Содержание

Предисловие

Программирование – одна из тех немногих дисциплин, что позволяют творить нечто буквально из ничего. Программист может творить целые миры, которые ведут себя в точности как задумал автор. Для этого нужен лишь компьютер.

Когда я учился в школе, курс программирования был в основном сфокусирован на императивном программировании – сначала это было процедурное программирование на языке C, затем объектно-ориентированное на языках C++ и Java. С поступлением в университет почти ничего не изменилось – основной парадигмой по-прежнему оставалось объектно-ориентированное программирование (ООП).

Поэтому тогда я едва не попался в мыслительную ловушку, убедив себя в том, что все языки программирования, в сущности, одинаковы, различия между ними – чисто синтаксические и программисту довольно изучить такие основополагающие понятия, как ветвление и цикл, в одном языке, чтобы программировать (с небольшими поправками) на всех остальных языках.

С языками функционального программирования я впервые познакомился в университете, когда в рамках одной из дисциплин понадобилось изучить язык LISP. Моей первой реакцией было смоделировать средствами языка LISP условный оператор `if-then-else` и оператор цикла `for`, чтобы сделать язык пригодным для работы. Вместо того чтобы привести свое восприятие в соответствие с языком, я решил доработать язык напильником, чтобы он позволял мне и дальше писать программы таким же образом, каким я привык писать на языке C. Нетрудно догадаться, что в те дни я не увидел никакого смысла в функциональном программировании, ведь все, чего я мог добиться, используя LISP, можно было гораздо проще сделать на языке C.

Прошло немало времени, прежде чем я снова стал поглядывать в сторону функционального программирования. Подтолкнула меня к этому неудовлетворенность слишком медленной эволюцией одного языка, который мне необходимо было использовать для нескольких проектов. В язык наконец добавили оператор цикла `for-each`, и это подавалось как громадное достижение. Программисту оставалось лишь загрузить новый компилятор, и жизнь должна была заиграть новыми красками.

Это заставило меня задуматься. Чтобы получить в свое распоряжение новую языковую конструкцию наподобие цикла `for-each`, мне нужно было дождаться новой версии языка и новой версии компилятора. Но на языке LISP я мог самостоятельно реализовать цикл `for` в виде обыкновенной функции. Никакого обновления компилятора при этом не требовалось.

Именно это склонило меня к функциональному программированию: возможность расширять язык без необходимости менять компилятор. Я по-прежнему оставался в плену объектно-ориентированного мировоззрения, но уже научился использовать конструкции, заимствованные из функционального стиля, чтобы упростить работу по созданию объектно-ориентированного кода.

Тогда я стал посвящать много времени исследованию функциональных языков программирования, таких как Haskell, Scala и Erlang. Меня поразило, что многие проблемы, заставляющие страдать объектно-ориентированных программистов, удается легко решить, посмотрев на них под другим углом – функциональным.

Поскольку большая часть моей работы связана с языком C++, я решил найти способ, как использовать в этом языке приемы функционального программирования. Оказалось, что я в этом не одинок: в мире полно других людей с похожими идеями. Некоторых из них мне посчастливилось встретить на различных конференциях по языку C++. Всякий раз это была превосходная возможность обменяться идеями, научиться чему-то новому и поделиться своим опытом применения идиом функционального программирования на языке C++.

На большей части таких встреч мы с коллегами сходились на том, что было бы замечательно, если бы кто-то написал книгу о функциональном программировании на C++. Вот только каждый из нас хотел, чтобы эту книгу написал кто-то другой, поскольку каждый искал источник готовых идей, пригодных для собственных проектов.

Когда издательство Manning предложило мне стать автором такой книги, это меня поначалу обескуражило: я считал, что мне следовало бы прочесть об этом книгу, а не написать ее. Однако затем я подумал, что если каждый будет рассуждать подобным образом, никто так и не увидит книгу о функциональном программировании на языке C++. Я решил принять предложение и отправиться в это путешествие. То, что из этого получилось, вы сейчас и читаете.

Благодарности

Я хотел бы поблагодарить всех, чье участие сделало эту книгу возможной: профессора Сашу Малкова за то, что привил мне любовь к языку C++; Ако Самарджича за то, что научил меня, насколько важно писать легкочитаемый код; моего друга Николу Йелича, который убедил меня, что функциональное программирование – это здорово; Золтана Порколаба, поддержавшего мою догадку, что функциональное программирование и язык C++ образуют хорошую смесь; Мирьяну Малькович за помощь в обучении наших студентов тонкостям современного языка C++, включая и элементы функционального программирования.

Почет и уважение Сергею Платонову и Йенсу Веллеру за организацию превосходных конференций по языку C++ для тех из нас, кто все еще живет старыми традициями. Можно смело сказать, что без всех перечисленных людей эта книга бы не состоялась.

Я хотел бы поблагодарить своих родителей, сестру Соню и свою вторую половинку Милицу за то, что всегда поддерживали меня в смелых начинаниях наподобие этого. Кроме того, я благодарен своим давним товарищам по проекту KDE, которые помогли мне вырасти как разработчику за прошедшее десятилетие, – прежде всего Марко Мартину, Аарону Сеиго и Себастиану Кюглеру.

Огромная благодарность команде редакторов, которую для меня организовало издательство Manning: Майклу (или просто Майку) Стивенсу за самое непринужденное из всех моих собеседований; замечательным ответственным редакторам Марине Майклс и Лесли Трайтс, которые научили меня писать книгу (благодаря им я научился гораздо большему, чем мог ожидать); техническому редактору Марку Эльстону за то, что заставлял меня держаться ближе к практике; блестящему Юнвэю Ву, который не только проделал работу по вычитке книги, но и помог улучшить рукопись множеством различных способов. Надеюсь, что доставил всем им не слишком много хлопот.

Также выражаю свою признательность всем, кто предоставил свой отзыв на рукопись: Андреасу Шабусу, Биннуру Курту, Дэвиду Кернсу, Димитрису Пападопулосу, Дрору Хелперу, Фредерику Флейолю, Георгу Эрхардту, Джанлуиджи Спануоло, Глену Сиракавиту, Жану Франсуа Морену, Керти Шетти, Марко Массенцио, Нику Гидео, Никосу Атанасиу, Нитину Годе, Ольве Маудалю, Патрику Регану, Шону Липпи, а в особенности Тимоти Театро, Ади Шавиту, Суманту Тамбе, Джану Лоренцо Меоччи и Николе Гиганте.

Об этой книге

Эта книга предназначена не для того, чтобы научить читателя языку программирования C++. Она повествует о функциональном программировании и о том, как воплотить его в языке C++. Функциональное программирование предлагает непривычный взгляд на разработку программ и иной подход к программированию, по сравнению с императивным и объектно-ориентированным стилем, который обычно используется совместно с языком C++.

Многие, увидев заглавие этой книги, могут счесть его странным, поскольку C++ часто по ошибке принимают за объектно-ориентированный язык. Однако хотя язык C++ и впрямь хорошо поддерживает объектно-ориентированную парадигму, он на самом деле выходит далеко за ее границы. Так, он поддерживает еще и процедурную парадигму, а в поддержке обобщенного программирования большинству других языков с ним не сравниться. Кроме того, язык C++ весьма хорошо поддерживает большинство, если не все, из идиом функционального программирования, в чем читатель сможет вскоре убедиться. С каждой новой версией язык C++ пополнялся новыми средствами, делающими функциональное программирование на нем все более удобным.

Кому стоит читать эту книгу

Эта книга предназначена в первую очередь для профессиональных разработчиков на языке C++. Предполагается, что читатель умеет самостоятельно настраивать среду для сборки программ, устанавливать и использовать сторонние библиотеки. Кроме того, от читателя потребуется хотя бы начальное знакомство со стандартной библиотекой шаблонов, автоматическим выводом типов – параметров шаблона и примитивами параллельного программирования – например, двоичными семафорами.

Впрочем, книга не окажется полностью непонятной для читателя, не искушенного в разработке на языке C++. В конце каждой главы приведен список статей, разъясняющих языковые конструкции, которые могли бы показаться читателю малознакомыми.

Последовательность чтения

Данную книгу лучше всего читать последовательно, так как каждая последующая глава опирается на понятия, разобранные в предыдущих. Если что-либо остается непонятным после первого прочтения, лучше перечитать сложный раздел еще раз, прежде чем двигаться дальше, по-

скольку сложность материала возрастает с каждой главой. Единственное исключение здесь составляет глава 8, которую читатель может пропустить, если не интересуется методами долговременного хранения структур данных.

Книга разделена на две части. Первая часть охватывает идиомы функционального программирования и способы их воплощения в языке C++.

- Глава 1 вкратце знакомит читателя с функциональным программированием и преимуществами, которые оно может привнести в мир C++.
- Глава 2 посвящена функциям высшего порядка – функциям, которые принимают другие функции в качестве аргументов или возвращают функции в качестве значений. Данное понятие иллюстрируется некоторыми из множества полезных функций высшего порядка, включенных в стандартную библиотеку языка программирования C++.
- Глава 3 повествует обо всех разнообразных сущностях, которые в языке C++ рассматриваются как функции, – от обычных функций в смысле языка C до функциональных объектов и лямбда-функций.
- Глава 4 содержит изложение различных способов создания новых функций из имеющихся. В этой главе рассказано о частичном применении функций с использованием операции `std::bind` и лямбда-выражений, а также представлен необычный взгляд на функции, известный как *карринг*.
- В главе 5 говорится о важности неизменяемых данных, то есть объектов данных, которые невозможно модифицировать после создания. Здесь освещены проблемы, возникающие при наличии изменяемого состояния, и методы создания программ без присваивания переменным новых значений.
- Глава 6 посвящена подробному разбору понятия ленивых вычислений. В ней показано, как ленивый порядок вычислений можно использовать для оптимизации, начиная с простых задач наподобие конкатенации строк и до алгоритмов оптимизации, основанных на методе динамического программирования.
- В главе 7 рассказано о диапазонах – современном дополнении к алгоритмам стандартной библиотеки, призванном повысить удобство восприятия кода и его производительность.
- Глава 8 содержит изложение неизменяемых структур данных, т. е. структур данных, которые хранят историю своих предыдущих состояний, пополняя ее при каждой модификации.

Во второй части книги речь пойдет о более сложных понятиях, по большей части относящихся к разработке программ в функциональном стиле.

- В главе 9 речь идет о том, как избавить программу от недопустимых состояний с помощью суммы типов. Показано, как реализовать сумму типов на основе наследования и шаблона `std::variant` и как

для их обработки использовать перегруженные функциональные объекты.

- Глава 10 посвящена функторам и монадам – абстракциям, способным существенно помочь при работе с обобщенными типами и, в частности, позволяющим создавать функции для работы с векторами, значениями типа `optional` и фьючерсами.
- В главе 11 содержится объяснение техник метапрограммирования на шаблонах, способствующих функциональному программированию на языке C++. В частности, разобраны статическая интроспекция, вызываемые объекты и техники метапрограммирования на шаблонах, предназначенные для создания встроенных предметно-ориентированных языков.
- В главе 12 весь предшествующий материал книги собран воедино, чтобы продемонстрировать функциональный подход к разработке параллельных программных систем. В этой главе рассказано, как монаду продолжения можно применить для создания реактивных программ.
- Глава 13 знакомит читателя с функциональным подходом к тестированию и отладке программ.

Автор советует читателю по мере чтения книги реализовывать все изложенные в ней понятия и запускать все встречающиеся примеры кода. Большую часть подходов, описанных в книге, можно использовать и в старых версиях языка C++, однако это потребовало бы написания большого объема дублирующегося кода; поэтому в книге упор сделан главным образом на языковые стандарты C++14 и C++17.

Примеры кода созданы в предположении, что у читателя есть работающий компилятор с поддержкой стандарта C++17. Можно, например, пользоваться компилятором GCC, как автор этой книги, или Clang. Последние выпущенные версии обоих этих компиляторов одинаково хорошо поддерживают все элементы стандарта C++17, использованные в данной книге. Все примеры кода протестированы с версиями[1] GCC 7.2 и Clang 5.0.

Для работы с примерами можно пользоваться обычным текстовым редактором и запускать компилятор вручную из командной строки через посредство утилиты GNU `make` (к каждому примеру приложен несложный сценарий Makefile); также можно воспользоваться полновесной интегрированной средой разработки наподобие QtCreator (www.qt.io), Eclipse (www.eclipse.org), Kdevelop (www.kdevelop.org) или CLion (www.jetbrains.com/clion) и импортировать в нее примеры. Тем, кто намерен пользоваться средой Microsoft Visual Studio, рекомендуется установить наиболее свежую версию, какую возможно загрузить, и настроить ее на применение компилятора Clang вместо используемого по умолчанию компилятора Microsoft Visual C++ (MSVC), которому на момент написания этого введения не хватало некоторых важных мелочей.

---

[1] На момент перевода этого введения актуальны версии GCC 9.1 и Clang 8.0.0. Качество поддержки стандарта C++17 более не должно составлять предмета для беспокойства. – *Прим. перев.*

Хотя для компиляции большей части примеров не нужны никакие внешние зависимости, некоторым примерам все же нужны сторонние библиотеки, такие как range-v3, catch и JSON, клоны которых доступны вместе с кодом примеров в директории common/3rd-party, а также коллекция библиотек Boost, которую можно загрузить с сайта www.boost.org.

Оформление кода и загрузка примеров

Исходный код примеров к этой книге можно загрузить на сайте издательства по адресу www.manning.com/books/functional-programming-in-c-plus-plus и с системы GitLab по адресу https://gitlab.com/manning-fp-cpp-book.

В книге содержится много примеров исходного кода – как в виде нумерованных листингов, так и короткими вставками в обычный текст. В обоих случаях код набран `моноширинным шрифтом наподобие этого`, чтобы его легко было отличить от текста на естественном языке.

Во многих случаях первоначальный текст примеров пришлось видоизменить, добавив разрывы строк и отступы, чтобы улучшить размещение исходного кода на книжной странице. В редких случаях даже этого оказалось недостаточно, и тогда в листинг пришлось ввести знаки продолжения строки ➥. Кроме того, комментарии из исходного кода часто удалялись, если этот код сопровождается подробными пояснениями в основном тексте. Пояснения прилагаются ко многим листингам, помогая понять важные детали.

Спорить о стилях оформления исходного кода – это превосходная возможность впустую потратить уйму времени. Это особенно характерно для языка C++, где едва ли не каждый проект претендует на собственный стиль.

Автор старался следовать соглашениям, используемым в ряде других книг по языку C++. Два стилистических правила стоит оговорить здесь особо:

- классы, моделирующие сущности из реального мира, например людей или домашних животных, получают имена с суффиксом `_t`. Благодаря этому становится проще понять в каждом конкретном случае, идет речь о реальном объекте (например, человеке) или о типе – имя `person_t` читается проще, чем «тип “человек”»;
- имена закрытых переменных-членов имеют приставку `m_`. Это отличает их от статических переменных-членов, чьи имена начинаются с префикса `s_`.

Форум книги

Покупка этой книги дает бесплатный доступ к закрытому форуму, функционирующему под управлением издательства Manning Publications, где читатели могут оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы

получить доступ к форуму, нужно зайти на страницу https://forums.manning.com/forums/functional-programming-in-c-plus-plus. Больше о форумах издательства Manning и о принятых там правилах поведения можно узнать на странице https://forums.manning.com/forums/about.

Издательство Manning берет на себя обязательство перед своими читателями обеспечить им удобную площадку для общения читателей как между собой, так и с автором книги. Это, однако, не предполагает какого-либо определенного объема участия со стороны автора, чье общение на форуме остается исключительно добровольным и неоплачиваемым. Задавая автору интересные вопросы, читатели могут освежать и подогревать его интерес к теме книги. Форум и архивы предыдущих обсуждений будут доступны на сайте издательства все время, пока книга остается в продаже.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Об авторе

Иван Чукич преподает современные методы программирования на языке C++ и функциональное программирование на факультете математики в Белграде. Он использует язык C++ с 1998 г. Он исследовал функциональное программирование на языке C++ перед и во время подготовки своей диссертации, а также применяет методы функционального программирования в реальных проектах, которыми пользуются сотни миллионов человек по всему миру. Иван – один из ключевых разработчиков среды KDE, крупнейшего в мире проекта с открытым кодом на языке C++.

1 Введение в функциональное программирование

О чем говорится в этой главе:

- понятие о функциональном программировании;
- рассуждение в терминах предназначения вместо шагов алгоритма;
- понятие о чистых функциях;
- преимущества функционального программирования;
- эволюция C++ в язык функционального программирования.

Каждому программисту приходится изучить за свою жизнь целый ряд языков программирования. Как правило, программист останавливается на двух или трех языках, на которых лично ему удобнее всего работать. Часто можно услышать, что изучить очередной язык программирования просто – что различия между языками в основном касаются лишь синтаксиса и что все языки предоставляют примерно одинаковые возможности. Тот, кто знает язык C++, наверняка легко выучит языки Java и C#, и наоборот.

В этом утверждении есть доля истины. Однако, берясь за изучение нового языка, мы невольно пытаемся имитировать на нем тот стиль программирования, который выработался в связи с предыдущим языком. Когда я впервые начал применять функциональный язык во время учебы в университете, то сразу начал с попыток определить на нем привычные операторы цикла `for` и `while` и оператор ветвления `if-then-else`. Это именно тот подход, которым пользуется большинство из нас, чтобы просто сдать экзамен и никогда больше не возвращаться к изученному.

Широко известен афоризм, что тот, у кого из инструментов есть лишь молоток, будет стремиться любую задачу считать гвоздем. Этот принцип

работает и в обратную сторону: если работать только с гвоздями, то любой попавший инструмент будет использоваться как молоток. Многие программисты, попробовав язык функционального программирования, решают, что он не стоит затраченных усилий; они не видят в новом языке преимуществ, поскольку пытаются использовать этот новый инструмент таким же способом, как использовали бы старый.

Цель этой книги состоит не в том, чтобы научить читателя новому языку программирования; вместо этого книга призвана научить иному способу использования старого языка (а именно языка C++) – способу, настолько отличному от привычного, что у программиста впрямь может возникнуть *ощущение*, что он использует новый язык. Этот новый стиль программирования помогает создавать более продуманные программы и писать более безопасный, понятный и читаемый код и даже – не побоюсь сказать – более изящный, чем код, который обычно пишут на языке C++.

1.1 Что такое функциональное программирование

Функциональное программирование – это довольно старая парадигма, зародившаяся в академической среде в конце 1950-х годов и долгое время остававшаяся в этой нише. Хотя эта парадигма всегда была излюбленной темой для научных исследований, она никогда не пользовалась популярностью в «реальном мире». Вместо этого повсеместное распространение получили императивные языки: сперва процедурные, затем объектно-ориентированные.

Часто звучали предсказания, что однажды функциональные языки будут править миром, но этого до сих пор не произошло. Наиболее известные языки функционального программирования, такие как Haskell или LISP, все еще не входят в десятку наиболее популярных языков. Первые места по традиции прочно занимают императивные языки, к которым относятся языки C, Java, и C++. Это предсказание, подобно большинству других, чтобы считаться сбывшимся, нуждается в известной свободе интерпретации. Вместо того чтобы популярность завоевали языки функционального программирования, произошло нечто иное: в популярные языки программирования стали добавлять все более элементов, заимствованных из языков функционального программирования.

Что *собой представляет* функциональное программирование (ФП)? На этот вопрос непросто ответить, так как не существует единого общепринятого определения. Согласно известному афоризму, если двух программистов, работающих в функциональном стиле, спросить, что такое ФП, они дадут по меньшей мере три различных ответа. Каждый специалист обычно старается определить ФП через другие связанные с ним понятия: чистые функции, ленивые вычисления, сопоставление с образцом и другие, и обычно при этом перечисляет характеристики своего любимого языка.

Чтобы не оттолкнуть читателя, уже проникшегося симпатией к тому или иному языку, начнем с чисто математического определения, взятого из группы Usenet, посвященной функциональному программированию:

> *Функциональное программирование – это стиль программирования, в котором основную роль играет вычисление выражений, а не выполнение команд. Выражения в таких языках образованы из функций, применяемых к исходным значениям. Функциональный язык – это язык, который поддерживает функциональный стиль и способствует программированию в этом стиле.*
>
> – FAQ из группы comp.lang.functional

В последующих главах этой книги будет разобран ряд понятий, относящихся к ФП. Оставим на усмотрение читателя выбор тех из них, которые захочется считать существенными признаками, за которые язык можно назвать *функциональным*.

Говоря шире, ФП – это стиль программирования, в котором основными блоками для построения программы являются функции в противоположность объектам или процедурам. Программа, написанная в функциональном стиле, не задает команды, которые должны быть выполнены для получения результата, а определяет, что есть этот результат.

Рассмотрим небольшой пример: вычисление суммы списка чисел. В императивном мире реализация состоит в том, чтобы перебрать один за другим все элементы списка, прибавляя значение каждого из них к переменной-накопителю. Программный текст представляет собой пошаговое описание процесса суммирования элементов списка. В функциональном же стиле, напротив, нужно лишь определить, что есть сумма чисел, и тогда компьютер сам будет знать, какие операции выполнить, когда ему понадобится вычислить сумму. Один из способов добиться этого состоит в том, что сумма списка чисел есть результат сложения первого элемента списка с суммой оставшейся части списка; кроме того, сумма равна нулю, если список пуст. Тем самым дано определение, что есть сумма, без подробного описания того, как ее вычислить.

Это различие и лежит в основе терминов «императивное программирование» и «декларативное программирование». *Императивное* программирование означает, что программист указывает компьютеру выполнить нечто, в явном виде описав каждый шаг, необходимый для вычисления результата. *Декларативное* же означает, что программист описывает требования к ожидаемому результату, а на язык программирования ложится ответственность за отыскание конкретного способа его получения. Программист определяет, что собой представляет сумма списка чисел, а язык использует это определение, чтобы вычислить сумму конкретного списка.

1.1.1 Соотношение функционального программирования с объектно-ориентированным

Невозможно однозначно сказать, какая из двух парадигм лучше: наиболее популярная из императивных – объектно-ориентированная (ООП) – или самая популярная из декларативных, т. е. парадигма ФП. У обеих есть свои сильные и слабые стороны.

Основная идея объектно-ориентированной парадигмы – это создание абстракций над данными. Этот механизм позволяет программисту скрывать внутреннее представление данных внутри объекта, предоставляя остальному миру лишь видимую его сторону, доступную через интерфейс.

В центре внимания ФП лежит создание абстракций над функциями. Это позволяет создавать структуры управления, более сложные, чем те, что непосредственно поддерживаются языком программирования. Когда в стандарт C++11 был добавлен цикл `for` по диапазону (часто называемый также циклом `foreach`), его поддержку нужно было реализовать во всех компиляторах, которых в мире имеется довольно много. С помощью методов ФП часто удается сделать нечто подобное без модификации компилятора. За прошедшие годы во многих сторонних библиотеках были реализованы собственные версии цикла по диапазону. Используя идиомы ФП, можно создавать новые языковые конструкции – например, цикл `for` по диапазону или иные, более сложные. Эти конструкции могут пригодиться, даже если разработка в целом ведется в императивном стиле.

В некоторых случаях удобнее оказывается одна парадигма программирования, в иных – другая. Часто самым верным решением оказывается комбинация обеих парадигм. Это видно уже из того факта, что многие старые и новые языки программирования стали поддерживать несколько парадигм, вместо того чтобы хранить верность своей первоначальной парадигме.

1.1.2 Сравнение императивной и декларативной парадигм на конкретном примере

Чтобы продемонстрировать различие между этими двумя стилями программирования, начнем с простой программы, написанной в императивном стиле, и преобразуем ее в эквивалентную функциональную программу. Одна из часто применяемых мер сложности программного обеспечения – это число строк кода (англ. lines of code, LOC). Хотя в общем случае адекватность этого показателя остается под вопросом, он превосходно подходит в качестве примера для демонстрации различий между императивным и функциональным стилями.

Предположим, нужно написать функцию, которая принимает на вход список файлов и подсчитывает число строк в каждом из них (рис. 1.1).

Чтобы сделать пример как можно проще, будем подсчитывать в каждом файле лишь число символов перевода на новую строку в предположении, что последняя строка файла непременно заканчивается этим символом.

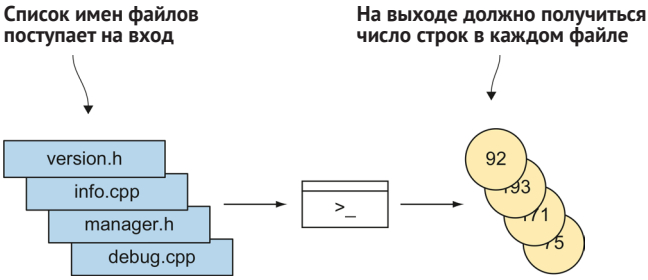

Рис. 1.1 Программа для подсчета числа строк в файлах

Рассуждая по-императивному, можно представить алгоритм решения этой задачи в виде такого списка шагов:

1. Открыть каждый файл.
2. Проинициализировать нулем счетчик строк.
3. Читать из файла символ за символом и наращивать счетчик всякий раз, когда попадается символ новой строки (\n).
4. Дойдя до конца файла, запомнить полученное значение счетчика строк.

Следующий листинг содержит реализацию описанной выше логики.

Листинг 1.1 Подсчет числа строк, императивное решение

```
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;
}
```

Получилось два вложенных цикла и несколько переменных, в которых хранится текущее состояние процесса. Несмотря на всю простоту этого примера, в нем есть несколько мест, где нетрудно ошибиться: например, можно оставить переменную непроинициализированной (или проинициализировать ее неправильно), неправильно изменить значение счетчика или неправильно указать условие цикла. Компилятор может выдать предупреждения о некоторых из них, но те, что пройдут компиляцию незамеченными, обычно с трудом обнаруживаются человеком, поскольку наш мозг устроен так, чтобы не замечать их, как мы не замечаем опечатки. Поэтому код лучше всего писать таким образом, чтобы свести к минимуму саму возможность подобных ошибок.

Читатели, более искушенные в языке C++, наверняка заметили, что в этом примере можно было воспользоваться стандартным алгоритмом `std::count`, вместо того чтобы подсчитывать переводы строки вручную. Стандартная библиотека языка C++ предоставляет ряд удобных абстракций, например итераторы по потокам, которые позволяют обращаться с потоками ввода-вывода таким же образом, как с обычными коллекциями: векторами и списками. Покажем пример их использования.

Листинг 1.2 Использование алгоритма `std::count` для подсчета символов перевода строки

```
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;
}
```

Подсчитывает символы перевода строки от текущей позиции в файловом потоке до его конца

Сохраняет результаты

С такой реализацией программист избавлен от необходимости знать детали того, как реализован подсчет. Программный код теперь прямо выражает, что нужно подсчитать количество символов перехода на новую строку в заданном входном потоке. В этом проявляется основная идея функционального стиля программирования: использовать абстракции, позволяющие описать *цель* вычисления, вместо того чтобы подробно

описывать *способ* ее достижения, – и именно этой цели служит большая часть техник, описанных в данной книге. По этой же причине парадигма функционального программирования тесно соседствует с обобщенным программированием, особенно если речь идет о языке C++: оба подхода позволяют программисту мыслить на более высоком уровне абстракции по сравнению с приземленным императивным взглядом на программу.

Можно ли назвать язык C++ объектно-ориентированным?

Меня всегда удивляло, что большинство разработчиков называют C++ объектно-ориентированным языком. Это удивительно потому, что в стандартной библиотеке C++ (которую часто называют стандартной библиотекой шаблонов – англ. Standard Template Library, STL) почти нигде не используется полиморфизм, основанный на наследовании, – ключевой элемент парадигмы ООП.

Библиотеку STL создал Александр Степанов, известный критик ООП. Он захотел создать библиотеку для обобщенного программирования и сделал это, воспользовавшись системой шаблонов языка C++ в сочетании с методами функционального программирования.

В этом состоит одна из причин столь широкого использования библиотеки STL в данной книге: хотя она и не является библиотекой для функционального программирования в чистом виде, в ней хорошо смоделированы многие понятия ФП, что делает библиотеку STL превосходной отправной точкой для вхождения в функциональный мир.

Преимущество этого подхода в том, что программисту нужно держать в голове меньше переменных состояния и становится возможным заняться тем, чтобы на высоком уровне абстракции формулировать предназначение программы – вместо подробного описания мелких шагов, которые необходимо сделать для получения нужного результата. Не нужно больше задумываться о том, как реализован подсчет. Единственная забота функции `count_lines` состоит в том, чтобы взять поступивший на вход аргумент (имя файла) и превратить его в нечто, понятное для функции `std::count`, т. е. в пару итераторов по потоку.

Теперь продвинемся еще на шаг в том же направлении и определим весь алгоритм в функциональном стиле, описав, *что* должно быть сделано, вместо описания того, *как* это сделать. В предыдущей реализации остался цикл `for` по диапазону, который применяет функцию ко всем элементам коллекции и собирает результаты в контейнер. Это широко распространенный шаблон, и было бы естественно ожидать его поддержки стандартной библиотекой языка программирования. В языке C++ именно для этого предназначен алгоритм `std::transform` (в других языках похожая функция обычно называется `map` или `fmap`). Реализация разобранной выше логики на основе алгоритма `std::transform` показана в следующем листинге. Функция `std::transform` перебирает элементы коллекции `files` один за другим, преобразовывает каждый из них по-

средством функции `count_lines`, а полученные результаты складывает в вектор `results`.

Листинг 1.3 Отображение имен файлов в счетчики строк функцией `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(),   <-- Откуда брать исходные данные
                   results.begin(),                <-- Куда сохранять результаты
                   count_lines);                   <-- Функция-преобразователь

    return results;
}
```

В этом коде описан уже не пошаговый алгоритм, а, скорее, преобразование, которому нужно подвергнуть входные данные, чтобы получить желаемый результат. Можно утверждать, что, избавив программу от переменных состояния и взяв за основу стандартную реализацию алгоритма подсчета, вместо изобретения собственной делает код более устойчивым к потенциальным ошибкам.

Получившийся листинг все еще имеет недостаток: в нем слишком много клише, чтобы его можно было считать более удобочитаемым, чем первоначальный вариант. На самом деле в этом коде лишь три важных слова:

- `transform` – что код делает[1];
- `files` – входные данные;
- `count_lines` – функция-преобразователь.

Все прочее – шелуха.

Эта функция могла бы стать гораздо более удобочитаемой, если бы можно было оставить только важные фрагменты и опустить все остальное. Как читатель увидит в главе 7, этого можно добиться с помощью библиотеки ranges. Покажем здесь, как будет выглядеть эта функция, если реализовать ее на основе библиотеки ranges, через диапазоны и их преобразования. Операция конвейера, обозначаемая вертикальной чертой (|), используется в библиотеке ranges, для того чтобы подать коллекцию на вход преобразователя.

Листинг 1.4 Преобразование коллекции средствами библиотеки ranges

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

[1] То есть поэлементное преобразование контейнера исходных данных в контейнер результатов. – *Прим. перев.*

Этот код делает то же самое, что и код из листинга 1.3, но смысл в данном случае более очевиден. Функция берет поступивший на вход список имен файлов, пропускает каждый элемент через функцию-преобразователь и возвращает список результатов.

Соглашения о способе записи типов функций

В языке C++ нет единого типа, представляющего понятие функции (в главе 3 речь пойдет обо всем разнообразии сущностей, которые в языке C++ могут рассматриваться как функции). Чтобы иметь возможность зафиксировать исключительно типы аргументов и возвращаемого значения функции, не задавая при этом точный тип, который она получит в языке C++, нам понадобится новая, отвлеченная от языка система обозначений.

Когда впредь мы будем писать `f: (arg1_t, arg2_t, ..., argn_t) → result_t`, это будет означать, что `f` есть функция, принимающая `n` аргументов, первый из которых имеет тип `arg1_t`, второй – тип `arg2_t` и т. д., причем возвращает эта функция значение типа `result_t`. Если функция принимает лишь один аргумент, скобки вокруг его типа будем опускать. Кроме того, не будем пользоваться в этих обозначениях константными ссылками.

Например, если написано, что функция `repeat` имеет тип `(char, int) → std::string`, это означает, что она принимает два аргумента: один символьного типа и один целочисленного – и возвращает строку. В языке C++ это можно было бы записать, например, так (второй вариант допустим начиная со стандарта C++11):

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

Реализация на основе диапазонов и их преобразований также упрощает поддержку кода. Читатель мог заметить, что функция `count_lines` страдает проектным недостатком. Глядя лишь на ее имя и тип `count_lines: std::string → int`, легко понять, что она принимает один аргумент текстового типа, но совершенно не очевидно, что этот текст должен представлять собой имя файла. Естественно было бы предположить, что функция подсчитывает число строк в поступившем на вход тексте. Чтобы избавиться от этого недочета, можно функцию разбить на две: функцию `open_file: std::string → std::ifstream`, которая принимает имя файла и возвращает открытый файловый поток, и функцию `count_lines: std::ifstream → int`, которая подсчитывает строки в заданном потоке символов. После такого разбиения предназначение функций становится совершенно очевидно из их имен и типов аргументов и возвращаемых значений. При этом построенная на диапазонах реализация функции `count_lines_in_files` потребует вставки лишь еще одного преобразования.

Листинг 1.5 Преобразование коллекции средствами библиотеки ranges, модифицированный вариант

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

В итоге решение получилось гораздо менее многословным, чем императивное решение из листинга 1.1, и гораздо более очевидным. На вход поступает коллекция имен файлов – не имеет значения даже, какой тип коллекции используется, для каждого ее элемента выполняются подряд два преобразования. А именно сначала из имени файла создается файловый поток, затем в потоке подсчитываются символы перевода строки. Именно это в точности выражает приведенный здесь код, без обременительных подробностей и дублирующегося кода.

1.2 Чистые функции

Один из главных источников ошибок при программировании – это наличие у программы изменяемого состояния. Очень сложно проследить все возможные состояния, в которых программа может находиться. Парадигма ООП дает возможность сгруппировать отдельные части состояния в объекты, тем самым помогая человеку понять состояния и управлять ими. Однако этот подход не в силах существенно уменьшить число возможных состояний.

Представим себе разработку текстового редактора. Набранный пользователем текст нужно сохранить в переменную. Предположим, что пользователь нажимает кнопку **Сохранить** и продолжает набор текста. Программа тем временем сохраняет текст, отправляя символы в файл по одному (эта картина, конечно, сильно упрощена, но автор просит читателя запастись терпением). Что произойдет, если пользователь изменит часть текста, пока программа занимается его сохранением в файл? Сохранит ли программа текст в том виде, в котором он пребывал на момент нажатия кнопки **Сохранить**, или в его нынешнем виде, или сделает что-то иное?

Проблема состоит в том, что все три случая могут произойти – это зависит от того, насколько далеко продвинулась запись текста в файл и в какую часть текста пользователь внес изменение. Так, в случае, изображенном на рис. 1.2, программа запишет в файл текст, которого никогда не было в редакторе.

В файл будут сохранены участки, взятые из текста, до того, как пользователь внесет свои изменения, другие участки попадут в файл из уже измененного текста. Таким образом, в файл вместе попадут куски двух различных состояний.

Рис. 1.2 Повреждение файла в результате модификации текста во время записи

Описанная ситуация просто не могла бы возникнуть, если бы функция сохранения текста в файл обладала собственной неизменяемой копией данных, которые ей необходимо записать (рис. 1.3). В этом состоит наиболее серьезная проблема изменяемых состояний: они создают зависимости между частями программы, которые никак не должны быть связаны между собой. Данный пример включает в себя два очевидно различных пользовательских действия: набор текста и сохранение набранного текста. Эти действия должны выполняться независимо друг от друга. Параллельное выполнение двух или более действий, относящихся к одному и тому же общему состоянию, создает нежелательную зависимость между действиями и способно привести к проблемам, подобным описанной выше.

Майкл Фезерс (Michael Feathers), автор книги «Эффективная работа с устаревшим кодом» (Working Effectively with Legacy Code – Prentice Hall, 2004), писал: «ООП делает код более понятным за счет инкапсуляции движущихся частей. ФП делает код более понятным, сокращая число движущихся частей». Даже локальные изменяемые переменные стоит считать нежелательными по той же причине. Они создают зависимости между различными частями функции, затрудняя выделение тех или иных ее участков в отдельные вспомогательные функции.

Одно из наиболее мощных понятий ФП – это понятие *чистой функции*, т. е. функции, которая использует лишь значения поступающих на вход аргументов для вычисления результата, но не изменяет аргументы. Если чистую функцию вызывают несколько раз с одними и теми же аргументами, она обязана возвращать одно и то же значение и не должна оставлять никаких следов своего вызова (т. е. не должна производить *побочные эффекты*). Все это означает, что чистые функции не могут менять состояние программы.

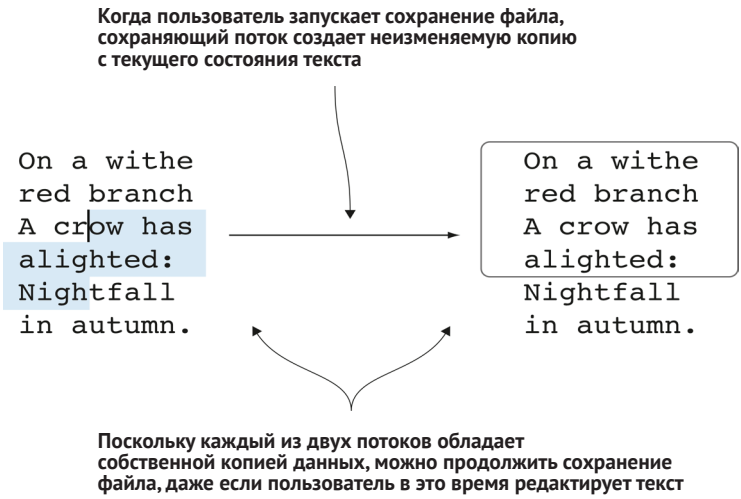

Рис. 1.3 Создание полной копии данных или использование структуры данных, запоминающей историю своих версий, позволяет разорвать зависимость между редактированием и сохранением текста

Чистые функции – это прекрасно, поскольку в этом случае программисту вообще не нужно заботиться о состоянии программы. Однако, к сожалению, чистота также означает, что функция не может читать из стандартного потока ввода, писать в стандартный поток вывода, создавать или удалять файлы, вставлять строки в базу данных и т. д. Если стремление к чистоте функций довести до предела, следовало бы запретить функциям даже изменять содержимое регистров процессора, оперативной памяти и тому подобного на аппаратном уровне.

Все это делает определение чистых функций практически непригодным для реального использования. Центральный процессор выполняет команды одна за другой, и ему необходимо следить за тем, какую команду выполнять следующей. Ничего нельзя выполнить на компьютере без того, чтобы изменить, по меньшей мере, внутреннее состояние процессора. Кроме того, невозможно создать сколько-нибудь полезную программу без взаимодействия с пользователем или другой программной системой.

Вследствие этого нам придется несколько ослабить требования и уточнить определение: *чистой* будем называть такую функцию, у которой отсутствуют наблюдаемые (на некотором высоком уровне) побочные эффекты. Клиентский код, вызывающий функцию, должен быть не в состоянии обнаружить какие-либо следы того, что функция была на самом деле выполнена, за исключением возврата ею значения. В этой книге мы не будем ограничивать себя использованием и написанием одних лишь чистых функций, однако будем всячески стараться к ограничению числа нечистых функций в наших программах.

1.2.1 Устранение изменяемого состояния

Разговор об ФП как стиле программирования начинался с разбора императивной реализации алгоритма, подсчитывающего символы перевода строки в коллекции файлов. Функция подсчета строк должна всегда возвращать один и тот же контейнер целых чисел, если ее многократно вызывать для одного и того же списка файлов (при условии что никакая внешняя функция не меняет сами файлы). Это означает, что нашу функцию можно реализовать чистым образом.

Глядя на первоначальную реализацию этой функции, показанную на листинге 1.1, можно увидеть лишь несколько операторов, нарушающих требование чистоты:

```
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);
}
```

Вызов метода `.get` для входного потока меняет как сам поток, так и значение, хранящееся в переменной `c`. Далее этот код изменяет контейнер `results`, добавляя в конец новые значения, и модифицирует переменную `line_count`, наращивая ее на единицу. На рис. 1.4 показано, как меняется состояние при обработке одного файла. Таким образом, данная функция, очевидно, реализована отнюдь не чистым образом.

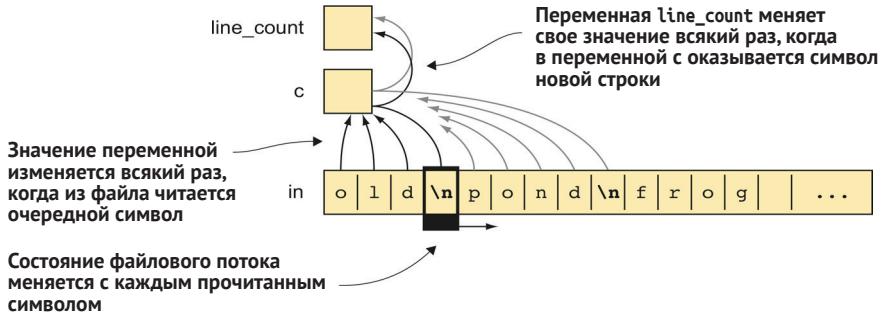

Рис. 1.4 Модификация нескольких независимых переменных при подсчете числа строк в файле

Однако наличие изменяющегося состояния – это не все, о чем следует задуматься. Еще один важный аспект – проявляется ли вовне нечистый

характер функции. В данном случае все изменяющиеся переменные локальны, через них не могут взаимодействовать даже параллельные вызовы этой функции, тем более они невидимы для вызывающего кода и каких бы то ни было внешних сущностей. Поэтому пользователи этой функции могут считать ее чистой, несмотря на то что ее внутренняя реализация не такова. Выгода для вызывающего кода очевидна, так как они могут быть уверены, что функция не изменит их состояния. С другой стороны, автор функции должен позаботиться о корректности ее внутреннего состояния. При этом нужно также удостовериться, что функция не изменяет ничего, что ей не принадлежит. Конечно, лучше всего было бы ограничить даже внутреннее состояние и сделать реализацию функции как можно более чистой. Если при реализации программы удастся использовать исключительно чистые функции, то вообще не нужно беспокоиться о возможных паразитных влияниях между состояниями, поскольку никаких изменяемых состояний в программе нет.

Во втором решении (листинг 1.2) подсчет выделен в функцию с именем `count_lines`. Эта функция снаружи выглядит чистой, хотя ее внутренняя реализация создает и модифицирует поток ввода. К сожалению, это лучшее, чего можно добиться с помощью класса `std::ifstream`:

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

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

Этот шаг не приводит к сколько-нибудь существенному улучшению функции `count_lines_in_files`. Некоторые элементы реализации, противоречащие чистоте функции, перемещаются из нее в другое место, однако в ней по-прежнему остается изменяемое состояние: переменная `results`. Функция `count_lines_in_files` сама не осуществляет никакого ввода-вывода и реализована исключительно на основе функции `count_lines`, которая с точки зрения вызывающего контекста выглядит чистой. Поэтому нет никаких причин, по которым функции `count_lines_in_files` следовало бы быть нечистой. Следующая версия кода, основанная на диапазонах, представляет собой реализацию функции `count_lines_in_files` без какого бы то ни было локального состояния (изменяемого или нет). Эта реализация определяет функцию через вызов другой функции на поступившем на вход контейнере:

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

Данное решение – превосходный пример того, как должно выглядеть программирование в функциональном стиле. Код получился кратким и содержательным, а принцип его работы – вполне очевидным. Более того, код с полной очевидностью не делает ничего иного: у него нет видимых извне побочных эффектов. Он всего лишь вычисляет желаемый результат на заданных входных данных.

1.3 Функциональный стиль мышления

Писать код сначала в императивном стиле, а затем преобразовывать его по кусочкам, пока он не превратится в функциональный, было бы неэффективно и контрпродуктивно. Напротив, следует с самого начала по-иному осмысливать задачу. Вместо того чтобы рассуждать о шагах, которые должен выполнить алгоритм, нужно подумать, что собой представляют входные данные и искомый результат и какие преобразования нужно выполнить, чтобы отобразить одно в другое.

В примере, показанном на рис. 1.5, даны имена файлов и нужно подсчитать число строк в каждом из них. Первое, что бросается здесь в глаза: задачу можно упростить, обрабатывая каждый раз по одному файлу. Хотя на вход поступил целый список имен файлов, каждое из них можно рассматривать независимо от остальных. Если изобрести решение задачи для единственного файла, можно легко решить также и исходную задачу (рис. 1.6).

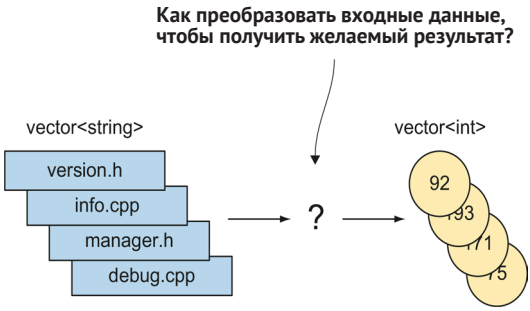

Рис. 1.5 Рассуждая о задаче в функциональном стиле, нужно думать о том, какое преобразование применить к входным данным, чтобы получить желаемый результат

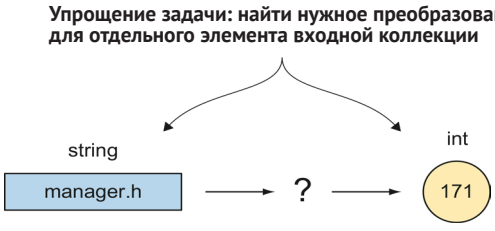

Рис. 1.6 Одно и то же преобразование применяется к каждому элементу коллекции. Это позволяет заняться более простой задачей: преобразованием единственного элемента вместо преобразования всей коллекции

Теперь важнейшей задачей становится создание функции, которая принимает на вход имя файла и подсчитывает число строк в файле с этим именем. Из этого определения вполне ясно, что на вход функции поступает что-то одно (имя файла), но для работы нужно ей что-то другое (а именно содержимое файла, чтобы было в чем подсчитывать символы перевода строки). Следовательно, нужна вспомогательная функция, которая может, получив на вход имя файла, предоставить его содержимое. Должно ли это содержимое предоставляться в виде строки, файлового потока или чего-то иного, остается всецело на усмотрение программиста. Нужна всего лишь возможность получать из этого содержимого по одному символу за раз, чтобы их можно было подать на вход функции, подсчитывающей среди них символы перевода строки.

Имея в руках функцию, которая по заданному имени файла отдает его содержимое (ее типом будет `std::string → std::ifstream`), к ее результату можно применить другую функцию, которая в заданном содержимом считает строки (т. е. функцию типа `std::ifstream → int`). Сочленение этих двух функций путем подачи файлового потока, созданного первой, на вход второй как раз и дает необходимую нам функцию, см. рис. 1.7.

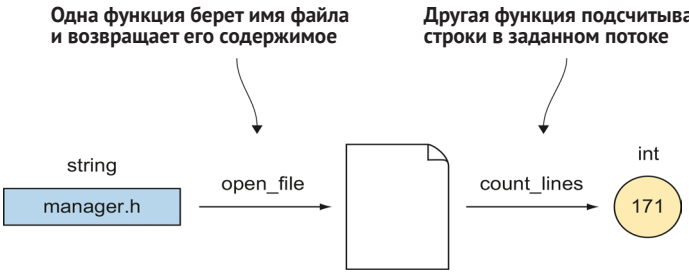

Рис. 1.7 Сложную задачу подсчета строк в файле с заданным именем можно разбить на две простые: открытие файла по заданному имени и подсчет строк в открытом файловом потоке

Тем самым задача подсчета строк в одном файле решена. Для решения же исходной задачи остается *поднять* (англ. lift) эти функции, чтобы они могли работать не над отдельными значениями, а над коллекциями значений. В сущности, это именно то, что делает функция `std::transform` (пусть и с более сложным интерфейсом): она берет функцию, которую можно применять к отдельно взятому значению, и создает на ее основе преобразование, которое можно применять к целым коллекциям значений, – см. рис. 1.8. Пока что читателю лучше всего представить себе *подъем* (англ. lifting) как обобщенный способ превратить функцию, оперирующую отдельными простыми значениями некоторого типа, в функцию, работающую с более сложными структурами данных, содержащими значения этого типа. Более подробно речь о подъеме функций пойдет в главе 4.

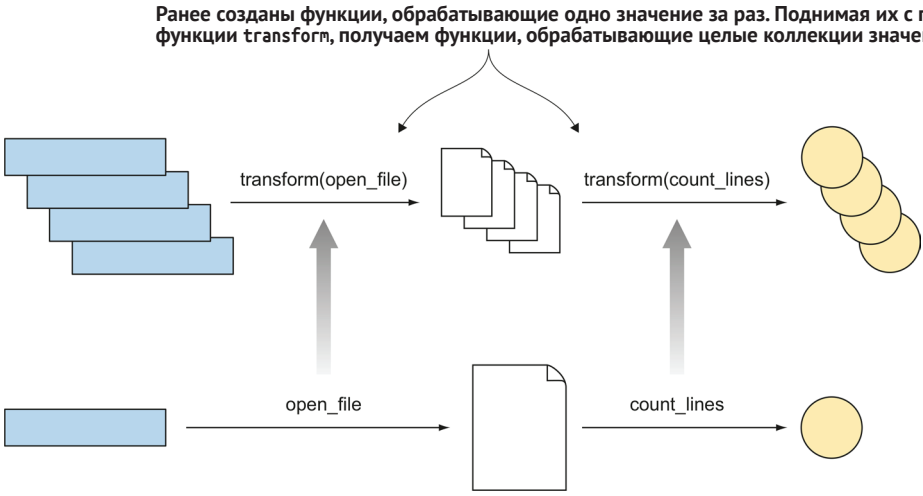

Рис. 1.8 С помощью функции `transform` можно создавать функции обработки коллекций на основе функций, обрабатывающих одно значение

Из этого простого примера хорошо виден функциональный подход к разбиению больших задач программирования на меньшие независимые подзадачи, решения которых затем легко сочленяются. Одна из полезных аналогий, помогающих представить себе композицию функций и их подъем, – это аналогия с движущейся лентой конвейера (рис. 1.9). В начале ленты находится сырье, в конце должно быть получено готовое изделие. Лента продвигается от станка к станку, каждый из которых совершает какое-то свое преобразование, пока не получится изделие в своем окончательном виде. Глядя на конвейер как целое, естественно рассуждать о цепочке преобразований, превращающей сырье в изделие, а не об элементарных действиях, совершаемых каждой машиной.

В этой аналогии сырье соответствует входным данным, а станки – функциям, применяемым последовательно к этим данным. Каждая функция узко специализирована для выполнения одной простой задачи и ничего не должна знать об остальной части конвейера. Каждой функции нужны только правильные данные на вход – но ей совершенно не важно, откуда они получены. Входные значения помещаются на конвейер одно за другим (или, возможно, на несколько параллельно работающих конвейерных лент, что позволяет обрабатывать несколько значений одновременно). Каждое значение подвергается преобразованиям, и с другого конца конвейера снимается коллекция результатов.

1.4 Преимущества функционального программирования

Каждый из многочисленных аспектов ФП обещает свои выгоды. Эти выгоды будут разобраны на протяжении всей книги, здесь же начнем с не-

скольких важнейших преимуществ, которые обеспечиваются большинством механизмов ФП.

Самое очевидное обстоятельство, которое отмечает большинство программистов, едва начав создавать программы в функциональном стиле, состоит в том, что код программ становится значительно короче. В некоторых проектах даже можно встретить разбросанные по коду комментарии наподобие «на языке Haskell это можно было бы реализовать в одну строку». Это происходит потому, что средства, предоставляемые функциональными языками, просты и в то же время очень выразительны, большую часть решения задачи они позволяют описать на высоком уровне абстракции, не вдаваясь в мелкие детали.

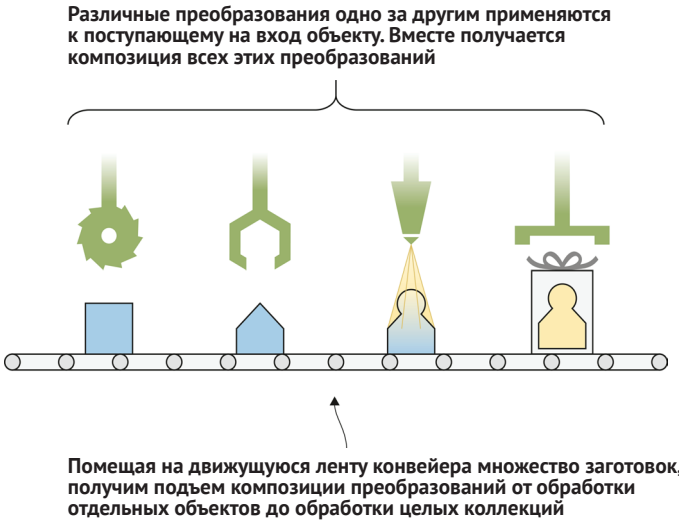

Рис. 1.9 Композицию функций и их подъем можно сравнить с движущейся лентой конвейера. Каждое отдельное преобразование работает над одним объектом. Подъем позволяет этим преобразователям работать над коллекциями объектов. Композиция подает результат одного преобразования на вход другого. В целом получается конвейер, применяющий последовательность операций к любому числу исходных объектов

Именно это свойство, вместе со свойством чистоты, стало привлекать к функциональному программированию в последние годы все больше внимания. Чистота способствует корректности кода, а выразительность позволяет писать меньше кода, то есть оставляет меньше места для ошибок.

1.4.1 Краткость и удобочитаемость кода

Приверженцы функциональной парадигмы обычно утверждают, что программы, написанные в функциональном стиле, проще понимать. Это суждение субъективно, и люди, привыкшие писать и читать императив-

ный код, могут с ним не согласиться. Объективно же можно утверждать, что программы, написанные в функциональном стиле, обычно оказываются короче и лаконичнее. Это с полной очевидностью проявилось в разобранном примере: начав с двадцати строк императивного кода, мы пришли к единственной строке кода для функции `count_lines_in_files` и примерно пяти строчкам для функции `count_lines`, из которых большую часть составляет клише, навязанное языком C++ и библиотекой STL. Достичь такого впечатляющего сжатия кода удалось благодаря использованию высокоуровневых абстракций, предоставляемых той частью библиотеки STL, что ориентирована на поддержку ФП.

Одна из печальных истин состоит в том, что многие программисты, работающие на языке C++, сторонятся использования высокоуровневых абстракций наподобие алгоритмов из библиотеки STL. У них могут быть для этого различные причины, от стремления написать своими руками более эффективный код до желания избежать кода, который коллегам было бы тяжело понять. Иногда эти причины не лишены смысла, но отнюдь не в большинстве случаев. Отказ от пользования наиболее передовыми возможностями своего языка программирования снижает мощь и выразительность языка, делает код более запутанным, а его поддержку – более трудоемкой.

В 1968 г. Эдсгер Дейкстра опубликовал знаменитую статью «О вреде оператора goto»[1]. В ней он выступал за отказ от оператора перехода goto, который в ту эпоху использовался чрезмерно, в пользу высокоуровневых средств структурного программирования, включая процедуры, операторы ветвления и цикла.

> *Разнузданное применение оператора go to имеет прямым следствием то, что становится ужасно трудно найти осмысленный набор координат, в которых описывается состояние процесса… Оператор go to сам по себе просто слишком примитивен; он создает слишком сильное побуждение внести путаницу в программу*[2].

Однако во многих случаях операторы цикла и ветвления тоже оказываются слишком примитивными. Как и оператор goto, циклы и ветвления могут сделать программу сложной для написания и понимания, и их часто удается заменить более высокоуровневыми конструкциями в функциональном стиле. Программисты, привычные к императивному стилю, часто дублируют один и тот же, по сути, код в разных местах программы, даже не замечая, что он одинаков, так как он работает с данными разных типов или имеет мелкие различия в поведении, которые с точки зрения функционального стиля легко было бы вынести в параметр.

---

[1] *Edsger Dijkstra.* Go To Statement Considered Harmful // Communications of the ACM: journal. 1968. March (vol. 11, no. 3). P. 147–148.

[2] Цит. по русскому переводу: http://hosting.vspu.ac.ru/~chul/dijkstra/goto/goto.htm. – *Прим. перев.*

Используя существующие абстракции, предоставляемые библиотекой STL или какими-либо сторонними библиотеками, или создавая собственные абстракции, программист может сделать код своих программ надежнее и короче. Кроме того, проще становится обнаруживать ошибки в этих абстракциях, так как один и тот же абстрактный код будет использоваться во множестве разных мест в программе.

1.4.2 Параллельная обработка и синхронизация

Главную трудность при разработке параллельных систем составляет управление общим изменяемым состоянием. От программиста требуется особая внимательность и аккуратность, чтобы обеспечить согласованную работу разных частей системы и целостность состояния.

Распараллеливание программ, созданных на основе чистых функций, напротив, тривиально, поскольку эти функции не изменяют никакого состояния. Поэтому нет необходимости синхронизировать их своими руками с помощью атомарных переменных или семафоров. Код, написанный для однопоточной системы, можно почти без изменений запускать в несколько потоков. Более подробно речь о этом пойдет в главе 12.

Рассмотрим следующий фрагмент кода, который суммирует квадратные корни значений, хранящихся в векторе `xs`:

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

Если реализация функции `sqrt` чистая (а для иного причин нет), реализация функции `sum` может автоматически разбить вектор входных данных на сегменты и сумму каждого из них вычислить в отдельном потоке. Когда все потоки завершатся, останется лишь собрать и просуммировать их результаты.

К сожалению, в языке C++ (по крайней мере, пока) отсутствует понятие чистой функции, поэтому распараллеливание не может выполняться автоматически. Вместо этого программисту следует в явном виде вызвать параллельную версию функции `sum`. Параллельная функция `sum` может даже сама на лету определить число процессорных ядер и, исходя из этого, решить, на сколько сегментов разбить вектор `xs`. Если бы суммирование корней было реализовано в императивном стиле, через цикл `for`, его бы не удалось распараллелить столь просто. Тогда пришлось бы позаботиться о том, чтобы переменные не модифицировались одновременно разными потоками, об определении оптимального числа потоков для той системы, на которой система выполняется, – вместо того чтобы предоставить все это библиотеке, реализующей алгоритм суммирования.

ПРИМЕЧАНИЕ Компиляторы языка C++ иногда способны сами выполнять векторизацию и ряд других оптимизаций, если обнаруживают, что тело цикла чистое. Эти оптимизации часто затрагивают код, использующий стандартные алгоритмы, поскольку они обычно бывают реализованы на основе циклов.

1.4.3 Непрерывная оптимизация

Использование высокоуровневых программных абстракций из библиотеки STL или других заслуживающих доверия библиотек приносит еще одну ощутимую выгоду: программа может со временем совершенствоваться, даже если автор не меняет в ее коде ни строчки. Каждое усовершенствование в языке программирования, реализации компилятора или в реализации используемой программой библиотеки будет автоматически приводить к совершенствованию программы. Хотя это равно справедливо как для функциональных, так и для нефункциональных абстракций высокого уровня, само использование понятий ФП в программе существенно увеличивает долю кода, основанного на этих абстракциях.

Это преимущество может показаться самоочевидным, но многие программисты до сих пор предпочитают вручную писать критичные по производительности участки низкоуровневого кода, иногда даже на языке ассемблера. Этот подход может быть оправдан, но в большинстве случаев он позволяет оптимизировать код лишь для конкретной вычислительной платформы и практически лишает компилятор возможности самому оптимизировать код для всех остальных платформ.

Рассмотрим для примера функцию `sum`. Ее можно оптимизировать для системы с упреждающей выборкой машинных команд, если в теле цикла обрабатывать сразу по два (или более) элемента на каждой итерации, – вместо того чтобы прибавлять числа по одному. Это уменьшит общее количество команд перехода в машинном коде, поэтому упреждающая выборка будет срабатывать чаще. Производительность программы на данной целевой платформе, очевидно, возрастет. Но что будет, если эту же программу скомпилировать и запустить на другой платформе? На некоторых платформах оптимальным может оказаться цикл в первоначальном виде, обрабатывающий по одному элементу за итерацию, на иных же может оказаться лучше выбирать за итерацию другое количество элементов. Некоторые системы могут даже поддерживать машинную команду, которая делает все, что нужно данной функции.

Пытаясь оптимизировать код вручную, легко потерять оптимальность на всех платформах, кроме одной. Напротив, пользуясь в программе высокоуровневыми абстракциями, программист полагается на труд множества других людей по оптимизации кода для разных частных случаев. Большинство реализаций библиотеки STL содержат оптимизации, специфичные для тех или иных платформ и компиляторов.

1.5 Эволюция C++ как языка функционального программирования

Язык C++ возник как расширение языка C, позволяющее писать объектно-ориентированный код, и поначалу вообще назывался «C с классами». Однако даже после выхода первого стандарта языка (C++98) его

сложно было бы назвать объектно-ориентированным. С введением в язык шаблонов и с появлением библиотеки STL, в которой наследование и виртуальные методы используются лишь эпизодически, язык C++ стал в полном смысле мультипарадигмальным.

Если внимательно рассмотреть принципы библиотеки STL и ее реализацию, можно даже прийти к убеждению, что C++ – язык вовсе не объектно-ориентированного, а в первую очередь обобщенного программирования. В основе *обобщенного программирования* лежит идея о том, что код можно написать один раз, используя в нем те или иные общие понятия, а затем применять его сколько угодно раз к различным сущностям, подпадающим под эти понятия. Так, например, библиотека STL предоставляет шаблон класса `vector`, который можно использовать с разными типами данных, включая целые числа, строки или пользовательские типы, отвечающие определенным требованиям. Для каждого из этих типов компилятор генерирует хорошо оптимизированный код. Этот механизм часто называют *статическим полиморфизмом*, или *полиморфизмом времени компиляции*, – в противоположность *динамическому полиморфизму*, или *полиморфизму времени выполнения*, который поддерживается наследованием и виртуальными методами.

Что касается поддержки функционального стиля языком C++, то решающая роль шаблонов обеспечивается не столько поддержкой контейнерных классов наподобие векторов, сколько тем, что они сделали возможным появление в библиотеке STL набора универсальных общеупотребительных алгоритмов – таких как сортировка или подсчет. Большинство из них позволяет программисту передавать в качестве параметров собственные функции, тем самым подстраивая их поведение без использования[1] типа `void*`. Таким способом, например, можно изменить порядок сортировки или определить, какие именно элементы следует пересчитать, и т. д.

Возможность передавать функции в другие функции в качестве аргументов и возможность возвращать из функций в качестве значений новые функции (точнее, некоторые сущности, *ведущие себя* подобно функциям, о чем речь пойдет в главе 3) превратили даже первую стандартизированную версию языка C++ в язык функционального про-

---

[1] Напомним, что в языке C, не обладающем средствами обобщенного программирования, единственный способ создать универсальную функцию, способную работать с данными произвольных типов, состоит в том, чтобы передать в качестве аргумента указатель на функцию, принимающую аргументы через универсальный указатель типа `void*`. Эта последняя функция отвечает за приведение указателей `void*` к фактическому типу данных и специфику работы с этим типом. Например, универсальная функция сортировки из стандартной библиотеки языка C имеет прототип `void qsort(void* base, size_t num, size_t size, int (*compar)(const void*, const void*))`, и ее последний аргумент представляет собой указатель на пользовательскую функцию сравнения элементов. Хотя использование указателей `void*` позволяет, в принципе, определять алгоритмы, работающие для произвольных типов данных, этот механизм прямо противоречит идее строгой типизации и чреват причудливыми, трудно поддающимися обнаружению ошибками. – *Прим. перев.*

граммирования. Стандарты C++11, C++14 и C++17 принесли с собой еще несколько нововведений, заметно упрощающих программирование в функциональном стиле. Эти дополнительные возможности представляют собой главным образом синтаксический сахар, как, например, ключевое слово `auto` и лямбда-выражения (рассматриваемые в главе 3). Кроме того, существенным усовершенствованиям подверглись алгоритмы из стандартной библиотеки. Следующая версия стандарта должна появиться в 2020 году, и в ней ожидается еще большее количество нововведений (ныне пребывающих в статусе технических спецификаций), ориентированных на поддержку функционального стиля: например, диапазонов (range), концептов (concept) и сопрограмм (coroutine).

Эволюция международных стандартов языка C++

Для языка C++ существует стандарт, утвержденный международной организацией по стандартизации ISO. Каждая новая версия стандарта проходит через строго регламентированный процесс рассмотрения, прежде чем быть опубликованной. Сам по себе язык и его стандартная библиотека разрабатываются комитетом, каждое нововведение всесторонне обсуждается, ставится на голосование и лишь затем становится частью окончательного проекта изменений к новой версии стандарта. В конце, когда все изменения внесены в тело стандарта, он должен пройти через еще одно голосование – итоговое, которое проводится для каждого нового стандарта в организации ISO.

Начиная с 2012 года работа комитета разделена между рабочими группами. Каждая группа работает над отдельными элементами языка и, когда считает этот элемент вполне законченным, выпускает так называемую техническую спецификацию (TC). TC существуют отдельно от стандарта, в стандарт они могут войти позднее.

Цель TC состоит в том, чтобы разработчики могли опробовать новые языковые средства и обнаружить в них скрытые недоработки и огрехи до того, как эти средства войдут в новый стандарт. От производителей компиляторов не требуется воплощать TC, но обычно они это делают. Более подробную информацию можно найти на сайте комитета https://isocpp.org/std/status.

Хотя большую часть средств, о которых говорится в этой книге, можно использовать и с более ранними версиями языка C++, мы будем ориентироваться преимущественно на стандарты C++14 и C++17.

1.6 Что узнает читатель из этой книги

Эта книга предназначена в первую очередь для опытных разработчиков, которые используют язык C++ в своей повседневной работе и желают пополнить свой арсенал более мощными инструментами. Чтобы извлечь максимум пользы из изучения этой книги, нужно хорошее знакомство с такими основополагающими элементами языка C++, как система типов, ссылки, спецификатор `const`, шаблоны, перегрузка операций, и др.

Нет необходимости знать нововведения, появившиеся в стандартах C++14/17, поскольку они подробно разобраны в книге; эти новые элементы еще не завоевали популярность, и многие читатели о них, скорее всего, еще не осведомлены.

Для начала будут разобраны такие важнейшие понятия, как функции высших порядков, что позволит повысить выразительность языка и сделать программы заметно короче. Также будет рассказано, как писать код, не имеющий изменяемого состояния, чтобы устранить необходимость в явной синхронизации при параллельном выполнении. Затем переключимся на вторую передачу и рассмотрим более сложные темы, такие как диапазоны (альтернатива алгоритмам обработки коллекций из стандартной библиотеки, хорошо поддерживающая композицию операций) и алгебраические типы данных (с помощью которых можно уменьшить число состояний, в которых может пребывать программа). Под конец поговорим об идиоме функционального программирования, вызывающей больше всего споров, – о пресловутых *монадах* и о том, как использовать монады для создания сложных и хорошо стыкующихся между собой систем.

К концу книги читатель будет в состоянии проектировать и реализовывать надежные параллельные системы, способные масштабироваться по горизонтали с минимальными усилиями; проектировать программу таким образом, чтобы свести к минимуму или вообще устранить возможность попадания в невалидное состояние вследствие сбоя или ошибки; представлять себе программу как поток данных сквозь череду преобразователей и использовать новое замечательное средство, диапазоны, для создания таких потоков, а также многое другое. Эти умения помогут создавать более компактный и менее подверженный ошибкам код, даже продолжая работать над объектно-ориентированными системами. Те, кто решит полностью погрузиться в функциональный стиль, получат возможность разрабатывать более стройные программные системы с удобными программными интерфейсами для взаимодействия с другими системами, в чем читатель убедится в главе 13, реализовав простой веб-сервис.

> **СОВЕТ** Более подробную информацию и дополнительные ресурсы по темам, затронутым в этой главе, можно найти на странице https://forums.manning.com/posts/list/41680.page.

Итоги

- Важнейший принцип функционального программирования состоит в том, что программисту не нужно брать на себя заботу о том, каким именно образом работает программа, – думать следует лишь о том, *что* она должна *сделать*.
- Оба подхода к программированию, функциональный и объектно-ориентированный, сулят программисту значительные выгоды. Нужно понимать, когда лучше применить первый подход, когда второй, а когда – их комбинацию.

- C++ представляет собой мультипарадигмальный язык программирования, с его помощью можно создавать программы в различных стилях: процедурном, объектно-ориентированном, функциональном, – а также комбинировать эти стили с использованием методов обобщенного программирования.
- Функциональное программирование идет рука об руку с обобщенным, особенно в языке C++. Оба подхода побуждают программиста не фиксировать внимание на аппаратном уровне, а поднимать уровень абстракции.
- Подъем функций позволяет создавать функции, работающие с коллекциями значений, на основе функций, обрабатывающих отдельные значения. С помощью композиции функций можно подать значение на цепочку преобразователей, в которой каждая функция передает свой результат на вход следующей функции.
- Избегание изменяемого состояния способствует корректности кода и устраняет необходимость в синхронизации параллельных потоков.
- Мыслить по-функциональному означает размышлять о входных данных и преобразованиях, которые нужно выполнить, чтобы получить желаемый результат.