a difference that the second definition would not be a match, but the third one would.

Now that we know how to get the type of an element in a collection, we can finally implement the function that will sum all items in it.

We will assume that the default-constructed value of the item type is the identity element for addition so that we can pass it as the initial value to `std::accumulate`.

```
template <typename C,
          typename R = contained_type_t<C>>
R sum_iterable(const C& collection)
{
    return std::accumulate(begin(collection),
                          end(collection),
                          R());
}
```

When we call this function on a specific collection, the compiler will have to deduce the types of `c` and `R`. The type `c` will simply be deduced as the type of collection we passed to the function.

Since we have not defined `R`, it will get a default value which we specified to be the result of `contained_type_t<C>` which will be the type of the value we would get by dereferencing an iterator to collection `c`, and then stripping the `const` qualifier and removing the references from it.

11.1.3 Providing meta-information about types

The previous examples showed how we can find out the type of an element contained in a collection. The problem is that it is tedious work and quite error-prone.

Because of this, it is a common practice to provide information like this in the

collection class itself. For collections, it is customary to provide the type of the contained items as a nested type definition named `value_type`. We could add this information to our implementation of `expected` quite easily:

```
template <typename T, typename E>
class expected {
public:
    using value_type = T;
    ...
};
```

All container classes in the standard library — even `std::optional` — provide this. And this is something that all well-behaved container classes from 3rd party libraries should also provide.

With this additional information, we could avoid all the meta-programming we have seen so far, and simply write:

```
template <typename C,
          typename R = typename C::value_type>
R sum_collection(const C& collection)
{
    return std::accumulate(begin(collection),
                          end(collection),
                          R());
}
```

Using the `value_type` nested type has an additional benefit in the cases where the collection iterator does not return an item directly, but a wrapper type. If we used the `contained_type_t` meta-function with a collection like this, we would get the wrapper type as the result, while we would probably want to get the type of the item itself. By providing the `value_type`, the collection tells us exactly how it wants us to see the items it contains.

11.2 Checking type properties at compile-time

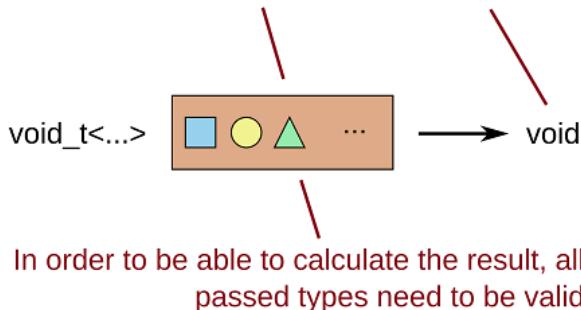
We have created two different `sum` functions:

- One that works for collections that have the `value_type` nested type definition, which is the preferred one,
- and one that works for any iterable collection.

It would be nice if we could detect whether a given collection has a nested `value_type` or not, and act accordingly.

Figure 11.2. The void_t meta-function is a strange one – it ignores all its parameters and always returns void. The reason why it is useful is that it can only be evaluated if the types we passed as its arguments are valid. If not, it will trigger a substitution failure and the compiler will ignore the definition that used it

The void_t meta-function takes an arbitrary number of types, but returns void regardless of which types we pass to it



For this, we first need to introduce the strangest meta-function yet — a function that takes an arbitrary number of types, and always returns void.

```
template <typename...>
using void_t = void;
```

This might look useless, but the result of this meta-function is not what makes it useful. The void_t is useful because it allows us to check validity of given types or expressions in the SFINAE context at compile-time. SFINAE, or "Substitution Failure Is Not An Error", is a rule that applies during the overload resolution for templates. If substituting the template parameter with the deduced type fails, the compiler will not report an error, it will just ignore that particular overload.

void_t in the standard library

void_t meta-function has been available in the standard library since C++17. If you are using an older standard library which does not support it, you can define your own like we did above.

This is exactly where void_t comes into play. We can provide it with as many types as we want, and if the types are invalid, the overload we are using void_t in will just be ignored. We can easily create a meta-function that checks whether a given type has a nested value_type or not.

Listing 11.4. Meta-function which detects whether a type has a nested value_type

```
template <typename C,
         typename = void_t<>>
struct has_value_type
```

1
1
1

```

    : std::false_type {};

template <typename C> ①
struct has_value_type<C,
    void_t<typename C::value_type>> ②
    : std::true_type {}; ③

```

- ① The general case — we assume that an arbitrary type does not have a nested `value_type` type definition
- ② We are creating a specialized case which will be considered only if `typename C::value_type` is an existing type — if `C` has a nested `value_type`

With this, we can now define a function that sums all items in a collection that takes care whether said collection has a nested `value_type` or not:

```

template <typename C>
auto sum(const C &collection)
{
    if constexpr (has_value_type<C>()) {
        return sum_collection(collection);
    } else {
        return sum_iterable(collection);
    }
}

```

If a given collection does not have a nested `value_type`, we can not call `sum_collection` on it. The compiler would try to deduce the template parameters and fail.

This is where the **constexpr-if** comes into play. The regular `if` statement checks its condition at run-time, and it requires both branches to be compilable. The **constexpr-if**, on the other hand, just requires both branches to have a valid syntax, but it will not compile both branches.

In our case, calling `sum_collection` on a collection that does not have a nested `value_type` would yield an error, but the compiler will only see the `else` branch in that case because `has_value_type<C>()` will be false.

Now, what will happen if we pass something that does not have a `value_type` type nor is iterable to `sum`? We will get a compilation error that `sum_iterable` can not be called on that type. It would be nicer if we also guarded against that just like we guarded against calling `sum_collection` when it is not applicable.

We need to check whether a collection is iterable — whether we can call `begin` and `end` on it, and whether we can dereference the iterator returned by `begin`. We do not care whether `end` can be dereferenced since it can be a special sentinel type (see chapter 7).

We can use `void_t` for this as well. While `void_t` checks for validity of types, it can also be used to check expressions with a little help of `decltype` and `std::declval`.

Listing 11.5. Meta-function which detects whether a type is iterable

```

template <typename C,          1
        typename = void_t<>> 1
struct is_iterable           1
    : std::false_type {};    1

template <typename C>          2
struct is_iterable<          2
    C, void_t<decltype(*begin(std::declval<C>())), 2
                  decltype(end(std::declval<C>()))>> 2
    : std::true_type {};     2

```

- ➊ The general case — we assume that an arbitrary type is not iterable
- ➋ We are creating a specialized case which will be considered only if C is iterable, and if its begin iterator can be dereferenced

We can now define a complete `sum` function that checks for validity of the collection type before calling any of the functions on that collection:

```

template <typename C>
auto sum(const C& collection)
{
    if constexpr (has_value_type<C>()) {
        return sum_collection(collection);
    } else if constexpr (is_iterable<C>()) {
        return sum_iterable(collection);
    } else {
        // do nothing
    }
}

```

Now we have a function that properly guards all its calls, and we can even choose to handle the case when we are given a type that does not look like a collection. We could even choose to report a compilation error in this case (check out the accompanying code example).

11.3 Making curried functions

Back in the chapter 4, we have talked about currying and how we could use it to improve APIs in our projects. We mentioned then that we were going to implement a generic function that turns any callable into its curried version in this chapter.

Just as a reminder, currying allows us to treat multi-argument functions as unary functions. Instead of having a function that we can call with n arguments and it gives us a result, we have a unary function that returns another unary function, that returns yet another unary function, and so on until all n arguments are defined and the last function can give us the result.

Let's recall the example we used in chapter 4. We had a function `print_person` that

had three arguments — the person to print, output stream to print to and the format.

```
void print_person(const person_t& person,
                  std::ostream& out,
                  person_t::output_format_t format);
```

When we implemented the curried version by hand, it became a chain of nested lambdas where each lambda had to capture all the previously defined arguments:

```
auto print_person_cd(const person_t& person)
{
    return [&](std::ostream& out) {
        return [&](person_t::output_format_t format) {
            print_person(person, out, format);
        };
    };
}
```

The curried version required us to pass arguments one by one because, as we said, all curried functions are unary:

```
print_person_cd(martha)(std::cout)(person_t::full_name);
```

It can be a bit tedious to write all these parentheses, so we are going to relax this a bit. We will allow the user to specify multiple arguments at the same time. Mind that this is just a syntactic sugar, the curried function is still just a unary function, we are just making it a bit more convenient to use.

The curried function, needs to be a function object with state since it has to remember the original function and all the previously given arguments. It will store copies of all the captured arguments inside of a `std::tuple`. For this, we will need to use `std::decay_t` to make sure that the type parameters for the tuple are not references but actual value types.

```
template <typename Function, typename... CapturedArgs>
class curried {
private:
    using CapturedArgsTuple = std::tuple<
        std::decay_t<CapturedArgs>...>;
    template <typename... Args>
    static auto capture_by_copy(Args&&... args)
    {
        return std::tuple<std::decay_t<Args>...>(
            std::forward<Args>(args)...);
    }

public:
    curried(Function function, CapturedArgs... args)
```

```

        : m_function(function)
        , m_captured(capture_by_copy(std::move(args)...)))
    {}

    curried(Function function, std::tuple<CapturedArgs...> args)
        : m_function(function)
        , m_captured(std::move(args))
    {}

    ...
}

private:
    Function m_function;
    std::tuple<CapturedArgs...> m_captured;
};

```

So far, we have a class that is able to store a callable object and an arbitrary number of function arguments. The only thing left to do is to make this class a function object — to add the call operator.

The call operator needs to cover two separate cases:

- the user provided all remaining arguments for us to be able to call the original function in which case we should call it and return the result, and
- we still do not have all the arguments we need in order to call the function, so we can just return a new curried function object.

In order to test whether we have a sufficient number of arguments or not, we are going to use the `std::is_invocable_v` meta-function. It accepts a callable object type, a list of argument types, and returns whether that object can be invoked with arguments of the specified types.

To check whether `Function` can be called only with the arguments we captured so far, we can write the following:

```
std::is_invocable_v<Function, CapturedArgs...>
```

In the call operator, we will need to check whether the function is callable not only with the captured arguments, but also with the newly defined ones. We will have to use the `constexpr-if` here because the call operator can return different types depending on whether it returns the result or a new instance of the curried function object.

```

template <typename... NewArgs>
auto operator()(NewArgs&&... args) const
{
    auto new_args = capture_by_copy(std::forward<NewArgs>(args)...);

    if constexpr(std::is_invocable_v<

```

```
        Function, CapturedArgs..., NewArgs...>) {
    ...
    } else {
        ...
    }
}
```

In the **else** branch, we just need to return a new instance of `curried` that contains the same function that the current instance does, but with newly specified arguments added to the tuple `m_captured`.

11.3.1 Calling all callables

As for the **then** branch, it needs to evaluate the function for the given arguments. The usual way to call functions is with the regular call syntax, so we might want to try to do this as well.

The problem is that there are a few things in C++ that look like functions but can not be called as such — pointers to member functions and member variables. When we have a type like `person_t` with a member function `name`, we can get a pointer to that member function with `&person_t::name`. But we can not call this pointer to a function like we can do with pointers to normal functions — we would get a compiler error:

```
&person_t::name(martha);
```

This is an unfortunate limitation of the C++ core language. Every member function is just like an ordinary function, where the first argument is the implicit `this` pointer. But still, we can not call it as a function.

The same goes for member variables — they can be seen as functions that take an instance of a class and return a value stored in its member variable.

Because of this language limitation, it is not easy to write generic code that can work with all callables — both with function objects and pointers to member functions and variables.

For this reason, the `std::invoke` was added to the standard library — as a remedy for the limitation of the language. With `std::invoke`, we can call any callable object regardless of whether it allows the usual call syntax or not. While the previous snippet would produce a compilation error, the following will compile and do exactly what is expected:

```
std::invoke(&person_t::name, martha);
```

The syntax for `std::invoke` is simple — the first argument is the callable object, and it is followed by the arguments that will be passed to that callable object.

Listing 11.6. Common call syntax with std::invoke for various callables

```
std::less<>(12, 14)           std::invoke(std::less<>, 12, 14)
fmin(42, 6)                  std::invoke(fmin, 42, 6)
martha.name()                std::invoke(&person_t::name, martha)
pair.first                   std::invoke(&pair<int,int>::first, pair)
```

Using `std::invoke` makes sense only in generic code — when we do not exactly know the type of the callable object. This means that every time we implement a higher-order function that takes another function as its argument, or when we have a class like `curried` which stores a callable of an arbitrary type, we should not use the normal function call syntax, but we should use the `std::invoke` instead.

Now, for `curried`, we can not really use `std::invoke` directly because we have a callable and a `std::tuple` containing the arguments we need to pass to the callable — we do not have the individual arguments.

We will use a helper function called `std::apply` instead. It behaves similarly to `std::invoke` (and is usually implemented using `std::invoke`) with just a slight difference — instead of accepting individual arguments, it accepts a tuple containing the arguments — exactly what we need.

Listing 11.7. Complete implementation of curried

```
template <typename Function, typename... CapturedArgs>
class curried {
private:
    using CapturedArgsTuple =
        std::tuple<std::decay_t<CapturedArgs>...>;
    template <typename... Args>
    static auto capture_by_copy(Args&&... args)
    {
        return std::tuple<std::decay_t<Args>...>(
            std::forward<Args>(args)...);
    }
public:
    curried(Function function, CapturedArgs... args)
        : m_function(function)
        , m_captured(capture_by_copy(std::move(args)...))
    {}
    curried(Function function,
            std::tuple<CapturedArgs...> args)
        : m_function(function)
        , m_captured(std::move(args))
    {}
}
```

```

template <typename... NewArgs>
auto operator()(NewArgs&&... args) const
{
    auto new_args = capture_by_copy(          1
        std::forward<NewArgs>(args)...);      1

    auto all_args = std::tuple_cat(           2
        m_captured, std::move(new_args));      2

    if constexpr(std::is_invocable_v<Function,     3
                CapturedArgs..., NewArgs...>) {      3
        3

        return std::apply(m_function, all_args);  3

    } else {                                4
        return curried<Function,             4
                    CapturedArgs...,           4
                    NewArgs...>(             4
            m_function, all_args);          4
    }
}

private:
    Function m_function;
    std::tuple<CapturedArgs...> m_captured;
};

```

- 1 Creating a tuple out of the new arguments
- 2 Concatenating the previously collected arguments with the new ones
- 3 If we can call `m_function` with the given arguments, let's do so
- 4 Otherwise, just return a new `curried` instance with all the arguments we got so far stored inside

An important thing to note is that the `call` operator returns different types depending on the branch of the `constexpr-if` which was taken.

We can now easily create a curried version of `print_person` like so:

```
auto print_person_cd = curried{print_person};
```

Since we are storing the arguments by value, if we want to pass a non-copyable object (like an output stream) when calling the curried function, or if we want to avoid copying for performance reasons (like copying a `person_t` instance), we can pass the argument inside of an reference wrapper:

```
print_person_cd(std::cref(martha))(std::ref(std::cout))(person_t::name_only);
```

This will call the `print_person` function passing it a `const` reference to `martha`, a mutable reference to `std::cout` and `person_t::name_only` will be passed by value.

It is important to note that this implementation of currying works for ordinary

functions, for pointers to member functions, for normal and generic lambdas, for classes with the call operator — both generic and not, and even for classes with multiple different overloads of the call operator.

11.4 DSL building blocks

Until now, we have mostly been focussed on writing generally useful utilities which make our code shorter and safer. Sometimes, these utilities are overly generic, while we might require something more specific.

We will sometimes notice patterns in our project, things that we do over and over again just with small differences. But things that do not look overly generic to warrant creating a library like ranges which the whole world would find useful. Sometimes, the problems we are solving might seem overly domain specific.

Still, following the **don't-repeat-yourself** mantra, we should do something about that.

Imagine the following scenario — we have a set of records for which when we update them, we need to update all fields of a record in a single transaction. If any of the updates fail, the record must remain unchanged. We could implement a transaction system and put **start-transaction** and **end-transaction** all over the code. This is error-prone — we could forget to end a transaction, we could even accidentally change a record without starting the transaction at all.

It would be much nicer if we could create a more convenient syntax for this, which will be less error-prone and which would allow us not to think about transactions at all.

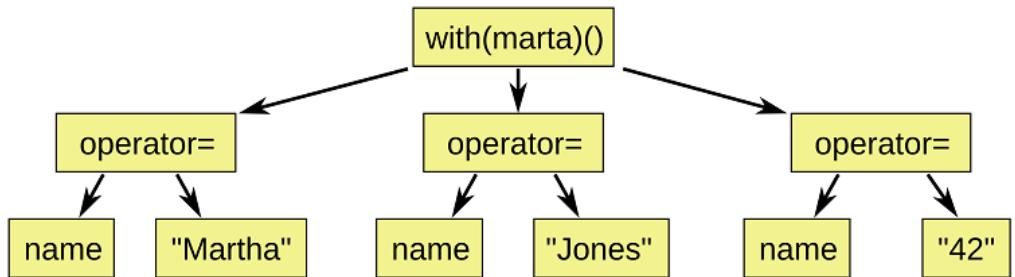
This is a perfect case for creating a small DSL — a domain-specific language. The language needs only to allow us to define the record updates in a nice way, and nothing else. It does not need to be generic, it is meant to be used only in this small domain — domain of transactional record updates. When we define what needs to be updated, the implementation of the DSL needs to handle the transaction itself.

We might want to create something that looks like this:

```
with(martha) (
    name = "Martha",
    surname = "Jones",
    age = 42
);
```

It is obvious that this is not **normal C++** — no curly braces, no semicolons. But it still can be valid C++ if we are willing to spend some time to implement the DSL. The implementation will not be pretty, but the point is to hide the complexity from the main code — to make writing the main program logic as easy as possible, while sacrificing the **under-the-hood** parts that most people will never see.

Figure 11.3. The abstract syntax tree that we want to modl contains three levels – the object we are updating, a list of updates that need to be performed, where each update contains two items – the field that should be updated and the new value that the field should have



Let's investigate the syntax in the example above:

- We have a function (or a type) called `with` — we know it is a function because we are calling it with the argument `marta`;
- The result of this call is another function which needs to accept an arbitrary number of arguments — we might need to update more fields at the same time;
- We also have entities `name` and `surname` which have the assignment operator defined on them;
- The results of these assignments are then passed to the function returned by `with(marta)`.

When implementing a DSL like this, it is the best approach to start from the innermost elements. And create all the types needed to represent an abstract syntax tree (AST) of the syntax we want to implement.

In this case, we need to start from the `name` and `surname` entities. Obviously, they are meant to represent the members of the person record. When we want to change a class member, we either need to have that member declared as `public`, or we need to go through a setter member function.

So, we need to create a simple structure that can store either a pointer to a member, or a pointer to a member function. We can do this simply by creating a dummy structure `field` which will be able to hold anything:

```

template <typename Member>
struct field {
    field(Member member)
        : member(member)
    {
    }

    Member member;
};
  
```

With this, we can provide fields for our types. You can see how to define the fields

for a type in the accompanying code example 11-ds1.

When we have the AST node to hold a pointer to the member or a setter member function, we can move on to implementing the syntax it needs to support. From the example above, we see that the `field` needs to have the assignment operator defined on it. Unlike a regular assignment operator, this one can not actually change any data — it only needs to return another AST node which we will name `update` because it defines one update operation. This node will be able to store the member pointer and the new value:

```
template <typename Member, typename Value>
struct update {
    update(Member member, Value value)
        : member(member)
        , value(value)
    {
    }

    Member member;
    Value value;
};

template <typename Member>
struct field {
    ...

    template <typename Value>
    update<Member, Value> operator=(const Value& value) const
    {
        return update{member, value};
    }
};
```

Now that we have the `update` node, we are only left with the main node — the `with` function. It takes an instance of a `person` record, and returns a function object representing a transaction which accepts a list of updates that need to be performed. Therefore, we will call this function object simply `transaction`. It will store a reference to the record, so that it can change the original one, and the list of `update` instances be passed to the call operator of `transaction`. The call operator will return a Boolean value indicating whether the transaction was successful or not:

```
template <typename Record>
class transaction {
public:
    transaction(Record& record)
        : m_record(record)
    {
    }
```

```

template <typename... Updates>
bool operator()(Updates... updates)
{
    ...
}

private:
    Record& m_record;
};

template <typename Record>
auto with(Record& record)
{
    return transaction(record);
}

```

Now we have all the AST nodes that we need which means that now we only need to implement the DSL behaviour.

Now we need to consider what a transaction means to us. If we are working with a database, we would need to start the transaction and commit it when all updates are processed. If we just need to send all updates to our records over the network in order to keep the distributed data synchronized, we could just wait for all the updates to be applied, and then send the new record over the network.

In order to keep the things simple, we will consider the actual C++ structure members update as a transaction. If an exception occurs while changing the data, or if a setter function returns `false`, we will consider that the update failed, and we will cancel the transaction.

The easiest way to implement this is the **copy-and-swap** idiom we saw in chapter 9. We will create a copy of the current record, perform all the changes on it, and swap with the original record if all updates were successful.

Listing 11.8. Implementation of the call operator for the transaction

```

template <typename Record>
class transaction {
public:
    transaction(Record& record)
        : m_record(record)
    {}

    template <typename... Updates>
    bool operator()(Updates... updates)
    {
        auto temp = m_record;           ①

        if (all(updates(temp)...)) {   ②
            std::swap(m_record, temp); ②
        }
    }
}

```

```

        return true;          ②
    }

    return false;
}

private:
    template <typename... Updates>
    bool all(Updates... results) const ③
    {
        return (... && results);      ③
    }

    Record &m_record;
};
```

- ① Creating a temporary copy to perform updates on
- ② Applying all the updates. If all updates are successful, we will swap the temporary copy with the original record and return true
- ③ Collect all the results of different updates, and return true if all of them succeeded

We are calling all updates on the temporary copy. If any of the updates returns `false` or throws an exception, the original record will remain unchanged. The only thing left to implement is the call operator for the `update` node. For it, we will have three different cases:

- We have a pointer to a member variable which we can change directly;
- We have an ordinary setter function;
- We have a setter function that returns a `bool` value indicating whether the update succeeded or not.

We have seen that we can use `std::is_invocable` to test whether a given function can be called with a specific set of arguments which we can use to check whether we have a setter function or a pointer to a member variable. The novelty here is that we also want to differentiate between setters that return `void` and those that return `bool` (or another type convertible to `bool`). We can do this with `std::is_invocable_r` which check both whether the function can be called, and whether it will return a desired type.

Listing 11.9. Full implementation of the update structure

```
template <typename Member, typename Value>
struct update {
    update(Member member, Value value)
        : member(member)
        , value(value)
    { }
```

```

template <typename Record>
bool operator()(Record& record)
{
    if constexpr (std::is_invocable_r<
        bool, Member, Record, Value>()) { ①
        return std::invoke(member, record, value); ①

    } else if constexpr (std::is_invocable<
        Member, Record, Value>()) { ②
        std::invoke(member, record, value); ②
        return true; ②

    } else {
        std::invoke(member, record) = value; ③
        return true; ③
    }
}

Member member;
Value value;
};

```

- ① If the Member callable object returns a `bool` when we pass it a record and a new value, this means we have a setter function that might fail
- ② If the result type is not `bool` or convertible to `bool`, we are just invoking the setter function and returning `true`
- ③ If we have a pointer to a member variable, just set it, and return `true`

C++ brings a lot to the table when implementing DSLs. Operator overloading and variadic templates being the main two. With these, we can develop quite complex DSLs.

The main problem is that implementing all the necessary structures to represent AST nodes can be a very tedious work and requires a lot of boiler-plate code.

While this is a significant down-side which makes DSLs in C++ not as popular as they might be, DSLs have two huge benefits. The first one being able to write short and concise main program logic. The second benefit is that we can switch between different meanings of transactions without changing a single line of the main program logic.

If we decided to serialize all our records to a database, we would just need to reimplement the call operator of the `transaction` class, and the rest of the program will suddenly start saving data to the database without changing a single line of the main program logic.

11.5 Summary

- Templates give us a Turing-complete programming language that gets executed during the program compilation. This was discovered accidentally by Erwin

Unruh who created a C++ program which prints first 10 prime numbers during compilation — as compilation errors;

- TMP is not only a Turing-complete language — it is also a pure functional language. All **variables** are immutable, and there is no mutable state in any form;
- The `type_traits` header contains many useful meta-functions for type manipulation;
- It happens from time to time that due to the limitations or missing features in the core programming language for the work-arounds to be added to the standard library. One of such cases is `std::invokewhich` allows us to call all function-like objects, even those that do not support the regular function call syntax;
- DSLs are tedious to write, but they allow us to significantly simplify the main program logic. After all, ranges are also in a sense a DSL — they define an AST for defining range transformations using the pipe syntax.

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch11

12

Functional design for concurrent systems

This chapter covers:

- Splitting the software into isolated components
- Treating messages as streams of data
- Transformation of reactive streams
- Stateful software components
- Benefits of reactive streams in concurrent and distributed system design

The biggest problem in software development is handling complexity. Software systems tend to grow significantly over time and they quickly outgrow the original designs. When it turns out that the features we need to implement collide with the design, we must either reimplement significant portions of the system, or we introduce horrible quick-and-dirty hacks to make things work.

This problem with complexity becomes more evident in software which has different parts that execute concurrently—from the simplest interactive user applications, to network services and distributed software systems.

*A large fraction of the flaws in software development are due to programmers not fully understanding all the **possible states** their code may execute in. In a multithreaded environment, the lack of understanding and the resulting problems are **greatly amplified**, almost to the point of panic if you are paying attention.*

-- John Carmack, - In-depth: Functional programming in C++

Most of the problems we have come from the entanglement of different system components. Having separate components that access and mutate the same data requires synchronizing said access. This synchronization is traditionally handled with mutexes or similar synchronization primitives, which works, but it introduces significant scalability problems and it kills concurrency.

One approach to solving the problem of shared mutable data is not to have any mutable data whatsoever. But we also have another option — to have mutable data, but never to share it. We focussed on the former in the chapter 5, we are now going to talk about the latter.

12.1 The actor model — thinking in components

In this chapter, we will see how to design the software as a set of isolated separate components. We will first need to discuss this in the context of object-oriented design in order to later see how we can make it functional.

When designing classes, we tend to create getter and setter functions — getters to retrieve information about an object, and setters to change attributes of an object in a valid way which will not violate the class invariants.

Many object-oriented design proponents consider this approach to be contrary to the philosophy of OO. They tend to call it **procedural** programming because we still think in algorithm steps, and the objects serve just as containers and validators for the data.

Step one in the transformation of a successful procedural developer into a successful object developer is a lobotomy.

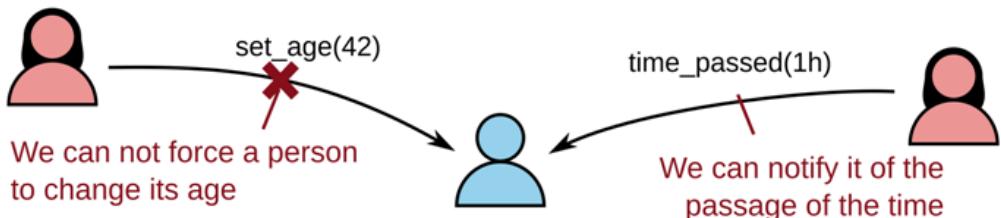
-- David West, - Object Thinking

Instead, we should stop thinking about what data an object contains, and instead think about what it can do. As an example, consider a class which represents a person. The way we would usually implement it is to create getters and setters for the name, surname, age and other attributes. Then, we could do something like:

```
douglas.set_age(42);
```

And this shows the problem. We have designed a class to be a data container, instead of designing it to behave like a person. Can we force a human being to be 42 years old? We can't. We can't change the age of a person, and we shouldn't design our classes to allow us to do so.

Figure 12.1. We can not set the attributes on real-life objects. We can send them messages, and let them react to them.



We should design the classes to have a set of actions or tasks they can perform, and then add the data necessary to implement those actions. In the case of the class which models a person, instead of having a setter for the age, we would need to create an action which will tell the person that some time has passed, and the object should react appropriately. Instead of `set_age`, the object could have a member function `time_passed` like so:

```
void time_passed(const std::chrono::duration& time);
```

When notified that the specified time has passed, the person object will be able to increase its age, but also to perform other related changes. For example, if this is relevant to our system, the person's height could be changed, the hair color etc. as the result of the person getting older. Therefore, instead of having getters and setters, we should only have a set of tasks that the object knows how to perform.

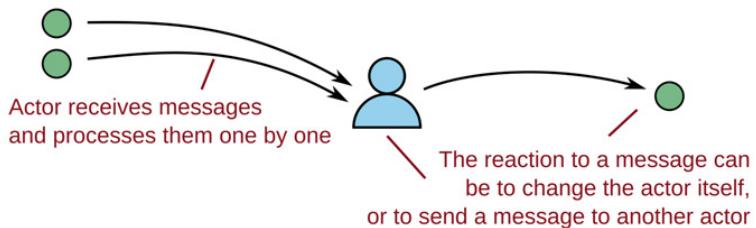
Don't ask for the information you need to do the work; ask the object that has the information to do the work for you.

-- Allen Holub

If we continue to model the person object after real-life people, we will also come to a realization that multiple person objects should not have any shared data. Real people **share** data by talking to each other, they do not have shared variables which everyone can access and change.

This is exactly the idea behind actors. In the actor model, actors are completely isolated entities which share nothing, but which can send messages to one another. The minimum that an actor class should have is a way to receive and send messages.

Figure 12.2. An actor is an isolated component that can receive and send messages. It processes the messages serially, and for each message it can change its state or behaviour, or it can send a new message to another actor in the system



Traditionally, actors can send and receive different types of messages, and each actor can choose which actor to send the message to. Also, the communication should be asynchronous.

C++ Actor Framework

You can find a complete implementation of the traditional actor model for C++ at actor-framework.org/ which you can use in your projects.

The C++ Actor Framework has an impressive set of features. The actors are lightweight concurrent processes (much more lightweight than threads) and it is network-transparent, meaning that you can distribute your actors over multiple separate machines and the things will just work without the need to change your code.

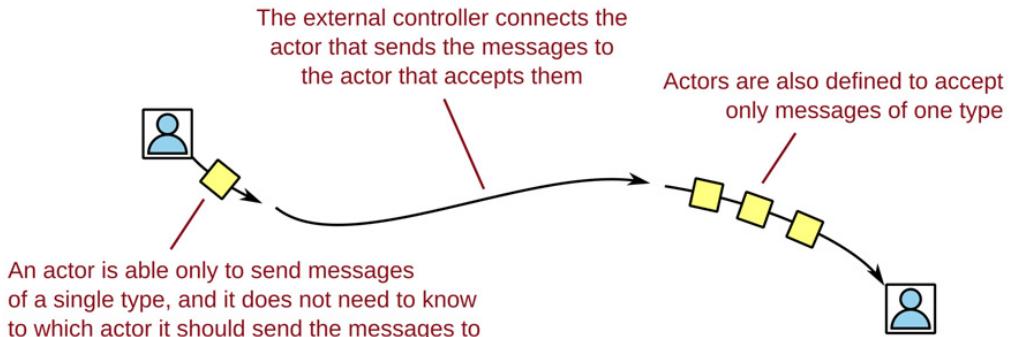
While traditional actors are not easily composed, they can easily be made to fit into the design we will cover in this chapter.

We are going to define a more rudimentary actor compared to actors as specified by the actor model and actors in the C++ Actor Framework because we want to focus more on the software design than on implementing a true actor model. While the design of actors presented in this chapter differs from the design of actors in the traditional actor model, the concepts that will be presented are applicable even with the traditional actors. We are going to design our actors:

- To be typed actors — actors that can receive only messages of a single type, and send messages of a single (not necessarily the same) type. If we have the need to support multiple different types for input or output messages, we can use `std::variant` or `std::any` as we have seen in chapter 9
- Instead of allowing each actor to choose to whom to send a message, we will leave this choice to an external controller which will allow us to compose the actors in a functional way. The external controller will schedule which sources an actor should listen to
- We will leave it up to the external controller to decide which messages should be

processed asynchronously and which not

Figure 12.3. We will use simplified typed actors that do not need to care about who sends the message to whom because that will be left to an external controller



Most software systems nowadays use or implement some kind of event loop which can be used to asynchronously deliver messages, so we won't concern ourselves here with implementing such a system — we will focus on the software design which can easily be adapted to work on any event-based system.

Listing 12.1. The minimal interface for an actor

```
template <typename SourceMessageType,
         typename MessageType>           ①
class actor {                         ①
public:
    using value_type = MessageType;   ④
    void process_message(SourceMessageType&& message); ②
    template <typename EmitFunction>
    void set_message_handler(EmitFunction emit);          ③
private:
    std::function<void(MessageType&&)> m_emit;        ③
};
```

- ① An actor can receive messages of one type, and send messages of another
- ② This function handles when a new message arrives
- ③ Sets the `m_emit` handler which the actor will call when it wants to send a message
- ④ Defines the type of the message the actor is sending, so that we can check it later when we need to connect the actors to one another

With this interface, we are clearly stating that an actor knows only how to receive a message, and how to send a message onwards. It can have as many private data as it needs to perform its work, but none of it should ever be available to the outside

world. Since the data can not be shared, we have no need to have any synchronization on it.

It is important to note that there can be actors that just receive messages (usually called **sinks**), actors that just send messages (usually called **sources**) and general actors that do both.

12.2 Creating a simple message source

In this chapter, we are going to create a small web service that receives and processes bookmarks (example:bookmarks-service). The clients will be able to connect to it, and send JSON input which defines a bookmark like this:

```
{ "FirstURL": "https://isocpp.org/", "Text": "Standard C++" }
```

For this, we will need a few external libraries. For network communication, we will be using the Boost.ASIO library¹⁵, and for working with JSON, we will be using Lohmann's JSON library¹⁶.

We'll first create an actor that listens for incoming network connections, and collect the messages that the clients send to the service. In order to make the protocol as simple as possible, messages will be line-based, that is, each message needs to have only one newline character in it — at the end of the message.

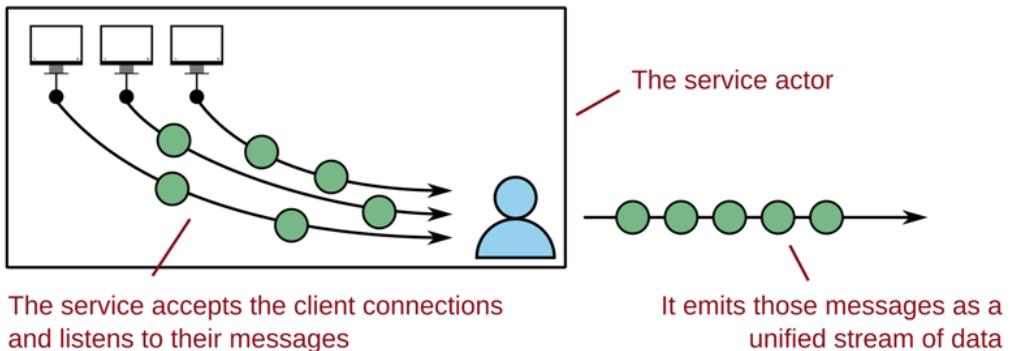
This actor is the source actor — it receives messages from the outside world (entities that are not a part of our service), and it is the source of those messages as far as our service is concerned. The rest of the system does not need to be concerned about where the messages are coming from, so we can consider the client connections to be an integral part of this source actor.

The service will have a similar interface to actor, just with an exception that we do not need the `process_message` function because this is a source actor — it does not receive messages from other actors in the system (as we said, it gets the messages from external entities — the clients — which we do not consider to be actors in our service) — it just sends them.

¹⁵ Boost.ASIO — www.boost.org/doc/libs/1_64_0/doc/html/boost_asio.html

¹⁶ JSON for Modern C++ — github.com/nlohmann/json

Figure 12.4. The service actor listens for the client connections and for their messages. This part is hidden from the rest of the program — the rest just knows that a stream of strings is arriving from somewhere



Listing 12.2. Service to listen for the client connections

```

class service {
public:
    using value_type = std::string;           ①

    explicit service(
        boost::asio::io_service& service,
        unsigned short port = 42042)
        : m_acceptor(service,           ②
                      tcp::endpoint(tcp::v4(), port)) ②
        , m_socket(service)           ②
    {
    }

    service(const service& other) = delete;   ③
    service(service&& other) = default;       ③

    template <typename EmitFunction>
    void set_message_handler(EmitFunction emit) ④
    {
        m_emit = emit;
        do_accept();                           ④
    }

private:
    void do_accept()
    {
        m_acceptor.async_accept(
            m_socket,
            [this](const error_code& error) {
                if (!error) {
                    make_shared_session(      ⑤
                        std::move(m_socket)), ⑤
                }
            });
    }
}

```

```

        m_emit
    )->start();

} else {
    std::cerr << error.message() << std::endl;
}

// Listening to another client
do_accept();
});

}

tcp::acceptor m_acceptor;
tcp::socket m_socket;
std::function<void(std::string&&)> m_emit;
};

```

- ➊ We are reading the client input line-by-line, so the messages we will be sending are strings
- ➋ Creating the service which listens at the specified port (by default 42042)
- ➌ Disabling copying, but we still want to allow moves
- ➍ There is no point in accepting the connections from the clients until someone registers to listen to the messages from `message_service`
- ➎ Creating and starting the session for the incoming client. Whenever the session object reads a message from a client, it will be passed to `m_emit`. The `make_shared_session` creates a shared pointer to a session object instance.

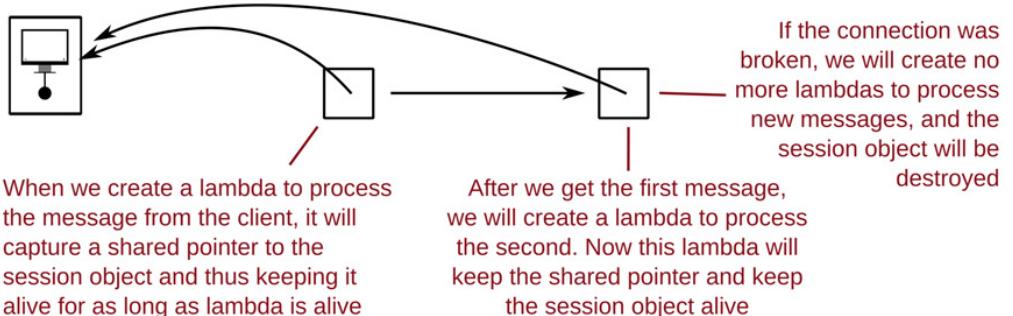
The service itself is mostly easy to understand. The only part that is a bit more difficult is the `do_accept` function due to the Boost.ASIO callback-based API. In a nutshell, it does the following:

- `m_acceptor.async_accept` schedules the lambda passed to it to be executed when a new client appears
- The lambda checks whether the client has connected successfully, and if so, it creates a new session for it.
- We want to be able to accept multiple clients, so we are calling `do_accept` again

The session object will do the most of the work here. It needs to read the messages from the client one by one, and notify the service of them, so that the service can act as the source of messages for the rest of the program.

The session object also needs to keep its own lifetime. As soon as there is an error, and the client disconnects, the session should destroy itself. We will use a trick here where the session object will inherit from `std::enable_shared_from_this`. This will allow a session instance which is managed by a `std::shared_ptr` to safely create additional shared pointers to itself.

Figure 12.5. We capture the shared pointer to the session object inside of the lambda that will process when new data arrives from the client. This means that the session object will be kept alive for as long as the client keeps the connection alive



Having a shared pointer to the session allows us to keep the session object alive for as long as there are parts of the system that use the session. We will capture the shared pointer to the session in the lambdas that process connection events. As long as there is an event the session is waiting for, the session object will not be deleted because the lambda that handles that event will hold an instance of the shared pointer. When there are no more events we want to process, the object will be deleted.

Listing 12.3. The session object reads the messages and emits them

```
template <typename EmitFunction>
class session {
public:
    session(tcp::socket&& socket, EmitFunction emit)
        : m_socket(std::move(socket))
        , m_emit(emit)
    {
    }

    void start()
    {
        do_read();
    }

private:
    using shared_session =
        std::enable_shared_from_this<session<EmitFunction>>;
    void do_read()
    {
        auto self = shared_session::shared_from_this(); ①
        boost::asio::async_read_until( ②

```

```

        m_socket, m_data, '\n',
        [this, self](const error_code& error,
                     std::size_t size) {

            if (!error) {
                std::istream is(&m_data);
                std::string line;
                std::getline(is, line);
                m_emit(std::move(line));
            }

            do_read(); ④
        }
    });

tcp::socket m_socket;
boost::asio::streambuf m_data;
EmitFunction m_emit;
};
```

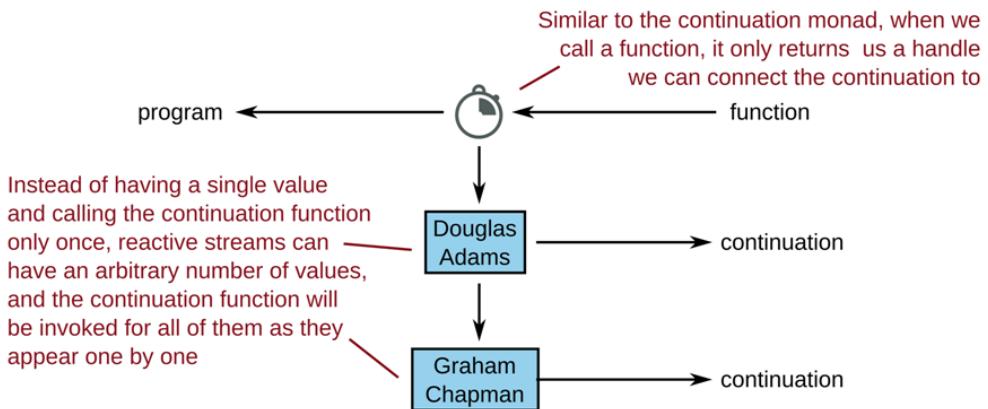
- ① Creating another pointer that has the shared ownership of this session
- ② Scheduling a lambda to be executed when we reach the newline character in the input
- ③ We should either have encountered an error, or we can read the line and send it to whoever registered to listen for messages
- ④ If we have read the message successfully, we want to schedule to read the next one.

While the users of the service class will not even know that the `session` object exists, it is the one that will actually send the messages to them.

12.3 Modelling reactive streams as monads

We have created a service that emits messages of type `std::string`. It can emit as many messages as it wants — zero or more. This kinda looks like a singly linked list — we have a collection of values of some type, and we can traverse it element by element until we reach the end. The only difference is that with lists we already have all the values to traverse over, whereas in this case the values are not yet known — they arrive from time to time.

Figure 12.6. Unlike the continuation monad which calls the continuation function only once, reactive streams can have an arbitrary number of values, and the continuation function will be called for each new value that arrives



If you recall, we had something similar in chapter 10. We had futures and the continuation monad. The future was a container-like structure that will contain a value of a given type at some point in time. Our service is similar, with a difference that it is not limited to a single value, but it sends new values from time to time. We will call structures like these asynchronous or **reactive streams**.

It is important to note that this is not the same as `future<list<T>>` — that would mean that we will get all values in a single point in time.

Reactive streams look like collections. They contain items of the same type, it is just that not all items are available at once. We have already seen a type that behaved in a similar way in chapter 7 — the input stream. We were able to use the input stream with the ranges library, and perform transformations on it:

```
auto words = istream_range<std::string>(std::cin)
    | view::transform(string_to_lower);
```

We were able to create the same transformations on futures and optionals. You probably see where I'm going with this — we were able to create the same transformations for all monads that we covered. So, the important question here is whether reactive streams are a monad?

Conceptually, they seem to be. Let's recap what we need to have for something to be a monad:

- It needs to be a generic type;
- We need a constructor — a function which creates an instance of a reactive stream that will contain a given value;
- We need the transform function — a function that will return a reactive stream which will emit transformed values coming from the source stream;

- We need a `join` function which will take all messages from all given streams and emit them one by one;
- And we need for it to obey the monad laws (proving this is outside of the scope of this book).

The first item is satisfied — reactive streams are generic types parametrised on the type of the messages that the stream is emitting (`value_type`).

In the following sections, we will make reactive streams a monad by:

- Creating a stream transformation actor
- Creating an actor that creates a stream of given values
- Creating an actor that can listen to multiple streams at once and emit the messages coming from them.

12.3.1 Creating a sink to receive messages

Before implementing all the functions needed to show that reactive streams are a monad, let's first implement a simple sink object which will allow us to test our service.

The sink is an actor that only receives messages, but does not send them. This means that here we do not need the `set_message_handler` function, we just need to define what `process_message` does. We will create a generic sink which will execute any given function every time a new message appears.

Listing 12.4. The implementation of the sink object

```
namespace detail {
    template <typename Sender,
              typename Function,
              typename MessageType = typename Sender::value_type>
    class sink_impl {
public:
    sink_impl(Sender&& sender, Function function)
        : m_sender(std::move(sender))
        , m_function(function)
    {
        m_sender.set_message_handler(
            [this](MessageType&& message) ❶
            {
                process_message( ❷
                    std::move(message)); ❸
            }
        );
    }

    void process_message(MessageType&& message) const
    {
        std::invoke(m_function,
```

```

        std::move(message));
    }

private:
    Sender m_sender;
    Function m_function;
};

}

```

- ① When the sink is constructed, it will connect to its assigned sender automatically
- ② When we get a message, we just pass it on to the function defined by the user

Note **Single-owner design**

One of the things you will notice in this chapter is that `std::move` and rvalue references are used often.

For example, the `sink_impl` takes the `sender` object as an rvalue reference. This means that the `sink` object will become the sole owner of the `sender` we assign to it. In a similar manner, other actors will become owners of their respective senders.

This implies that the data flow pipeline will own all the actors in it, and when the pipeline is destroyed, all the actors will be as well.

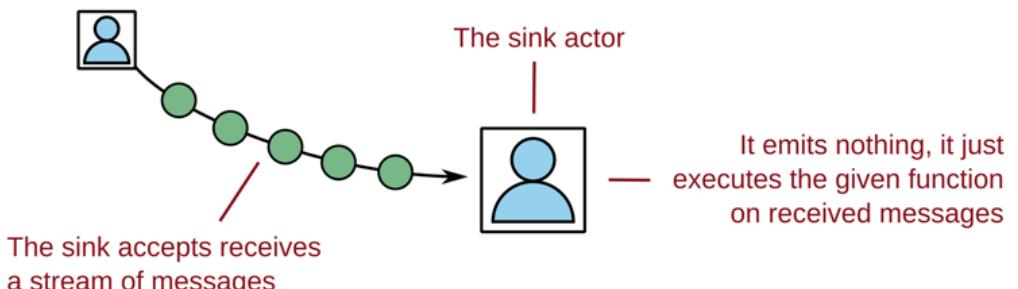
Also, the messages will be passed on through different actors using rvalue references to indicate that only one actor has the access to a message in one point in time.

This design is simple and easy to reason about, which is why I think it is the best approach to demonstrate the concept of actors and the data flow design.

The downside is that we can not have multiple components in our system listen to a single actor, nor we can share actors between different data flows in our system.

This can be easily fixed by allowing shared ownership of actors (`std::shared_ptr` would be a perfect choice for this), and allowing each sender to have multiple listeners by keeping a collection of message handler functions instead of having only one.

Figure 12.7. The sink actor just calls a function for each message it receives. It emits nothing. We can use it to print out all the messages we receive to `std::cerr` or something more advanced like saving the messages to a file or a database



Now we just need to create a function that creates an instance of `sink_impl` given a

sender and a function:

```
template <typename Sender, typename Function>
auto sink(Sender&& sender, Function&& function)
{
    return detail::sink_impl<Sender, Function>(
        std::forward<Sender>(sender),
        std::forward<Function>(function));
}
```

We can now easily test whether the service object works simply by connecting it to a sink that writes all messages to `cerr`.

Listing 12.5. Starting the service

```
int main(int argc, char* argv[])
{
    boost::asio::io_service event_loop; ①

    auto pipeline =
        sink(service(event_loop),
            [] (const auto& message) { ②
                std::cerr << message << std::endl; ②
            });
    event_loop.run(); ③
}
```

- ① `io_service` is a class in Boost.ASIO which handles the event loop — it will listen to events and call the appropriate callback lambdas for those events
- ② Creating the service and connecting the sink to it
- ③ Starting to process events

C++17 and class template deduction

C++17, which is required to compile examples in this chapter, supports class template deduction. This means that it was not strictly necessary to create the `sink` function — we could have just named the class itself `sink` and the code above would still work.

The reason for separating the `sink` and `sink_impl` is to be able to support the range-like syntax on reactive streams — we will have two different `sink` functions that return different types depending on the number of arguments passed to it. This would be more difficult to achieve if `sink` were not a proper function.

This looks similar to calling the `for_each` algorithm on a collection — we pass it a collection, and a function that gets executed for each item of that collection. This syntax is a bit awkward, so we are going to replace it with the pipe-based notation — the same that the ranges library uses.

For this, we will need a `sink` function that takes only the function that will be invoked on each message, without taking the sender object as well. It just needs to return a temporary helper object holding that function. The instance of `sink_impl` class will be created by the pipe operator when we specify the sender. You can think of this as a partial function application — we bind the second argument, and leave the first one to be defined later. The only difference is that we specify the first argument using the pipe syntax instead of using the normal function call syntax we used with partial function application in chapter 4.

```
namespace detail {
    template <typename Function>
    struct sink_helper {
        Function function;
    };
}

template <typename Sender, typename Function>
auto operator|(Sender&& sender,
                 detail::sink_helper<Function> sink)
{
    return detail::sink_impl<Sender, Function>(
        std::forward<Sender>(sender), sink.function);
}
```

We will define an `operator|` for each of the transformations we create in a similar way to this one. Each of them will accept any sender as the first argument, and a `_helper` class that defines the transformation. Now we can make the main program more readable:

```
auto sink_to_cerr =
    sink([](const auto& message) {
        std::cerr << message << std::endl;
    });

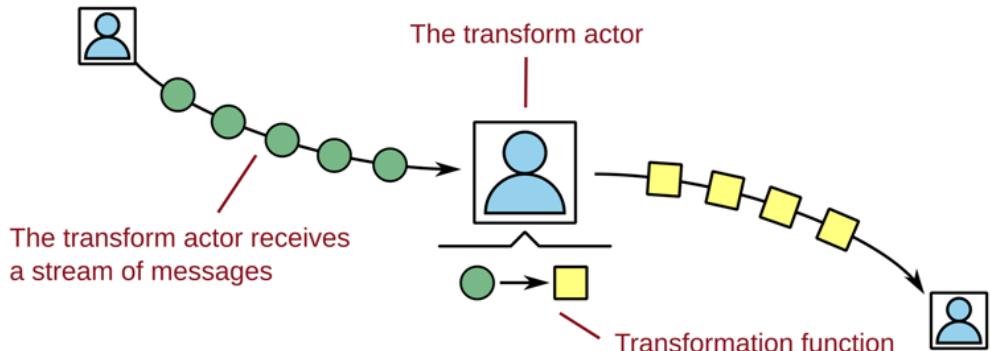
auto pipeline = service(event_loop) | sink_to_cerr;
```

Now we have a nice way to test whether the service works properly. Whatever stream we have, we can write all its messages to `cerr`. You can compile the program, start it and use `telnet` or a similar application with which you can test simple textual connections to connect to it on the port 42042. Each message any of the clients send will automatically appear on the output of the server.

12.3.2 Transforming reactive streams

We can now return to the task of making reactive streams a monad. The most important task is to create the `transform` stream modifier. It should take a reactive stream and an arbitrary function, and it needs to return a new stream — one that emits the messages from the original one, but transformed using the given function.

Figure 12.8. Similar to ranges, transform on reactive streams applies the given transformation function on each message it receives, and it sends the function result to the next actor



In other words, the `transform` modifier will be a proper actor which both receives and sends messages. For each message it receives, it will call the given transformation function, and emit the result it produces as a message.

Listing 12.6. Transform stream modifier implementation

```
namespace detail {
    template <
        typename Sender,
        typename Transformation,
        typename SourceMessageType =
            typename Sender::value_type,
        typename MessageType =
            decltype(std::declval<Transformation>()(
                std::declval<SourceMessageType>())))
    class transform_impl {
public:
    using value_type = MessageType;

    transform_impl(Sender&& sender, Transformation transformation)
        : m_sender(std::move(sender))
        , m_transformation(transformation)
    {
    }

    template <typename EmitFunction>
    void set_message_handler(EmitFunction emit)
    {
        m_emit = emit;
        m_sender.set_message_handler(
            [this](SourceMessageType& message) {
                process_message(
                    std::move(message));
            }
        );
    }
}
```

```

        });
    }

    void process_message(SourceMessageType&& message) const
    {
        m_emit(std::invoke(m_transformation,
                           std::move(message))); ③
    }

private:
    Sender m_sender;
    Transformation m_transformation;
    std::function<void(MessageType&&)> m_emit;
};

}

```

- ➊ In order to be able to define the message receiving and sending functions properly, we need the types of the messages that we receive, and the type of the messages that we send
- ➋ When we get an actor that is interested in our messages, we connect ourselves to the actor that will send us the messages
- ➌ When we receive a message, we transform it with the given function and send the result to the actor that listens to us

One thing worth noting here is that unlike the `sink` actor, `transform` is not immediately connecting to its sender. If we do not have anyone to send messages to, there is no reason why we should process them at all. We will start listening to the messages coming from our sender only when the `set_message_handler` function is called — when we get someone to listen to our messages.

After we create all the helpers and the pipe operator, we will be able to use the `transform` modifier in the same way we would use it with ranges. For example, if we wanted to trim the messages before printing them out, we could do:

```

auto pipeline =
    service(event_loop)
    | transform(trim)
    | sink_to_cerr;

```

This is starting to look very similar to ranges. And that is the main point, reactive streams are meant to allow us to think about software as a collection of input streams, the transformations we need to perform on them, and where we should put the results — just like we could do with ranges.

12.3.3 Creating a stream of given values

The `transform` function made reactive streams a functor. In order to make it a proper monad, we need to have a way to construct the stream out of a value, and we need the `join` function.

First, let's create the easier of the two. Given a value, or a list of values (for

convenience), we want to create a stream that emits them. This is a stream that does not accept any messages, but only emits them, just like the service we saw before.

This means that we do not need to listen for messages coming from other actors. We need to allow the user to specify the values upon construction, store them, and when an actor starts listening to the messages from this class, to send the values to it.

```
template <typename T>
class values {
public:
    using value_type = T;

    explicit values(std::initializer_list<T> values)
        : m_values(values)
    {}

    template <typename EmitFunction>
    void set_message_handler(EmitFunction emit)
    {
        m_emit = emit;
        std::for_each(m_values.cbegin(), m_values.cend(),
                      [&](T value) { m_emit(std::move(value)); });
    }

private:
    std::vector<T> m_values;
    std::function<void(T&>) m_emit;
};
```

This class can be used as a monad constructor for reactive streams. We can easily test whether this works by passing the values directly to a `sink` object:

```
auto pipeline = values{42} | sink_to_cerr;
```

This will create a stream containing only a single value, and when we connect `sink_to_cerr` to it, it will print that value to `std::cerr`.

12.3.4 Joining a stream of streams

The last thing we need to do in order to make reactive streams a monad is to define a `join` function.

Say we want to have a few different ports that we want our service to listen on. We would like to create a `service` instance for each port, and then join the messages from all those instances into a single unified stream.

We want to be able to do something like this:

```

auto pipeline =
    values{42042, 42043, 42044}
| transform([&](int port) {
    return service(event_loop, port);
})
| join()
| sink_to_cerr;

```

This might seem like a difficult thing to do, but with all things that we have already learnt, this becomes rather easy to implement. The implementation will be quite similar to `transform`. Both `join` and `transform` receive messages of one type, and emit messages of a different type. The only difference is that in the case of `join`, we are receiving messages that are in fact new streams. We just need to listen to messages from those streams, and then pass them on.

Listing 12.7. Implementing the join transformation

```

namespace detail {
    template <
        typename Sender,
        typename SourceMessageType =
            typename Sender::value_type,
        typename MessageType =
            typename SourceMessageType::value_type> ①
    class join_impl {
public:
    using value_type = MessageType;

    ...

    void process_message(SourceMessageType&& source)
    {
        m_sources.emplace_back(std::move(source)); ②
        m_sources.back().set_message_handler(m_emit); ③
    }

private:
    Sender m_sender;
    std::function<void(MessageType&&)> m_emit;
    std::list<SourceMessageType> m_sources; ④
};
}

```

- ① The `SourceMessageType` will be the type of the streams that we need to listen to
- ② The `MessageType` is the type of the messages that will be sent by the streams we are listening to — the messages that we need to pass on to our listeners
- ③ When we get a new stream to listen to, we are storing it, and forwarding its messages as our own
- ④ We need to save all the streams we listen to in order to expand their life-times. We are using a `list` to minimize the number of reallocations.

Now that we have both `join` and `transform`, we can finally say that reactive streams are monads.

12.4 Filtering reactive streams

So far, we have shown that reactive streams, with the transformations we created are monads, and we have tried to make them look like ranges as much as possible. Let's implement another useful function that we used with ranges.

In the previous examples, we wrote all the messages coming from a client to the error output. Say we want to be able to ignore some of them. For example, we want to filter out empty messages and the messages starting with the pound symbol (#) because those messages represent comments in the data that the client is sending.

We want to be able to do something like this:

```
auto pipeline =
    service(event_loop)
    | transform(trim)
    | filter([](const std::string& message) {
        return message.length() > 0 &&
               message[0] != '#';
    })
    | sink_to_cerr;
```

For this, we need to create a new stream modifier similar to `transform`. It will receive messages, and emit only those that satisfy the given predicate. The main difference is that unlike `transform` and `join`, filtering listens for and emits the same type of messages.

Listing 12.8. Transformation actor which filters messages in a stream

```
template <typename Sender,
          typename Predicate,
          typename MessageType =
          typename Sender::value_type> ①
class filter_impl { ①
public:
    using value_type = MessageType;
    ...
    void process_message(MessageType&& message) const
    {
        if (std::invoke(m_predicate, message)) { ②
            m_emit(std::move(message)); ②
        }
    }
private:
```

```

    Sender m_sender;
    Predicate m_predicate;
    std::function<void(MessageType&&)> m_emit;
};

```

- ➊ We are receiving the same type of message that we are sending
- ➋ Every time we receive a message, we will check whether it satisfies the predicate, in which case we will pass it on

Filtering is useful whenever we want to discard some invalid data, or the data we have no interest in.

12.5 Error handling in reactive streams

Since we are trying to implement a web service that receives JSON-formatted messages, we need to be able to handle the parsing errors.

We have discussed a few ways to do error handling in a functional way in the chapters 9 and 10. We have `optional<T>` which we can leave empty when we want to denote an error in a computation. We can also use `expected<T, E>` for when we want to be able to tell which exact error has occurred.

The library we are using to handle JSON input is heavily exception-based. Because of this, we will use the `expected<T, E>` for error handling. The type `T` will be the type of the message we are sending, where `E` will be a pointer to an exception (`std::exception_ptr`). Every message will either contain a value, or a pointer to an exception.

In order to wrap the functions that throw exceptions into our code which uses `expected` for error handling, we will use the `mtry` function defined in chapter 10. As a quick reminder, `mtry` is a helper function with which we can convert functions that throw exceptions into functions that return an instance of `expected<T, std::exception_ptr>`. We can give any callable object to `mtry`, and it will be executed. If everything went well, we will get a value wrapped inside of an `expected` object. If an exception was thrown, we will get an `expected` object containing a pointer to said exception.

With `mtry`, we can wrap the `json::parse` function and use `transform` to parse all messages we receive from the client into JSON objects. We will get a stream of `expected_json` objects (`expected<json, std::exception_ptr>`):

Listing 12.9. Parsing strings into JSON objects

```

auto pipeline =
    service(event_loop)
    | transform(trim)
    | filter([](const std::string& message) {
        return message.length() > 0 &&
               message[0] != '#';
    });

```

```

        })
| transform([](const std::string& message) {    ①
    return mtry([&] {
        return json::parse(message);                ②
    });
})
| sink_to_cerr;

```

- ① For each string we receive, we will try to parse it. The result will either be a JSON object, or a pointer to an exception (or, specifically `expected<json, std::exception_ptr>`)

We now need to extract the data from each JSON object into a proper structure. We will define a structure to hold the URL and text of a bookmark, and we will create a function that takes a JSON object and gives us a bookmark if the object contains required data, or an error if it does not:

```

struct bookmark_t {
    std::string url;
    std::string text;
};

using expected_bookmark = expected<bookmark_t, std::exception_ptr>

expected_bookmark bookmark_from_json(const json& data)
{
    return mtry([&] {
        return bookmark_t{data.at("FirstURL"), data.at("Text")};
    });
}

```

The JSON library will throw an exception if we try to access something with the `at` function and it is not found. Because of this, we need to wrap it into `mtry` just like we did with `json::parse`. Now we can continue processing the messages.

So far, we have parsed the strings and got the `expected<json, ...>`. We need to skip the invalid ones, and just try to create the `bookmark_t` values from the valid JSON objects. Further, since the conversion to `bookmark_t` can fail, we also need to skip all the failed values. We can use a combination of `transform` and `filter` for this.

```

auto pipeline =
    service(event_loop)
    | transform(trim)
    | filter(...)

    // Getting only valid JSON objects
    | transform([](const std::string& message) {
        return mtry([&] {
            return json::parse(message);
        });
    })

```

```

| filter(&expected_json::is_valid)
| transform(&expected_json::get)

// Getting only the valid bookmarks
| transform(bookmark_from_json)
| filter(&expected_bookmark::is_valid)
| transform(&expected_bookmark::get)

| sink_to_cerr;

```

The above pattern is clear — perform a transformation that can fail, filter out all the invalid ones, and extract the value from `expected` object for further processing.

The problem is that it is overly verbose. But it is not the main problem. A bigger problem is that we forget the error as soon as we encounter it. If we wanted to forget errors, we would not use `expected`, we would use `optional`.

Now, this is the point where this example starts to be more interesting. We got a stream of values, where each value is an instance of `expected` monad. So far in this example, we have only treated streams as monads, and we have treated all the messages as normal values. Will this code become more readable if we treated `expected` as a monad, which it is?

Instead of doing the whole `transform-filter-transform` shebang, let's transform the instances of `expected` in a monadic manner. If we look at the signature of the `bookmark_from_json` function, we will see that it takes a value, and gives an instance of the `expected` monad. We have seen that we can compose functions like these with monadic composition — `mbind`.

Listing 12.10. Treating expected as a monad

```

auto pipeline =
    service(event_loop)
    | transform(trim)
    | filter(...)

    | transform([](const std::string& message) {           1
        return mtry([&] {                                     1
            return json::parse(message);                      1
        });
    })                                                       1

    | transform([](const auto& exp_json) {                  2
        return mbind(exp_json, bookmark_from_json);          2
    })                                                       2

    ...
    | sink_to_cerr;

```

- ➊ Until this point, we had a stream of normal values. We get here a stream of expected instances — a monad inside of a monad
- ➋ If we can use `mbind` to transform instances of `expected`, we can lift it with `transform` to work on streams of `expected` objects
- ➌ We can concatenate as many fallible transformations as we want.

This is a nice example of how lifting and monadic bind can work together. We started with a function that works on normal values of `json` type, then bound it so that it can be used with `expected_json`, and then lifted it to work on streams of `expected_json` objects.

12.6 Replying to the client

The service we implemented so far just receives requests from the client, but it never replies. This might be useful if we just wanted to store the bookmarks the clients send us — instead of `sink_to_cerr`, we could write the bookmarks to a database.

It is more often the case that we need to send some kind of reply to the client, at least to confirm that we have received the message.

At first glance, this seems like a problem given the design of our service. We have collected all messages into a single stream — our main program does not even know that clients exist.

Now we have two choices. One is to go back to the drawing board. The other one is to listen to that voice in the back of our head that whispers "Monads. You know this can be done with monads.". Instead of deleting everything we have implemented so far, let's listen to that voice.

If we want to be able to respond to a client instead of writing the bookmarks to `std::cerr` or to a database, we need to know which client sent us which message. The only component in the system which can tell us that is the service object. This means that we somehow need to pass the information about the client through the whole pipeline — from `service(event_loop)` up to the sink object without it being modified in any of the steps.

Instead of the service object sending messages containing only strings, it needs to send messages that contain strings and a pointer to the socket we can use to further communicate with the client. Since the socket needs to be passed on through all the transformations while the message type changes, we will create a template class to keep the socket pointer along any type of message.

Listing 12.11. Structure to hold a socket along with the message

```
template <typename MessageType>
struct with_client {
    MessageType value;
```

```

tcp::socket* socket;

void reply(const std::string& message) const
{
    // Copy and retain the message until the async_write
    // finishes its asynchronous operation
    auto sptr = std::make_shared<std::string>(message);
    boost::asio::async_write(
        *socket,
        boost::asio::buffer(*sptr, sptr->length()),
        [sptr](auto, auto) {});
}
};

```

In order to simplify the main program not to have any dependency on Boost.ASIO, we have also created a `reply` member function (see the full implementation in the accompanying example `bookmark-service-with-reply`) which we can use to send messages to the client.

The `with_client` is a generic type that holds some extra information. We have learnt that we should think **functor** and **monad** every time see something like this. It is easy to create all required functions in order to show that `with_client` is a monad.

Join function for the with_client

The only function that deserves a bit of consideration is `join` — if we have a `with_client` nested inside of another `with_client`, we will have one value and two pointers to a socket, whereas we just need the value with a single socket after the join.

We can choose always to keep the socket from the innermost instance of the `with_client` or to always keep the socket from the outermost instance. In our use-case, whatever we do, we always want to reply to the client that initiated the connection which means that we need to keep outermost socket.

Alternatively, we could have changed the `with_client` class not to keep only one socket, but to keep a collection of sockets. In that case, joining a nested instance of `with_client` would just need to merge these two collections.

If we change the service to emit messages of type `with_client<std::string>` instead of plain strings, what else should we change in order for our program to compile?

Obviously, we need to change the sink. It needs to send the messages to the client instead of writing them to `std::cerr`. The sink will receive messages of type `with_client<expected_bookmark>`. It needs to check whether the `expected` object contains an error or not, and act accordingly:

```

auto pipeline =
    service(event_loop)
    ...
    | sink([](const auto& message) {
        const auto exp_bookmark = message.value;

        if (!exp_bookmark) {
            message.reply("ERROR: Request not understood\n");
            return;
        }

        if (exp_bookmark->text.find("C++") != std::string::npos) {
            message.reply("OK: " + to_string(exp_bookmark.get()) +
                          "\n");
        } else {
            message.reply("ERROR: Not a C++-related link\n");
        }
    });
}

```

If there was any error while we parsed the bookmarks, we will notify the client. Also, since we want to accept only C++-related bookmarks, we will report error if the text of the bookmark does not contain "C++".

We have changed the service, and we have changed the sink to be able to reply to the client. What else needs changing?

We could go and change all transformations one by one until we make them all understand the newly introduced `with_client` type. But we could be smarter than that. Just like we handled fallible transformations with `mbind` instead of passing each message through a `transform-filter-transform` chain of modifiers, we should try to do similar here.

This is just another level of monads. We have a stream (which is a monad) of `with_client` values (which is also a monad) which each of them containing a `expected<T, E>` value (a third nested monad). We just need to be able to lift everything one level further.

We want to redefine the `transform` and `filter` functions implemented for our reactive streams which reside in the `reactive::operators` namespace (see example `bookmark-service-with-reply`) to work on a reactive stream of `with_client` values:

```

auto transform = [](auto f) {
    return reactive::operators::transform(lift_with_client(f));
};
auto filter = [](auto f) {
    return reactive::operators::filter(apply_with_client(f));
};

```

The `lift_with_client` is a simple function that lifts any function from `T1` to `T2` to a

function from `with_client<T1>` to `with_client<T2>`, while `apply_with_client` does similar, only that it returns an unwrapped result value instead of putting it into the `with_client` object.

This is everything we needed to do, the rest of the code will continue functioning without any changes whatsoever.

Listing 12.12. The final version of the server (example:bookmark-service-with-reply/main.cpp)

```
auto transform = [](auto f) {
    return reactive::operators::transform(lift_with_client(f));
};
auto filter = [](auto f) {
    return reactive::operators::filter(apply_with_client(f));
};

boost::asio::io_service event_loop;

auto pipeline =
    service(event_loop)
    | transform(trim)

    // Ignoring comments and empty messages
    | filter([](const std::string& message) {
        return message.length() > 0 && message[0] != '#';
    })

    // Trying to parse the input
    | transform([](const std::string& message) {
        return mtry([&] { return json::parse(message); });
    })

    // Converting the result into the bookmark
    | transform([](const auto& exp) {
        return mbind(exp, bookmark_from_json);
    })

    | sink([](const auto& message) {
        const auto exp_bookmark = message.value;

        if (!exp_bookmark) {
            message.reply("ERROR: Request not understood\n");
            return;
        }

        if (exp_bookmark->text.find("C++") != std::string::npos) {
            message.reply("OK: " + to_string(exp_bookmark.get()) +
                         "\n");
        } else {
            message.reply("ERROR: Not a C++-related link\n");
        }
    })
}
```

```

    }
});

// Starting the Boost.ASIO service
std::cerr << "Service is running...\n";
event_loop.run();

```

This shows the power of using generic abstractions like functors and monads. We have managed to dig through the whole processing line just by changing a few things and leaving the main program logic intact.

12.7 Creating actors with a mutable state

While we should always try not to have any mutable state, there are situations where it is useful. It might have gone unnoticed until this point, but we have already created one transformation which has a mutable state — the `join` transformation. It keeps a list of all sources whose messages it forwards.

In this case, it is an implementation detail — we need to keep the sources alive. But there are also situations where explicitly stateful actors are necessary.

In order to keep the service responsive, we can not give the same priority to all messages. Say we have a client which is trying to perform a DoS (Denial of Service) attack by flooding us with messages so that we become unable to reply to other clients.

There are various approaches to dealing with problems like these. One of the simpler ones is message throttling. When we accept a message from the client to process, we reject all subsequent messages until some predefined time interval has passed. For example, we might define a throttle of one second, which would mean that after we accept a message from a client, we will ignore that client for one second.

For this, we can create an actor that accepts messages and remembers the client that sent the message, along with the absolute time when we will start accepting messages from that client again. This will require the actor to have mutable state — it needs to remember and update timeouts for each client.

In the ordinary concurrent software systems, having mutable state would require synchronization. This is not the case in actor-based systems. An actor is a single-threaded component completely isolated from all other actors. This means that the state we want to mutate can not be mutated from different concurrent processes. Since we do not share any resources, we do not need to do any synchronization.

As previously mentioned, the actors in our bookmarks service example are oversimplified. We can process many clients concurrently — we can handle quite a number of clients at the same time, but we are still doing all the processing in a single thread. And we use asynchronous messaging only in the places where we

communicate with the client (in the parts of the code that uses Boost.ASIO).

In the general actor model, each actor lives in a separate process or a thread (or in something even more lightweight that behaves like a thread). Since all actors have their own little world where the time passes on independent to the time of other actors, there can be no synchronous messages.

This means that all actors need to have their own message queues to which we can add as many messages as we want, and the actor (as a single-threaded component) will process the messages in the queue one by one.

In our implementation, messages were synchronous. Calling `m_emit` in one of the actors would immediately call the `process_message` function in another. If we wanted to create a multi-threaded system, we would just need to make these calls indirect. We would need to have a message processing loop in each thread, and that loop would just need to deliver the messages to the right actor.

The change in the infrastructure would not be trivial, but the concept of an actor being an isolated component which receives and sends messages would not change. Only the message delivery mechanism would.

While the underlying implementation would change, the design of the software would not need to. While designing the message pipeline, we have not relied on our system being single-threaded. We designed it as a set of isolated components which process each other's messages — we did not require any of those messages to be delivered immediately.

This means that the message pipeline we designed can stay completely intact — both conceptually and code-wise — even if we completely revamp the underlying system.

12.8 Writing distributed systems with actors

There is another benefit of designing concurrent software systems as a set of actors which send messages to one another.

We said that all actors are isolated — they share nothing, not even the timelines. The only thing that is guaranteed is that the messages an actor has in its queue are processed in the order they were sent.

This means that actors do not care whether they live in the same thread, a different thread in the same process, a different process on the same computer, or in different computers, as long as they can send messages to one another.

One implication that follows from this is that we can easily scale our bookmark service horizontally without the need to change its main logic. Each of the actors we created can live on a separate computer and send messages over the network.

Just like switching from a single-threaded to a multi-threaded system did not incur

any changes to the main program logic, switching an ordinary system based on actors and reactive streams to a distributed system will also leave it intact.

The only necessary change is in the message delivery system. With multi-threaded execution, we needed to create message processing loops in each thread, and know how to deliver the right messages to the right loop, and therefore to the right actor. In the distributed systems, the story is the same — we have just another level of indirection. The messages need to be able not only to travel between threads, but also to be serialized for sending over the network.

12.9 Summary

- Most of C++ programmers are writing procedural code. I recommend reading "Object Thinking" by David West to start writing better object-oriented code. It is beneficial even when doing Functional Programming.
- Humans tend to achieve grand things when talking to one another. We do not have a shared mind, but the ability to communicate helps us to achieve very complex goals. This is the exact reasoning that lead to the invention of the actor mode.
- Monads can cooperate quite well together. We should not be afraid to stack them on each other.
- We can implement similar transformations for reactive streams that we have for input ranges. We can not implement things like sorting on them because for sorting we need random access to all elements, and we do not even know how many elements a reactive stream will have — they are all potentially infinite.
- Just like futures, common implementations of reactive streams are not limited to sending only the values, they can also send special messages like 'stream ended'. This can be useful for more efficient memory handling — we can destroy a stream when we know it will send us no more messages

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch12

13

Testing and debugging

This chapter covers:

- Avoiding runtime errors by moving them to compile-time
- What are benefits of pure functions in unit testing
- Automatically generating tests cases for pure functions
- Testing our code by comparing to existing solutions
- Testing monad-based concurrent systems

Computers are becoming omnipresent in all parts of our lives. We are getting smart watches, TVs, toasters etc. Consequences of bugs in software today range from minor annoyances to serious problems like identity theft and even life endangerment.

Because of this, it is more important than ever that the software we write is correct — that it does exactly what it should and that it does not contain bugs.

While this sounds like a no-brainer — because who in their right mind would want to write bug-ridden software — it is a widely accepted stance that all non-trivial programs contain bugs. We are even so accustomed to this fact that we tend to subconsciously develop work-arounds to avoid the bugs we discover in programs we are using.

While this is the sad truth, it still is not an excuse for us not to try to write correct programs. The issue is that this is not easy to do.

Most features of higher-level programming languages are introduced exactly with

this in mind. This is especially true for C++ where most of the recent evolution is focussed on making safe programs easier to write. Or to be more precise, to make it easier for programmers to avoid the common programming mistakes.

We have seen safety improvements in dynamic memory management with smart pointers, automatic type deduction with `auto` to shorten the code and avoid accidental implicit casts, and monads like `std::future` for making it easier to develop correct concurrent programs without bothering with low level concurrency primitives like mutexes. We have also seen the push towards stronger type-based programming with algebraic data types with `std::optional` and `std::variant`, units and user-defined literals (`std::chrono::duration` as an example), etc.

13.1 Is the program that compiles correct?

All these features are here to help us avoid some common programming mistakes and move the error detection from runtime to the compilation time.

One of the most famous examples of how a simple error can create a huge loss was the Mars Climate Orbiter bug where most of the code assumed that the distances are measured in metric units, where a part of the code used the imperial (English) units.

The MCO MIB has determined that the root cause for the loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file, “Small Forces,” used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used in the software application code titled SM_FORCES (small forces).

-- NASA -- Mars Climate Orbiter Mishap Investigation Report

This error could have been easily avoided if the code used stronger typing instead of raw values. We can easily create a type to handle distances that forces a certain measurement unit.

```
template <typename Representation,
          typename Ratio = std::ratio<1>>
class distance {
    // ...
};
```

This type would allow us to easily create different types for different measurement units, and to represent the number of those units with an arbitrary numeric type — be it integers, floating-point numbers, or some special type we created. If we assume that meters are the default, we can create others easily (rounded up miles-to-meters for simplicity):

```

template <typename Representation>
using meters = distance<Representation>;

template <typename Representation>
using kilometers = distance<Representation, std::kilo>;

template <typename Representation>
using centimeters = distance<Representation, std::centi>;

template <typename Representation>
using miles = distance<Representation, std::ratio<1609>>;

```

We can also make it easier for these to be used by creating user defined literals for them.

```

constexpr kilometers<long double> operator ""_km(long double distance)
{
    return kilometers<long double>(distance);
}

// ... and similar for other units

```

Now we can write our program using any unit we want. But if we try to mix and match different units, we will get a compilation error because the types do not match:

```
auto distance = 42.0_km + 1.5_mi; // error!
```

We could provide conversion functions for this to be more usable, but we already achieved the main goal. We have a small zero-cost abstraction that moves the error from runtime to the compilation time. This makes a huge difference during the software development cycle — the difference between losing a space probe, to having the bug detected long before the probe is even scheduled for launch.

By using the higher-level abstractions we have covered during the course of this book, we can move many common software bugs to the compilation time.

Because of this, some people tend to say that once you successfully compile a functional program, it is bound to work correctly.

Obviously, all non-trivial programs have bugs, and this applies to FP-style programs as well. But the fact is that the shorter the code is (and, as we've seen, FP and the abstractions it introduces allow us to write significantly shorter code compared to the usual imperative approach) fewer places we have to make mistakes, and by making as many mistakes detectable at compile-time as possible, we significantly reduce the number of runtime errors.

13.2 Unit testing and pure functions

While we should always try to write the code in a way that makes potential

programming errors detectable during compilation, it is not always possible. Our programs still need to process real data during runtime, and we could still make logic errors, or produce incorrect results.

For this, we need to have automatic tests for our software. Automatic tests are also useful for regression testing when we change already existing code.

The lowest level of testing are the unit tests. The goal of unit testing is to isolate small parts of the program and test them individually to assure their correctness. These test the correctness of the units themselves, not how they integrate with one another.

The good thing is that unit testing in functional programming is quite similar to unit testing of imperative programs, and we can use all the same libraries we are accustomed to while writing normal unit tests. The only difference is that testing pure functions is a bit easier.

Traditionally, a unit test for a stateful object consists of setting up the object state, performing an action on that object, and checking the result.

Imagine we have a class that handles a textual file. It might have several member functions, including one that counts the number of lines in said file in the same way we counted the lines in chapter 1 — by counting the number of newline characters in the file.

```
class textual_file {
public:
    int line_count() const;

    // ...
};
```

If we wanted to create a unit test for this function, we would need to create several files, create instances of `textual_file` for all of them, and check the result of the `line_count` function.

This is a common approach if a class that we are testing has state — we need to initialize the state, and only then we can perform the test. We often need to perform the same test with various different states that the class can be in.

This often means that in order to write a good test, we need to know which parts of the class state can influence the test we are writing. For example, the state for the `textual_file` class might include a flag to tell us whether the file is writeable or not. We need to know its internals to be able to tell that this flag has no influence on the result of `line_count`, or we would need to create tests which cover both writable and read-only files.

This becomes much simpler when testing pure functions. The function result can depend only on the arguments we pass into the function, and if we do not add

superfluous arguments to functions just for the fun of it, we can assume that all arguments are used in order to calculate the result.

We do not need to set up any external state before running tests, and we can write tests without considering how the function that we are testing is implemented. This decoupling of a function and the external state also tends to make the function more general, which increases reusability and allows testing of the same function in various different contexts.

Consider the following pure function:

```
template <typename Iter, typename End>
int count_lines(const Iter& begin, const End& end)
{
    using std::count;
    return count(begin, end, '\n');
}
```

As a pure function, it does not need any external state in order to calculate the result, it does not use anything besides its arguments, and it does not modify the arguments.

When testing, we can call this function without any previous preparations, and we can call it on several different types — from lists and vectors, to ranges and input streams:

```
// Testing with a string
std::string s = "Hello\nworld\n";
assert(count_lines(begin(s), end(s)) == 2);

// Testing with a range
auto r = s | view::transform([](char c) { return toupper(c); });
assert(count_lines(begin(r), end(r)) == 2);

// Testing with an input stream (instead of being limited only to files)
std::istrstream ss("Hello\nworld\n");
assert(count_lines(std::istreambuf_iterator<char>(ss),
                  std::istreambuf_iterator<char>()) == 2);

// Testing with a singly-linked list
std::forward_list<char> l;
assert(count_lines(begin(l), end(l)) == 0);
```

If we were so inclined, we could implement overloads that would be more comfortable to use, instead of having to call `count_lines` always with pairs of iterators. Those would just be one-line wrappers and not something that would require thorough testing.

Our task when writing unit tests is to isolate small parts of the program and test them individually. Every pure function is already an isolated part of the program.

This, along with the fact that pure functions are easy to test, makes each pure function a perfect candidate for a **unit**.

13.3 Automatically generating tests

While unit tests are useful (and necessary) the main problem with them is that we need to write them by hand.

This makes them error-prone because we might make coding errors in the tests themselves, and we are in the risk of writing incorrect or incomplete tests. Just as it is harder to find spelling errors in your own writing than in somebody else's, it is more difficult to write tests for your own code — you are likely to skip the same corner-cases you forgot to cover in the implementation.

It would be much more convenient if the tests could be automatically generated for us based on what we are testing.

13.3.1 Generating test cases

When implementing the `count_lines` function, we have its specification — given a collection of characters, return the number of newlines in that collection.

What is the inverse problem of line counting? Given a number, generate all collections whose line counts are equal to the given number. This would yield a function like this (covering only strings as the collection type):

```
std::vector<std::string> generate_test_cases(int line_count);
```

If these problems are inverse of one another, this means that for any collection generated by `generate_test_cases(line_count)`, the `count_lines` function needs to return the same value `line_count` we passed to `generate_test_cases`. And this needs to be true for any value of `line_count` — from zero to infinity. We could write this rule as follows:

```
for (int line_count : view::ints(0)) {
    for (const auto& test : generate_test_cases(line_count)) {
        assert(count_lines(test) == line_count);
    }
}
```

This would be a perfect test, but it has one **small** problem. The number of cases we are testing is infinite as we are traversing the range of all integers starting with zero. And for each of them we can have an infinite number of strings that have the given number of newlines.

Since we can not check all of these, we will need to generate only a subset of problems, and check whether the rule holds only for that subset. Generating a single example of a string that has a given number of newlines is trivial. We can generate it by creating a sufficient number of random strings and concatenate them

by putting a newline between each two. Each string will have a random length and random characters inside — we must just make sure they do not have newlines in them.

```
std::string generate_test_case(int line_count)
{
    std::string result;

    for (int i = 0; i < line_count; ++i) {
        result += generate_random_string() + '\n';
    }

    result += generate_random_string();
    return result;
}
```

This will generate a single test case — a single string that contains exactly `line_count` number of newlines. Now we can simply define the function that returns an infinite range of these examples:

```
auto generate_test_cases(int line_count)
{
    return view::generate(std::bind(generate_test_case, line_count));
}
```

Now we just need to limit the number of tests we are performing. Instead of covering all integers, and processing an infinite number of collections for each, we can add some predefined limits:

Listing 13.1. Testing the count_lines function on a set of randomly generated tests

```
for (int line_count :
    view::ints(0, MAX_NEWLINE_COUNT)) { ①
    for (const auto& test :
        generate_test_cases(line_count) ②
        | view::take(TEST_CASES_PER_LINE_COUNT)) { ②
        assert(line_count ==
            count_lines(begin(test), end(test)));
    }
}
```

- ① Instead of covering all integers, check only up to some predefined value
- ② Specifying how many test cases we want to have for each number of lines

While this covers only a subset of all possible inputs, each time the tests are run, a new set of random examples will be generated. This means that with each new run, the space of inputs that we have checked the correctness for will be expanded.

The downside of the randomness is that the tests might fail in some invocations, and in some not. This could sometimes lead to a wrong implication that when the

tests fail that the last change we made to the program is at fault. In order to remedy this, it is always a good idea to write out the seed we used for the random number generator to the test output so that we can later easily reproduce the error and find the software revision in which it was introduced.

13.3.2 Property-based testing

Sometimes we have problems for which checks are already known, or much simpler than the problem itself. Imagine we want to test a function that reverses the order of the items in a vector:

```
template <typename T>
std::vector<T> reverse(const std::vector<T>& xs);
```

We could create a few test cases and check whether `reverse` works correctly for them. But, again, this covers only a small number of cases. In this case, we could try to find out some rules that apply to the `reverse` function.

First, let's see what is the inverse problem of reversing a given collection `xs` — we need to find all collections which, when reversed, give that original collection `xs`. There exists only one such, and it is the `reverse(xs)`. So, reversing a collection is the inverse problem of itself:

```
xs == reverse(reverse(xs));
```

And this needs to hold for any collection `xs` that we can think of.

We can also add a few more properties of the `reverse` function:

- The number of elements in the reversed collection needs to be the same as the number of elements in the original collection
- The first element of a collection needs to be the same as the last element in the reversed collection
- The last element of a collection needs to be the same as the first element in the reversed collection

All these need to hold for any collection. This means that we can generate as many random collections as we want, and check that all these rules hold for them.

Listing 13.2. Generating test cases and checking that properties hold

```
for (const auto& xs : generate_random_collections()) {
    const auto rev_xs = reverse(xs);

    assert(xs == reverse(rev_xs));           ①

    assert(xs.length() == rev_xs.length());   ②

    assert(xs.front() == rev_xs.back());     ③
    assert(xs.back() == rev_xs.front());     ③
```

{}

- ① If we reverse a collection two times, we get the original collection
- ② The original and reversed collections need to have the same number of elements
- ③ The first element in the original collection is the same as the last element of the reversed collection, and vice-versa

Just like in the previous case where we checked the `count_lines` function for correctness, we will check a different part of the function input space with each new run of the tests. The difference here is that we do not need to create a smart generator function for our test examples. Any randomly generated example has to satisfy all the properties of the `reverse` function.

In a similar manner, we could do the same for other problems as well. Problems that are not inverse of themselves, but that still have some properties that need to hold.

Imagine we now need to test whether a sorting function works correctly. There are more than a few different ways to implement sorting, some are more efficient when sorting in-memory data, some are better when sorting data on storage devices. But all of them have the same rules they need to follow:

- The original collection needs to have the same number of elements as the sorted one
- The minimum element of the original collection needs to be the same as the first element in the sorted collection
- The maximum element of the original collection needs to be the same as the last element in the sorted collection
- Each element in a sorted collection must be greater or equal to its predecessor
- Sorting a reversed collection should give the same result as sorting the collection without reversing it

We have generated a set of properties that we can easily check (this list is of course not extensive, but is sufficient for demonstration purposes):

Listing 13.3. Generating test cases and checking whether the sorting properties hold

```
for (const auto& xs : generate_random_collections()) {
    const auto sorted_xs = sort(xs);

    assert(xs.length() == sorted_xs.length()); ①

    assert(min_element(begin(xs), end(xs)) == ②
          sorted_xs.front());
    assert(max_element(begin(xs), end(xs)) == ②
          sorted_xs.back());

    assert(is_sorted(begin(sorted_xs)), ③
```

```

        end(sorted_xs));
③

assert(sorted_xs == sort(reverse(xs)));
④
}

```

- ① Checking that the sorted collection has the same number of elements as the original one
- ② Checking the smallest and largest elements in the original collection are the first and last elements in the sorted collection
- ③ Checking that each element in the sorted collection is greater than or equal to its predecessor
- ④ Checking that sorting a reversed list yields the same result as sorting the original list (assuming total ordering of elements)

When we define a set of properties for a function, and implement checks for them, we can just generate random input and feed it to the checks. If any of the properties fail on any of the cases, we know that we have a buggy implementation.

13.3.3 Comparative testing

So far we have seen how to automatically generate tests for functions that we know how to solve the inverse problem of, and how to test function properties that need to hold regardless what data we provide to the function.

There is a third option where randomly generated tests can improve our unit tests.

Imagine we want to test the implementation of the bitmapped vector trie (BVT) data structure we saw in the chapter 8. We designed it to be an immutable (persistent) data structure. It looks and behaves like the standard vector with one exception — it is optimized for copies and it does not allow in-place mutation.

The easiest way to test a structure like this is to test it against the structure it aims to mimic — against a normal vector. We need to test all operations we defined on our structure, and compare the result with the same or equivalent operation performed on the standard vector.

For this, we will just need to be able to convert between the standard vector and a BVT vector, and to be able to check whether a given BVT and standard vectors contain the same data.

When we have that, we can create a set of rules we want to check against. Again, these rules need to hold for any random collection of data we have. The first thing we need to check is that the BVT constructed from a standard vector contains the same data as that vector, and vice-versa. After that, we need to test all the operations we have — to perform them on both the BVT and the standard vector and check whether the resulting collections also hold the same data.

Listing 13.4. Generating test cases and comparing whether BVT and vector behave the same

```
for (const auto& xs : generate_random_vectors()) {
    const BVT bvt_xs(xs);

    assert(xs == bvt_xs); ❶

    {
        auto xs_copy = xs;
        xs_copy.push_back(42); ❷
        assert(xs_copy == bvt_xs.push_back(42)); ❸
    }

    if (xs.length() > 0) {
        auto xs_copy = xs;
        xs_copy.pop_back();
        assert(xs_copy == bvt_xs.pop_back());
    }

    // ...
}
```

- ❶ If both collections support iterators, this is trivial to implement with `std::equal` algorithm
- ❷ Since BVT is immutable, we need to simulate that behaviour with the standard vector as well — we first create a copy, and then modify it

These approaches to automatic test generation are not exclusive. We can use them together to test a single function.

For example, in order to test a custom implementation of the sort algorithm, we can use all three approaches.

- For each sorted vector, we can create an unsorted one simply by shuffling it. Sorting the shuffled version needs to return the original vector.
- We can test against a few different properties that all sorting implementations need to have, which we have already seen.
- And we can generate random data and sort it with our sorting algorithm and `std::sort` to make sure they give the same output.

When we get a list of checks, we can feed them as many randomly generated examples as we want. Again, if any of the checks fail, our implementation is not correct.

13.4 Testing monad-based concurrent systems

We have seen the implementation of a simple web service in the chapter 12. That implementation was based mostly on reactive streams, and we used a couple of other monadic structures — `expected<T, E>` to handle errors,

and `with_socket<T>` to transfer the socket pointer through the program logic so that we could send a reply to the clients.

This monadic data-flow software design has a few benefits we have already seen. It is composable — we split the program logic into a set of completely isolated range-like transformations. Transformations that can easily be reused in other parts of the same program, or in other programs.

Another big benefit is that we were able to modify the original server implementation to be able to reply to the client without changing a single line in the main program logic — in the data flow pipeline. We just needed to lift the transformations one level up — to teach them how to handle the `with_socket<T>` type, and everything else just worked.

In this section, we are going to leverage the fact that all monadic structures are alike — they all have `mbind`, `transform` and `join` defined on them — and that if we base our logic on these functions (or functions built on top of these), we can freely switch between different monads without changing the main program logic so that we can implement tests for our program.

One of the main problems when testing concurrent software systems, or software which has parts that are executed asynchronously from the main program, is that it is not easy to write tests to cover all the different possible interactions between concurrent processes in the system. If two concurrent processes need to talk to each other, or share the same data, a lot can change if in some situations one of them takes more time to finish than expected.

Simulating this during testing is hard and is almost impossible to detect all the problems that get exposed in production. Furthermore, replicating a problem detected in production can be a real pain because it is difficult to replicate the same timings of all processes.

In the design of our small web service, we never made any assumptions (explicit nor implicit) about how much time any of the transformations take. We did not even assume whether the transformations are synchronous or not.

The only thing that we assumed is that we have a stream of messages coming from a client. While this stream was asynchronous, the defined data flow has no need for it to be — it will have to work even for synchronous streams of data — for ranges.

As a short reminder, this is what the data flow pipeline looked like:

```
auto pipeline =
    source
    | transform(trim)

    // Ignoring comments and empty messages
    | filter([](const std::string& message) {
        return message.length() > 0 && message[0] != '#';
```

```

    })

// Trying to parse the input
| transform([](const std::string& message) {
    return mtry([&] { return json::parse(message); });
})

// Converting the result into the bookmark
| transform([](const auto& exp) {
    return mbind(exp, bookmark_from_json);
})

| sink([](const auto& message) {
    const auto exp_bookmark = message.value;

    if (!exp_bookmark) {
        message.reply("ERROR: Request not understood\n");
        return;
    }

    if (exp_bookmark->text.find("C++") != std::string::npos) {
        message.reply("OK: " + to_string(exp_bookmark.get()) +
                      "\n");
    } else {
        message.reply("ERROR: Not a C++-related link\n");
    }
});

```

We have a stream of strings coming from the source, and we parse them into bookmarks. If we just got this code without being given any sort of context, the first thing that we would think is not that the source is a service based on Boost.ASIO, but that it is some kind of a collection, and that we are using the range-v3 library to process that collection.

And this is the main benefit of this design when testing is concerned — we can switch asynchronicity on and off as we please. When the system runs, we will use reactive streams, when we need to test the main logic of the system, we will be able to use normal data collections.

Let's see what exactly we need to change in order to make a test program from our web service.

Since we do not need the actual service component when testing the pipeline, we can just strip out all the code that uses Boost.ASIO. The only thing that needs to remain is the wrapper type we use to send the messages back to the client. Since we no longer have clients, instead of a pointer to the socket, we will store the expected reply message in that type. Then, when the pipeline calls the `reply` member function, we will check whether we got the message we expected.

```
template <typename MessageType>
struct with_expected_reply {
    MessageType value;
    std::string expected_reply;

    void reply(const std::string& message) const
    {
        REQUIRE(message == expected_reply);
    }
};
```

Just like `with_socket`, this structure carries the message along with some contextual information. We can use this class as a drop-in replacement for `with_socket` — as far as the data-flow pipeline is concerned, nothing has changed.

The next step is to redefine the pipeline transformations to use the transformations from the range library instead of the transformations from our simple reactive streams library. Again, we do not need to change the pipeline, we will just change the definitions of `transform` and `filter` we had in the original program. They need to lift the range transformations to work with the `with_expected_reply<T>`:

```
auto transform = [](auto f) {
    return view::transform(lift_with_expected_reply(f));
};
auto filter = [](auto f) {
    return view::filter(apply_with_expected_reply(f));
};
```

We also need to define the `sink` transformation since ranges do not have that one. The `sink` transformation should call the given function on each value from the source range. We can use `view::transform` for this, but with a slight change. The function we passed to `sink` returns `void`, and we can not pass it to `view::transform` directly because it would result in a range of voids. We will need to wrap the transformation function into a function that will return an actual value.

```
auto sink = [](auto f) {
    return view::transform([f](auto&& ws) {
        f(ws);
        return ws.expected_reply;
    });
};
```

This is all, we can now create a vector of `with_expected_reply<std::string>` instances and pass it through the pipeline. As each item in the collection gets processed, it will test whether the reply was correct or not. For a full implementation of this example, check out the accompanying example `bookmark-service-testing`.

It is important to note that this is just the test for the main program logic. It does not relieve us from writing tests for the service component, and for the individual transformation components like `filter` and `transform`. Usually, tests for small components like these are easy to write, and most bugs do not arise from the component implementation, but from different components' interaction. And this interaction is exactly what we simplified the testing for.

13.5 Summary

- Every pure function is a good candidate to be unit tested. We know exactly what it uses in order to be able to calculate the result, and we know it does not change any external state — its only effect is that it returns a result
- One of the most famous libraries that do property-checking against a randomly generated data set is Haskell's QuickCheck. It inspired similar projects for many different programming languages, even for C++
- By changing the random function, we can change which types of tests are more often performed. For example, when generating random strings, we can favour shorter strings by using a random function with normal distribution and with mean zero.
- Fuzzing is another testing method that uses random data. There, the idea is to test whether software works correctly given invalid (random) input. It is highly useful for programs that accept unstructured input
- Remembering the initial random seed allows us to replicate the tests that have failed.
- Correctly designed monadic systems should work without problems if the continuation monad or reactive streams are replaced with normal values and normal data collections.
- This essentially allows us to **switch on and off** concurrency and asynchronous execution on the fly. We can use this switch during testing.

Additional resources: cukic.co/to/fp-in-cpp/additional-resources/ch13