Начала программирования

Elements of Programming

Alexander Stepanov,
Paul McJones

Addison-Wesley

Начала программирования

Александр Степанов,
Пол Мак-Джоунс

Москва • Санкт-Петербург • Киев
2011

ББК 32.973.26–018.2.75
С79
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С. Н. Тригуб*
Перевод с английского и редакция *К. А. Птицына*
Под редакцией *К. А. Птицына* и *А. А. Степанова*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Степанов, Александр Александрович, **Мак-Джонс**, Пол.

С79 Начала программирования. : Пер. с англ. — М. : ООО "И. Д. Вильямс", 2011. — 272 с. : ил. — Парал. тит. англ.

ISBN 978–5–8459–1708–9 (рус.)

ББК 32.973.26–018.2.75

Научно-популярное издание
Александр Александрович Степанов, Пол Мак-Джонс
Начала программирования

Литературный редактор *И. А. Попова*
Верстка *А. Н. Полинчик*
Художественный редактор *В. Г. Павлютин*
Корректор *Л. А. Гордиенко*

Подписано в печать 08.04.2011. Формат 70×100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 21,1. Уч.-изд. л. 14
Тираж 1000 экз. Заказ №0000

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978–5–8459–1708–9 (рус.)
ISBN 978–0–321–63537–2 (англ.)

Оглавление

Содержание

Дорогой читатель!

Две идеи, на которых основана эта книга, пришли мне в голову в середине 1970-х годов, когда я еще жил в Москве. Первая идея состояла в том, чтобы разбивать программы на небольшие (5–15 строк) алгоритмические части, каждая из которых сама по себе могла бы использоваться в других программах. Вторая идея состояла в том, что каждый такой алгоритм должен работать на любых типах данных, к которым он применим. Я называю их *обобщенными алгоритмами*. Мне казалось, что эти две идеи самоочевидны. К моему удивлению, мне много лет не удавалось изложить их удовлетворительно для самого себя. Несколько лет назад Пол Мак-Джонс уговорил меня вспомнить мою математическую молодость, и мы вместе написали *Начала программирования*.

Эта книга не учит программировать, не объясняет новую программную технологию, не пропагандирует новый язык программирования. Некоторым читателям может показаться, что она содержит случайный набор алгоритмов. Это не так. Надеюсь, что у читателя будет достаточно интереса, чтобы разобраться в структуре книги, ибо цель книги именно в том, чтобы показать глубокую взаимосвязь отдельных алгоритмов.

Надеюсь, что русский перевод нашей книги позволит мне хотя бы в малой мере воздать за то, что я получил от русской науки. Посвящаю это издание всем замечательным российским математикам, меня учившим.

Александр Александрович Степанов

Предисловие

В настоящей книге применяется дедуктивный подход к программированию, основанный на объединении программ с абстрактными математическими теориями, которые обеспечивают их работу. Представлены вместе описания этих теорий, алгоритмы, записанные в терминах этих теорий, а также теоремы и леммы, описывающие их свойства. Реализация алгоритмов на реальном языке программирования является центральной темой книги. Теоретические описания адресованы людям, поэтому в них научная строгость должна сочетаться с некоторой неформальностью; код предназначен для компьютера, поэтому должен быть абсолютно точным, даже если он имеет общее назначение.

Как и в других областях науки, в программировании наиболее подходящей основой является дедуктивный метод. Он обеспечивает декомпозицию сложных систем на компоненты с математически обоснованным поведением. Создание таких компонентов, в свою очередь, служит необходимой предпосылкой разработки эффективного, надежного, безопасного и экономически эффективного программного обеспечения.

Эта книга предназначена для тех, кто стремится глубже понять суть программирования, будь то профессиональные программисты или ученые и инженеры, для которых программирование составляет важную часть их профессиональной деятельности.

Книга предназначена для чтения от начала и до конца. Читатели смогут достичь понимания материала, только изучая код, доказывая леммы и выполняя упражнения. Кроме того, мы предлагаем несколько проектов, причем решения нам не известны. Изложение в книге является кратким, но внимательный читатель в конечном итоге сумеет обнаружить связь между ее частями и понять, чем мы руководствовались при выборе материала. Читатель должен поставить перед собой цель — раскрыть принципы построения книги.

Мы предполагаем наличие способности выполнять элементарные алгебраические манипуляции[1]. Мы также предполагаем, что читатель знаком с основ-

[1]Для проведения переподготовки по элементарной алгебре рекомендуем книгу [Chrystal, 1904].

ной терминологией логики и теории множеств на уровне базовых университетских курсов по дискретной математике; в приложении A приведены сводные данные об используемой нами системе обозначений. Мы приводим определения некоторых понятий абстрактной алгебры, необходимых для формулировки алгоритмов. Предполагаем также наличие развитых навыков программирования и понимания архитектуры ЭВМ[2] и знакомство с элементарными алгоритмами и структурами данных[3].

Мы выбрали язык C++, поскольку в нем механизмы абстракции сочетаются с точным представлением существующей машины[4]. Мы используем небольшое подмножество языка и записываем требования как структурированные комментарии. Мы надеемся, что следовать за изложением в книге смогут и читатели, не знакомые с языком C++. Подмножество языка, используемое в книге, определено в приложении B[5]. Везде, где имеется различие между математической системой обозначений и языком C++, типографский шрифт и контекст определяют, приведено ли описание математического обоснования или кода C++. У многих концепций и программ из этой книги есть параллели в STL (Standard Template Library — стандартная библиотека шаблонов C++), но некоторые решения, принятые в книге, отличаются от таковых в STL. В книге также игнорируются проблемы, которые должны быть решены в реальной библиотеке, такой как STL: пространства имен, видимость, встроенные директивы и т. д.

В главе 1 приведено описание значений, объектов, типов, процедур и концепций. Главы 2–5 посвящены алгоритмам на алгебраических структурах, таких как полугруппы и полностью упорядоченные множества. В главах 6–11 описываются алгоритмы на абстрактных представлениях памяти. В главе 12 рассматриваются объекты, содержащие другие объекты. В послесловии представлены наши размышления о подходе, изложенном в книге.

Благодарности

Мы благодарны компании Adobe Systems и ее руководителям за поддержку курса “Foundations of Programming” и этой книги, которая выросла из него. В частности, Грег Гилли (Greg Gilley) был инициатором проведения курса и предложил написать книгу; Дэйв Стори (Dave Story), а затем Билл

[2]Рекомендуем книгу [Patterson and Hennessy, 2007].

[3]См. [Tarjan, 1983].

[4]Стандартным справочником является [Stroustrup, 2000].

[5]Код из книги компилируется и выполняется в версиях Microsoft Visual C++ 9 и g++ 4. Этот код может быть загружен с сайта www.elementsofprogramming.com вместе с несколькими простейшими макросами, которые обеспечивают его компиляцию, а также тестовыми модулями.

Хенслер (Bill Hensler) обеспечивали неуклонную поддержку. Наконец, книга не была бы возможна без грамотного руководства Шона Пэрента (Sean Parent) и непрерывного контроля качества кода и текста. Идеи, изложенные в книге, стали итогом нашего тесного, почти тридцатилетнего сотрудничества с Дэйвом Массером (Dave Musser). Бьярне Страуструп (Bjarne Stroustrup) специально доработал язык C++ для поддержки этих идей. И Дэйв и Бьярне были достаточно любезны, чтобы прибыть в Сан-Хосе и тщательно рассмотреть предварительный проект книги. Шон Пэрент и Бьярне Страуструп написали приложение, определяющее подмножество C++, которое используется в книге. Джон Брандт (Jon Brandt) просмотрел несколько черновых вариантов книги. Джон Уилкинсон (John Wilkinson) тщательно прочитал окончательную версию рукописи, предоставив неисчислимое количество ценных предложений.

Книга существенно выиграла благодаря усилиям редактора Питера Гордона (Peter Gordon), редактора проекта Элизабет Райан (Elizabeth Ryan), литературного редактора Эвелин Пайл (Evelyn Pyle) и редакционных рецензентов: Мэтта Аустерна (Matt Austern), Эндрю Кёнига (Andrew Koenig), Дэвида Массера (David Musser), Арчи Робисона (Arch Robison), Джерри Шварца (Jerry Schwarz), Джереми Сика (Jeremy Siek) и Джона Уилкинсона (John Wilkinson).

Мы благодарим всех студентов — участников курса, который проводился в компании Adobe, а также проводившегося ранее курса в компании SGI, за их предложения. Мы надеемся, что нам удалось связать материал этих курсов в единое целое. Мы благодарны за комментарии Дэйву Абрахэмсу (Dave Abrahams), Андрею Александреску (Andrei Alexandrescu), Константину Аркудасу (Konstantine Arkoudas), Джону Бэннингу (John Banning), Гансу Боему (Hans Boehm), Анджело Борсотти (Angelo Borsotti), Джиму Денерту (Jim Dehnert), Джону Детревиллу (John DeTreville), Борису Фомичеву (Boris Fomitchev), Кевлину Хенни (Kevlin Henney), Юсси Кетонену (Jussi Ketonen), Карлу Мальбрену (Karl Malbrain), Мэту Маркусу (Mat Marcus), Ларри Мэзинтеру (Larry Masinter), Дэйву Пэренту (Dave Parent), Дмитрию Полухину (Dmitry Polukhin), Джону Риду (Jon Reid), Марку Рузону (Mark Ruzon), Джеффу Скотту (Geoff Scott), Дэвиду Симонсу (David Simons), Анне Степановой (Anna Stepanov), Тони Ван Ирду (Tony Van Eerd), Уолтеру Ваннини (Walter Vannini), Тиму Винклеру (Tim Winkler) и Олегу Заблуде (Oleg Zabluda). Благодарим Джона Баннинга (John Banning), Боба Инглиша (Bob English), Стивена Граттона (Steven Gratton), Макса Гальперина (Max Hailperin), Евгения Кирпичова (Eugene Kirpichov), Алексея Некрасова (Alexei Nekrassov), Марка Рузона (Mark Ruzon) и Хао Сонга (Hao Song) за то, что выявили ошибки в первой редакции книги. Благодарим Фостера Брereтона (Foster Brereton), Габриэль Дос Реис (Gabriel Dos Reis), Райана Ернста (Ryan Ernst), Абрахама Себастьяна (Abraham Sebastian), Майка Спертуса (Mike Spertus), Хеннинга Тилеманна (Henning Thielemann) и Карлу Вилло-

рию Бургацци (Carla Villoria Burgazzi) за то, что выявили ошибки во второй редакции книги. Благодарим Синдзи Досака (Shinji Dosaka), Райана Ернста (Ryan Ernst) и Стивена Граттона (Steven Gratton) за то, что выявили ошибки во третьей редакции книги.[6]

Наконец, мы благодарны всем людям, которые учили нас посредством своих опубликованных работ или лично, а также университетам и компаниям, которые позволили нам углубить наше понимание программирования.

[6] С последними по времени замечаниями и исправлениями можно ознакомиться на сайте www.elementsofprogramming.com. (См. раздел “Errata”).

Об авторах

Александр Степанов (Alexander Stepanov) изучал математику в Московском государственном университете с 1967 по 1972 гг. Он работает в области программирования с 1972 года: сначала в Советском Союзе, а после эмиграции в 1977 году в Соединенных Штатах. Он занимался программированием операционных систем, инструментов программирования, компиляторов и библиотек. Его работа над началами программирования поддерживалась General Electric, Политехническим институтом Бруклина, AT&T, HP, SGI и Adobe. В 1995 году он получил премию "Excellence in Programming" *Dr. Dobb's Journal* за проект стандартной библиотеки шаблонов C++.

Пол Мак-Джонс (Paul McJones) изучал прикладную математику в Калифорнийском университете, Беркли, с 1967 до 1971 гг. С 1967 года он занимался программированием в областях операционных систем, сред программирования, систем обработки транзакций и приложений для промышленных предприятий и потребительского рынка. Он работал в Калифорнийском университете, IBM, Xerox, Tandem, DEC и в Adobe. В 1982 году он и его соавторы получили премию "ACM Programming Systems and Languages Paper" за статью "Диспетчер по восстановлению системы управления базами данных System R".

Глава 1
Вводные определения

Начиная с краткой таксономии идей, мы введем понятия *значения*, *объекта*, *типа*, *процедуры* и *концепции*, которые представляют различные категории идей в компьютере. Затем определим центральное понятие книги — *регулярность*. Для процедур регулярность означает, что процедуры возвращают равные результаты для равных аргументов. Для типов регулярность означает, что типы позволяют воспользоваться оператором проверки на равенство, а также сохраняющим равенство конструированием копии и присваиванием. Регулярность дает нам возможность заменять значения на другие, но равные им значения, для преобразования и оптимизации программ.

1.1. Категории идей: сущность, вид, род

Чтобы объяснить, в чем состоят объекты, типы и другие фундаментальные понятия программирования, полезно дать краткий обзор некоторых категорий идей, которые соответствуют этим понятиям.

Абстрактная сущность — это предмет, который является вечным и неизменным, тогда как *конкретная сущность* — это предмет, который может обрести свое существование в пространстве и времени или перестать существовать. *Атрибут* определяет соответствие между конкретной сущностью и абстрактной сущностью — он описывает некоторое свойство, измерение или качество конкретной сущности. *Идентичность*, одно из примитивных понятий нашего восприятия действительности, определяет одинаковость предмета, изменяющегося в течение времени. Атрибуты конкретной сущности могут изменяться, не затрагивая его идентичность. *Моментальный снимок* конкретной сущности — это полная коллекция его атрибутов в определенный момент времени. Конкретными сущностями являются не только физические объекты, но также и правовые, финансовые или политические сущности. Синий цвет и число 13 — это примеры абстрактных сущностей. Сократ и Соединенные

Штаты Америки — примеры конкретных сущностей. Цвет глаз Сократа и число американских штатов — это примеры атрибутов.

Абстрактный вид описывает общие свойства эквивалентных абстрактных сущностей. Примеры абстрактных видов — натуральное число и цвет. *Конкретные виды* описывают множество атрибутов почти полностью эквивалентных конкретных сущностей. Примеры конкретных видов — человек и штат США.

Функция — это правило, которое связывает одну или несколько абстрактных сущностей, называемых *аргументами*, из соответствующего вида с абстрактной сущностью, называемой *результатом*, из другого вида. Примерами функций являются функция определения преемника, которая связывает каждое натуральное число с непосредственно следующим за ним, и функция, которая связывает с двумя цветами результат их смешивания.

Абстрактный род описывает различные абстрактные виды, которые являются подобными в некотором отношении. Примеры абстрактных родов — число и бинарный оператор. *Конкретный род* описывает различные конкретные виды, подобные в некотором отношении. Примеры конкретных родов — млекопитающее и двуногое.

Сущность принадлежит к одному и только одному виду, который регламентирует правила для ее построения или существования. Сущность может принадлежать к нескольким родам, каждый из которых описывает определенные свойства.

Ниже мы покажем, что объекты и значения представляют сущности, типы представляют виды, а концепции представляют роды.

1.2. Значения

Если мы не знаем интерпретацию, то единственными предметами, которые мы видим в компьютере, являются нули и единицы. *Элемент данных* представляет собой конечную последовательность нулей и единиц.

Тип значения определяет соответствие между видом (абстрактным или конкретным) и множеством элементов данных. Элемент данных, соответствующий определенной сущности, называется *представлением* сущности; сущность называется *интерпретацией* элемента данных. Мы называем элемент данных вместе с его интерпретацией *значением*. Примерами значений являются целые числа, представленные в формате 32-битовых двоичных чисел в дополнительном коде с обратным порядком байтов, и рациональные числа, представленные как конкатенация двух 32-битовых последовательностей, интерпретируемых как целочисленный числитель и знаменатель, которые пред-

ставлены как значения двоичных чисел в дополнительном коде с обратным порядком байтов.

Элемент данных является *полностью сформированным* относительно типа значения, если и только если этот элемент данных представляет абстрактную сущность. Например, каждая последовательность из 32 битов является полностью сформированной, будучи интерпретируемой как целое число в дополнительном двоичном коде; значение NaN (Not a Number — не число) с плавающей запятой по стандарту IEEE 754 не является полностью сформированным, будучи интерпретируемым как вещественное число.

Тип значений является *собственно частичным*, если его значения представляют собственное подмножество абстрактной сущности в соответствующем виде; в противном случае он является *полным*. Например, тип `int` — собственно частичный, тогда как тип `bool` — полный.

Тип значений является *уникально представленным*, если и только если каждой абстрактной сущности соответствует не больше чем одно значение. Например, тип, представляющий истинностное значение как байт, который интерпретирует нулевое значение как ложь и отличное от нуля значение как истину, не является уникально представленным. Тип, представляющий целое число как знаковый бит и величину без знака, не обеспечивает уникальное представление нуля. Тип, представляющий целое число как двоичное число в дополнительном коде, является уникально представленным.

Тип значений является *неоднозначным*, если и только если значение типа имеет больше чем одну интерпретацию. Отрицание неоднозначного является *однозначным*. Например, неоднозначен тип, представляющий календарный год за период дольше чем одно столетие как две десятичные цифры.

Два значения из типа значений являются *равными*, если и только если они представляют одну и ту же абстрактную сущность. Они являются *представительно равными*, если и только если их элементы данных представляют собой идентичные последовательности нулей и единиц.

Лемма 1.1. Если тип значений является уникально представленным, то из равенства следует представительное равенство.

Лемма 1.2. Если тип значений не является неоднозначным, то из представительного равенства следует равенство.

Если тип значений уникально представлен, мы осуществляем проверку на равенство, проверяя, не являются ли обе последовательности нулей и единиц одинаковыми. В противном случае мы должны осуществлять проверку на равенство таким способом, который сохраняет его согласованность с интерпретациями его аргументов. Неуникальные представления выбираются, если проверка на равенство осуществляется менее часто, чем операции, вырабатывающие новые значения, и если есть возможность ускорить выработку новых

значений за счет замедления проверки на равенство. Например, два рациональных числа, представленные как пары целых чисел, равны, если они приводятся к одному к тому же несократимому виду. Два конечных множества, представленные как неотсортированные последовательности, равны, если после сортировки и устранения дубликатов равны их соответствующие элементы.

Иногда осуществление истинной *поведенческой* проверки на равенство становится слишком дорогим или даже невозможным, как в примере с типом кодировок вычислимых функций. В этих случаях мы вынуждены согласиться на более слабую *представительную* проверку на равенство, которая показывает, что два значения представляют собой одну и ту же последовательность нулей и единиц.

Компьютеры *реализуют* функции на абстрактных сущностях как функции на значениях. Безусловно, значения находятся в памяти, но функция на значениях, реализованная должным образом, не зависит от конкретных адресов памяти: она реализует отображение из значений в значения.

Функция, определенная применительно к данному типу значений, является *регулярной*, если и только если она соблюдает проверку на равенство: подстановка равного значения для аргумента дает равный результат. Большинство числовых функций являются регулярными. Примером числовой функции, не являющейся регулярной, служит функция, которая возвращает числитель рационального числа, представленного как пара целых чисел, поскольку $\frac{1}{2} = \frac{2}{4}$, но $\mathrm{numerator}(\frac{1}{2}) \neq \mathrm{numerator}(\frac{2}{4})$. Регулярные функции позволяют провести *рассуждение, основанное на равенствах*: подстановку равных вместо равных.

Нерегулярная функция зависит от представления, а не только от интерпретации ее аргумента. При проектировании представления для типа значений приходится одновременно выполнять две задачи: осуществление проверки на равенство и принятие решения о том, какие функции должны быть регулярными.

1.3. Объекты

Память — это множество слов, каждое из которых имеет *адрес* и *содержание*. Адрес — это значение фиксированного размера, называемого *длиной адреса*. Содержание — это значение другого фиксированного размера, называемого *длиной слова*. Получение содержания по адресу осуществляется в операции *загрузки*. Изменение связи содержания с адресом осуществляется в операции *сохранения*. Примерами реализации памяти являются байты в оперативной памяти и блоки на диске.

Объект — это представление конкретной сущности как значения в памяти. Объект имеет *состояние*, которое представляет собой значение, относящееся

к некоторому типу значений. Состояние объекта является изменяемым. Если некоторый объект соответствует конкретной сущности, то его состояние соответствует моментальному снимку этой сущности. Объект владеет множеством *ресурсов*, таких как слова в памяти или записи в файле, предназначенных для хранения его состояния.

В то время как значение объекта представляет собой непрерывную последовательность нулей и единиц, ресурсы, в которых хранятся эти нули и единицы, не обязательно расположены непрерывно. В этом состоит интерпретация, которая придает целостность объекту. Например, два значения `double` могут интерпретироваться как одно комплексное число, даже если они не являются смежными. Ресурсы объекта могут даже находиться не в одной памяти. Однако в настоящей книге рассматриваются только объекты, находящиеся в единственной памяти с одним адресным пространством. Каждый объект имеет уникальный *начальный адрес*, из которого могут быть достигнуты все его ресурсы.

Тип объектов — это шаблон для хранения и изменения значений в памяти. Каждому типу объектов соответствует тип значений, описывающий состояния объектов этого типа. Каждый объект принадлежит к некоторому типу объектов. Примером типа объектов могут служить целые числа, представленные в формате 32-битового двоичного числа в дополнительном коде с прямым порядком байтов, выровненные к 4-байтовой границе адреса.

Значения и объекты играют взаимно дополняющие роли. Значения являются неизменными и независимыми от любой конкретной реализации в компьютере. Объекты являются изменяемыми и имеют реализации, зависимые от компьютера. Состояние объекта в любой точке во времени может быть описано значением, которое можно в принципе записать на бумаге (создание моментального снимка) или *сериализовать* (преобразовать в последовательную форму) и отправить по линии связи. Описание состояний объектов в терминах значений позволяет нам абстрагироваться от определенных реализаций объектов, когда речь идет о равенстве. Функциональное программирование имеет дело со значениями, а императивное — с объектами.

Мы используем значения для представления сущностей. Так как значения являются неизменными, они могут представить абстрактную сущность. Последовательности значений могут также представлять последовательности моментальных снимков конкретных сущностей. Объекты хранят значения, представляющие сущности. Так как объекты являются изменяемыми, они могут представлять конкретные сущности, принимая новое значение для представления изменения в сущности. Объекты могут также представить абстрактную сущность, оставаясь постоянными или принимая различные приближения к абстрактному.

Мы используем объекты в компьютере по трем причинам.

1. Объекты моделируют изменяемые конкретные сущности, такие как карточки служащих в приложении учета заработной платы.
2. Объекты обеспечивают удобный способ реализации функции на значениях, таких как процедура, извлекающая квадратный корень числа с плавающей запятой с использованием итеративного алгоритма.
3. Компьютеры с памятью составляют единственную доступную реализацию универсального вычислительного устройства.

Некоторые свойства типов значений переносятся на типы объектов. Объект является *полностью сформированным*, если и только если полностью сформировано его состояние. Тип объектов является *собственно частичным*, если и только если его тип значений собственно частичен; в противном случае он является *полным*. Тип объектов является *уникально представленным*, если и только если уникально представлен его тип значений.

Так как конкретные сущности имеют идентичности, представляющие их объекты нуждаются в соответствующем понятии идентичности. *Маркер идентичности* — это уникальное значение, выражающее идентичность объекта, которое вычисляется из значения объекта и адреса его ресурсов. Примерами маркеров идентичности могут служит адрес объекта, индекс в массиве, в котором хранится объект, и номер служащего в картотеке персонала. Проверка маркеров идентичности на равенство соответствует проверке на идентичность. В течение времени существования приложения каждый конкретный объект может использовать различные маркеры идентичности в ходе перемещения либо в пределах структуры данных, либо из одной структуры данных в другую.

Два объекта одного и того же типа *равны*, если и только если равны их состояния. Если два объекта равны, мы говорим, что каждый из них является *копией* другого. Внесение изменения в объект не затрагивает ни одной его копии.

В настоящей книге используется язык программирования, в котором отсутствует какой-либо способ описания значений и типов значений как отдельных от объектов и типов объектов. Поэтому с данного момента будем предполагать, что если речь идет о типах, без уточнения, то подразумеваются типы объектов.

1.4. Процедуры

Процедура — это последовательность команд, которая изменяет состояние некоторых объектов; процедура может также создавать или уничтожать объекты.

Объекты, с которыми взаимодействует процедура, могут быть разделены на четыре разновидности в соответствии с намерениями программиста.

1. Объекты *входные/выходные* состоят из объектов, передаваемых в процедуру или из процедуры прямо или косвенно с применением ее аргументов или возвращаемого результата.
2. Объекты *локального состояния* состоят из объектов, создаваемых, уничтожаемых и, как правило, изменяемых в течение одного вызова процедуры.
3. Объекты *глобального состояния* состоят из объектов, доступных для этой и других процедур на протяжении многочисленных вызовов.
4. Объекты *собственного состояния* состоят из объектов, доступных только для данной процедуры (и присоединенных к ней процедур), но предоставляемых для общего доступа на протяжении многочисленных вызовов.

Объект передан *прямо*, если он передан как аргумент или возвращен как результат, и передан *косвенно*, если он передан через указатель или объект, подобный указателю. Объект является *входным* объектом процедуры, если происходит его чтение, но не изменение процедурой. Объект является *выходным* объектом процедуры, если происходит его запись, создание или уничтожение процедурой, но не чтение его начального состояния процедурой. Объект является *входным/выходным* объектом процедуры, если происходит не только его изменение, но и чтение процедурой.

Вычислительный базис для типа — это конечное множество процедур, которые обеспечивают построение любой другой необходимой процедуры применительно к данному типу. Базис является *эффективным*, если и только если любая процедура, реализованная с его использованием, оказывается такой же эффективной, как и эквивалентная процедура, написанная на основе альтернативного базиса. Например, базис для k-битовых целых чисел без знака, который предоставляет только значение нуль, процедуру проверки на равенство и функцию определения преемника, не эффективен, поскольку сложность реализации операции сложения на основе определения последующего элемента является экспоненциальной в k.

Базис является *выразительным*, если и только если он позволяет создавать компактные и удобные определения процедур применительно к данному типу. В частности, необходимо обеспечить поддержку всех обычных математических операций, если они являются применимыми. Например, операция вычитания может быть реализована с использованием операций перемены знака и сложения, но должна быть включена в выразительный базис. Аналогично перемена знака может быть реализована с использованием операции вычитания и значения нуль, но должна быть включена в выразительный базис.

1.5. Регулярные типы

Может быть предложено множество процедур, включение которых в вычислительный базис типа позволит нам помещать объекты в структуры данных и использовать алгоритмы для копирования объектов из одной структуры данных в другую. Мы называем типы, имеющие такой базис, *регулярными*, так как их использование гарантирует регулярность поведения и благодаря этому функциональную совместимость типа[1]. Мы выводим семантику регулярных типов от встроенных типов, таких как `bool` и `int`, а также, если ограничиваемся полностью сформированными значениями, `double`. Тип является *регулярным*, если и только если его базис включает процедуру проверки на равенство, процедуру присваивания, деструктор, стандартный конструктор, конструктор копии, процедуру полного упорядочения[2] и основополагающий тип[3].

Равенство — это процедура, которая принимает два объекта одного и того же типа и возвращает значение true, если и только если состояния объектов равны. Неравенство определено тогда же, когда и равенство, и возвращает значение, противоположное равенству. Мы используем следующую систему обозначений:

	Определение	C++
Равенство	$a = b$	`a == b`
Неравенство	$a \neq b$	`a != b`

Присваивание — это процедура, которая принимает два объекта одного и того же типа и делает первый объект равным второму, не изменяя его. Смысл присваивания не зависит от начального значения первого объекта. Мы используем следующую систему обозначений:

	Определение	C++
Присваивание	$a \leftarrow b$	`a = b`

Деструктор — это процедура, вызывающая прекращение существования объекта. После того как по отношению к объекту вызван деструктор, к этому объекту больше не может быть применена ни одна процедура, а его прежние местоположения в памяти и ресурсы могут быть повторно использованы в других целях. Деструктор обычно вызывается неявно. Глобальные объекты уничтожаются после завершения работы приложения, локальные объекты — после выхода из блока, в котором они объявлены, а элементы структуры данных — после уничтожения структуры данных.

[1] Регулярные типы лежат в основе проекта STL, но были впервые формально представлены в [Dehnert and Stepanov 2000].

[2] Строго говоря, как будет показано в главе 4, может быть предусмотрено либо полное упорядочение, либо стандартное полное упорядочение.

[3] Определение понятия основополагающего типа приведено в главе 12.

Конструктор — это процедура, преобразовывающая местоположения в памяти в объект. Возможные варианты поведения находятся в пределах от невыполнения никаких действий до установления состояния сложного объекта.

Объект находится в *частично сформированном* состоянии, если к нему можно применить процедуру присваивания или уничтожения. Для объекта, который частично, но не полностью сформирован, результат любой процедуры, кроме присваивания (только на левой стороне) и уничтожения, не определен.

Лемма 1.3. Полностью сформированный объект является частично сформированным.

Стандартный конструктор не принимает аргументы и оставляет объект в частично сформированном состоянии. Мы используем следующую систему обозначений:

	C++
Локальный объект типа `T`	`T a;`
Анонимный объект типа `T`	`T()`

Конструктор копии принимает дополнительный аргумент того же типа и создает новый, равный ему объект. Мы используем следующую систему обозначений:

	C++
Локальная копия объекта `b`	`T a = b;`

1.6. Регулярные процедуры

Процедура является *регулярной*, если и только если замена ее входных объектов равными объектами приводит к получению равных выходных объектов. Как и применительно к типам значений, определяя тип объекта, мы должны сделать непротиворечивые выборы, касающиеся того, как осуществлять проверку на равенство и какие процедуры, относящиеся к данному типу, должны быть регулярными.

Упражнение 1.1. Распространите понятие регулярности на входные и выходные объекты процедуры, т. е. на объекты, которые не только считываются, но и изменяются.

Безусловно, регулярность применяется по умолчанию, но, как показано ниже, есть причины для применения нерегулярного поведения процедур.

1. Процедура возвращает адрес объекта; примером может служить встроенная функция `addressof`.
2. Процедура возвращает значение, определенное состоянием реального мира, такое как показания часов или другого устройства.

3. Процедура возвращает значение в зависимости от собственного состояния; примером является генератор псевдослучайных чисел.
4. Процедура возвращает зависящий от представления атрибут объекта, допустим, объем памяти, зарезервированной для структуры данных.

Функциональная процедура — это регулярная процедура, определенная на регулярных типах, с одним или несколькими непосредственно передаваемыми входными объектами и единственным выходным объектом, возвращаемым как результат процедуры. Регулярность функциональных процедур обеспечивает возможность применения двух способов передачи входных параметров. Если размер параметра является небольшим или если процедуре требуется копия, в которую она могла бы вносить изменения, передаем параметр *по значению*, выполняя локальную копию. В противном случае параметр передается *по неизменяющейся ссылке*. Функциональная процедура может быть реализована как функция C++, указатель на функцию или функциональный объект[4].

Это — функциональная процедура:

```
int plus_0(int a, int b)
{
    return a + b;
}
```

Это — семантически эквивалентная функциональная процедура:

```
int plus_1(const int& a, const int& b)
{
    return a + b;
}
```

Это — семантически эквивалентная, но не функциональная процедура, поскольку ее входные и выходные объекты не передаются непосредственно:

```
void plus_2(int* a, int* b, int* c)
{
    *c = *a + *b;
}
```

В plus_2 объекты a и b являются входными, тогда как c — выходной объект. Понятие функциональной процедуры — это синтаксическое, а не семантическое свойство: в нашей терминологии процедура plus_2 является регулярной, но не функциональной.

Областью определения функциональной процедуры является подмножество значений ее входных объектов, для применения к которому она предназначена. Функциональная процедура всегда заканчивает свою работу на

[4]Функции C++ не являются объектами и не могут быть переданы как аргументы; указатели на функции C++ и функциональные объекты являются объектами и могут быть переданы как аргументы.

входном объекте из своей области определения; в то время как она может завершиться применительно к входному объекту вне своей области определения, но, возможно, не возвратит значимого значения.

Однородная функциональная процедура — это такая функциональная процедура, все входные объекты которой имеют одинаковый тип. *Домен* однородной функциональной процедуры представляет собой тип ее входных объектов. Вместо того чтобы определять домен неоднородной функциональной процедуры как прямое произведение его входных типов, мы обращаемся к входным типам процедуры по отдельности.

Кодомен для функциональной процедуры представляет собой тип его выходного объекта. *Область значений* для функциональной процедуры представляет собой множество всех значений из ее кодомена, возвращенное процедурой применительно к входным объектам из ее области определения.

Рассмотрим функциональную процедуру

```
int square(int n) { return n * n; }
```

В то время как ее домен и кодомен имеют тип `int`, ее областью определения является множество целых чисел, квадрат которых является представимым с помощью этого типа, а областью значений — множество квадратов целых чисел, представимых с помощью того же типа.

Упражнение 1.2. В предположении, что тип `int` является типом 32-битового двоичного числа в дополнительном коде, сформулируйте его точное определение и установите область значений.

1.7. Концепции

Процедура, в которой используется тип, зависит от синтаксических и семантических свойств вычислительного базиса типа, а также от свойств, определяющих сложность этого базиса. Синтаксически она зависит от присутствия определенных литералов и процедур с конкретными именами и сигнатурами. Семантика зависит от свойств этих процедур. Сложность зависит от времени и пространственной сложности этих процедур. Программа остается правильной, если тип заменяется другим типом с такими же свойствами. Повышению полезности программного компонента, такого как библиотечная процедура или структура данных, способствует его разработка не в терминах конкретных типов, а в терминах требований к типам, выраженных как синтаксические и семантические свойства. Мы называем коллекцию требований *концепцией*. Типы представляют виды, концепции — роды.

Чтобы мы могли описывать концепции, требуется несколько механизмов работы с типами: атрибуты типов, функции и конструкторы типа. *Атрибут*

типа — это отображение из типа в значение, описывающее некоторую характеристику типа. Примерами атрибутов типов могут служить встроенный атрибут типа `sizeof(T)` в языке C++, выравнивание объекта определенного типа и количество элементов в структуре `struct`. Если F — функциональная процедура типа, то Arity(F) возвращает количество ее входных объектов. *Функция типа* — это отображение из типа в другой тип. Примером определения функции типа является следующее: если задан "указатель на T", тип T. В некоторых случаях полезно определить *индексированную* функцию типа с дополнительным постоянным целочисленным параметром; примером может служить функция типа, возвращающая тип i-го элемента типа структуры (с отсчетом от 0). Если F — тип функциональной процедуры, то функция типа Codomain(F) возвращает тип результата. Если F — тип функциональной процедуры и $i < \mathsf{Arity}(F)$, то индексированная функция типа $\mathsf{InputType}(F, i)$ возвращает тип i-го параметра (с отсчетом от 0)[5]. *Конструктор типа* представляет собой механизм создания новых типов исходя из одного или нескольких существующих типов. Например, `pointer(T)` — встроенный конструктор типа, который принимает тип `T` и возвращает тип "указатель на `T`"; `struct` — встроенный n-арный конструктор типа; шаблон структуры — определяемый пользователем n-арный конструктор типа.

Если T представляет собой n-арный конструктор типа, мы обычно обозначаем результат его применения к типам $T_0, \ldots, T_{n-1}$ как $T_{T_0,\ldots,T_{n-1}}$. Важным примером является pair, который, будучи применен к регулярным типам T_0 и T_1, возвращает структуру `struct` типа pair_{T_0,T_1} с элементом $m0$ типа T_0 и элементом $m1$ типа T_1. Для обеспечения того, чтобы сам тип pair_{T_0,T_1} был регулярным, необходимо определить процедуры проверки на равенство и присваивания, деструктор и конструкторы на основе поэлементных расширений соответствующих операций над типами T_0 и T_1. Тот же метод используется для любого типа кортежа, такого как triple. В главе 12 мы покажем реализацию pair_{T_0,T_1} и опишем, как обеспечить регулярность с помощью более сложных конструкторов типа.

Несколько более формально *концепция* представляет собой описание требований к одному или нескольким типам, выраженных в терминах существования и свойств процедур, атрибутов типов и функций типа, определенных на типах. Мы говорим, что концепция *моделируется* конкретными типами или что типы *моделируют* концепцию, если удовлетворяются требования к этим типам. Чтобы указать, что концепция C моделируется типами $T_0, \ldots, T_{n-1}$, пишем $C(T_0, \ldots, T_{n-1})$. Концепция C' *уточняет* концепцию C, если каждый раз, когда C' удовлетворяется для множества типов, C также удовлетворяется для этих типов. Мы говорим, что концепция C *ослабляет* концепцию C', если C' уточняет C.

[5] В приложении B показано, как определять атрибуты и функции типа в языке C++.

Концепция типа представляет собой концепцию, определенную на одном типе. Например, в языке C++ определена концепция типа *целочисленный тип*, уточнениями который являются концепции типа *целочисленный тип без знака* и *целочисленный тип со знаком*, тогда как в STL определена концепция типа *последовательность*. Мы используем примитивные концепции типа *Regular* и *FunctionalProcedure*, соответствующие неформальным определениям, которые мы привели ранее.

Мы определяем концепции формально с использованием стандартной математической системы обозначений. Чтобы определить концепцию C, мы пишем

$$\begin{aligned} &\mathsf{C}(\mathsf{T}_0, \ldots, \mathsf{T}_{n-1}) \triangleq \\ &\quad \mathsf{E}_0 \\ &\quad \wedge\ \mathsf{E}_1 \\ &\quad \wedge\ \ldots \\ &\quad \wedge\ \mathsf{E}_{k-1} \end{aligned}$$

где $\triangleq$ читается как "равно по определению", T_i представляют собой формальные параметры типа, а E_j являются предложениями концепции, которые принимают одну из трех форм.

1. Приложение предварительно определенной концепции, указывающее подмножество параметров типа, которое моделирует концепцию.
2. Сигнатура атрибута типа, функции типа или процедуры, которая должна существовать для любых типов, моделирующих концепцию. Сигнатура процедуры принимает форму $\mathsf{f} : \mathsf{T} \to \mathsf{T}'$, где T — домен; T' — кодомен. Сигнатура функции типа принимает форму $\mathsf{F} : \mathsf{C} \to \mathsf{C}'$, где домен и кодомен — концепции.
3. Аксиома, выраженная в терминах этих атрибутов типа, функций типа и процедур.

Мы иногда включаем определения атрибутов типов, функций типов или процедур вслед за их сигнатурами в предложения концепции второй разновидности. Определение принимает форму $x \mapsto \mathsf{F}(x)$ для некоторого выражения F. В какой-то конкретной модели такое определение могло быть перекрыто другой, но совместимой реализацией.

Например, следующая концепция описывает унарную функциональную процедуру:

$$\begin{aligned} &\mathit{UnaryFunction}(\mathsf{F}) \triangleq \\ &\quad \mathit{FunctionalProcedure}(\mathsf{F}) \\ &\quad \wedge\ \mathsf{Arity}(\mathsf{F}) = 1 \\ &\quad \wedge\ \mathsf{Domain} : \mathit{UnaryFunction} \to \mathit{Regular} \\ &\qquad \mathsf{F} \mapsto \mathsf{InputType}(\mathsf{F}, 0) \end{aligned}$$

Следующая концепция описывает однородную функциональную процедуру:

$$\begin{aligned} &\mathit{HomogeneousFunction}(\mathsf{F}) \triangleq \\ &\quad \mathit{FunctionalProcedure}(\mathsf{F}) \end{aligned}$$

$$\begin{aligned}
&\wedge\ \mathsf{Arity}(\mathsf{F}) > 0\\
&\wedge\ (\forall \mathsf{i},\mathsf{j} \in \mathbb{N})(\mathsf{i},\mathsf{j} < \mathsf{Arity}(\mathsf{F})) \Rightarrow (\mathsf{InputType}(\mathsf{F},\mathsf{i}) = \mathsf{InputType}(\mathsf{F},\mathsf{j}))\\
&\wedge\ \mathsf{Domain} : \mathit{HomogeneousFunction} \to \mathit{Regular}\\
&\qquad \mathsf{F} \mapsto \mathsf{InputType}(\mathsf{F},0)
\end{aligned}$$

Отметим следующее:

$$(\forall \mathsf{F} \in \mathit{FunctionalProcedure})\ \mathit{UnaryFunction}(\mathsf{F}) \Rightarrow \mathit{HomogeneousFunction}(\mathsf{F})$$

Для определения параметров *абстрактной* процедуры применяются типы и значения констант с указанием требований к этим параметрам[6]. Мы используем функциональные шаблоны и шаблоны функционального объекта. Параметры следуют за зарезервированным словом `template` и вводятся с помощью `typename` применительно к типам и с помощью `int` или другого целочисленного типа применительно к значениям констант. Для определения требований применяется предложение `requires`, аргументом которого является выражение, состоящее из значений констант, конкретных типов, формальных параметров, приложения атрибутов типа и функций типа, операций проверки на равенство на значениях и типах, концепций и логических связок[7].

Пример абстрактной процедуры приведен ниже.

```
template<typename Op>
    requires(BinaryOperation(Op))
Domain(Op) square(const Domain(Op)& x, Op op)
{
    return op(x, x);
}
```

Значения домена могли быть большими, поэтому передаем их с помощью неизменяющейся ссылки. Представления операций в памяти, как правило, невелики (в качестве примера можно назвать указатель на функцию или небольшой функциональный объект), поэтому мы передаем их по значению.

Концепции описывают свойства, которым удовлетворяют все объекты определенного типа, тогда как *предварительные условия* описывают свойства конкретных объектов. Например, процедура может требовать, чтобы параметр был простым числом. Требование по применению целочисленного типа задается в соответствии с концепцией, в то время для указания необходимости применять простое число служит предварительное условие. Тип указателя на функцию выражает только ее сигнатуру, а не семантические свойства. Напри-

[6] Абстрактные процедуры были представлены по существу в той форме, в которой мы их используем, в 1930 году в работе [van der Waerden 1930], будучи основанными на лекциях Эмми Нётер (Emmy Noether) и Эмиля Артина (Emil Artin). Джордж Коллинс (George Collins) и Дэвид Массер (David Musser) использовали их в контексте компьютерной алгебры в конце 1960 — начале 1970-х. См., например, [Musser 1975].

[7] Полный синтаксис предложения `requires` см. в приложении B.

мер, процедура может потребовать, чтобы параметр был указателем на функцию, реализующую ассоциативную двухместную операцию на целых числах. Требование по применению двухместной операции на целых числах задается в соответствии с концепцией; ассоциативность конкретной функции определяется согласно предварительному условию.

Чтобы определять предварительное условие для семейства типов, мы должны использовать математические обозначения, такие как кванторы всеобщности и существования, операторы импликации и т. д. Например, чтобы указать, что целое число должно быть простым, определяем

property(N : *Integer*)
prime : N
$n \mapsto (|n| \neq 1) \wedge (\forall u, v \in \mathsf{N})\, uv = n \Rightarrow (|u| = 1 \vee |v| = 1)$

где в первой строке представлены формальные параметры типа и моделируемые ими концепции, во второй строке названо свойство и дана его сигнатура, а в третьей приведен предикат, который устанавливает, соблюдается ли свойство для данного аргумента.

Чтобы определить регулярность унарной функциональной процедуры, пишем следующее:

property(F : *UnaryFunction*)
regular_unary_function : F
$f \mapsto (\forall f' \in \mathsf{F})(\forall x, x' \in \mathsf{Domain}(\mathsf{F}))$
$(f = f' \wedge x = x') \Rightarrow (f(x) = f'(x'))$

Это определение легко распространяется на n-арные функции: применение равных функций к равным аргументам дает равные результаты. Проводя расширение понятия, называем абстрактную функцию регулярной, если все ее реализации являются регулярными. Если не указано иное, в настоящей книге каждый процедурный аргумент представляет собой регулярную функцию; мы опускаем предварительное условие, в котором это утверждается явно.

Проект 1.1. Распространите понятия равенства, присваивания и конструирования копии на объекты различных типов. Обдумайте интерпретации двух типов и аксиом, соединяющих процедуры, в которых применяются разные типы.

1.8. Резюме

Обыденные взгляды на действительность, разделяемые многими людьми, могут найти свое выражение в компьютере. Основывая толкования значений и объектов на их интерпретациях, мы получаем простое, целостное представление. А если учитывается их соответствие сущностям, то становится проще принятие проектных решений, например, касающихся того, как определить равенство.

Глава 2
Преобразования и их орбиты

В этой главе определено преобразование как унарная регулярная функция, принимающая и возвращающая значения одного и того же типа. Последовательные применения преобразования, начинающиеся с некоторого начального значения, определяют орбиту этого значения. Учитывая зависимость только от регулярности преобразования и конечности орбиты, мы реализуем алгоритм определения структуры орбиты, который может найти свое применение в различных областях. Например, он может использоваться для обнаружения цикла в связном списке или анализа генератора псевдослучайных чисел. Мы выводим интерфейс к алгоритму как множество взаимосвязанных процедур и определений для их аргументов и результатов. Этот анализ алгоритма структуры орбиты позволяет нам описать наш подход к программированию в наиболее простом представлении.

2.1. Преобразования

Несомненно, можно предусмотреть применение таких функций, которые принимают значения из любой последовательности типов и возвращают значение любого типа, но обычно встречаются определенные классы сигнатур. В этой книге мы часто используем два таких класса: *однородные предикаты* и *операции*. Однородные предикаты имеют форму $\mathsf{T} \times \cdots \times \mathsf{T} \to \mathsf{bool}$; операции представляют собой функции в форме $\mathsf{T} \times \cdots \times \mathsf{T} \to \mathsf{T}$. Несмотря на наличие $\mathfrak{n}$-арных предикатов и $\mathfrak{n}$-арных операций, мы главным образом сталкиваемся с унарными и бинарными однородными предикатами и унарными и бинарными операциями.

Предикат — это функциональная процедура, возвращающая истинностное значение:

$$\begin{aligned} &\mathit{Predicate}(\mathsf{P}) \triangleq \\ &\qquad \mathit{FunctionalProcedure}(\mathsf{P}) \\ &\quad \wedge\ \mathsf{Codomain}(\mathsf{P}) = \mathsf{bool} \end{aligned}$$

Однородным предикатом называют такой предикат, который является также однородной функцией:

$$\mathit{HomogeneousPredicate}(\mathsf{P}) \triangleq \mathit{Predicate}(\mathsf{P}) \wedge \mathit{HomogeneousFunction}(\mathsf{P})$$

Унарный предикат — это предикат, принимающий один параметр:

$$\mathit{UnaryPredicate}(\mathsf{P}) \triangleq \mathit{Predicate}(\mathsf{P}) \wedge \mathit{UnaryFunction}(\mathsf{P})$$

Операция — это однородная функция, кодомен которой равен ее домену:

$$\mathit{Operation}(\mathsf{Op}) \triangleq \mathit{HomogeneousFunction}(\mathsf{Op}) \wedge \mathsf{Codomain}(\mathsf{Op}) = \mathsf{Domain}(\mathsf{Op})$$

Примеры операций:

```
int abs(int x) {
    if (x < 0) return -x; else return x;
} // унарная операция

double euclidean_norm(double x, double y) {
    return sqrt(x * x + y * y);
} // бинарная операция

double euclidean_norm(double x, double y, double z) {
    return sqrt(x * x + y * y + z * z);
} // тернарная операция
```

Лемма 2.1. $\mathsf{euclidean\_norm}(\mathrm{x},\mathrm{y},\mathrm{z}) = \mathsf{euclidean\_norm}(\mathsf{euclidean\_norm}(\mathrm{x},\mathrm{y}),\mathrm{z})$

Эта лемма показывает, что тернарная версия может быть получена из бинарной версии. По соображениям эффективности, выразительности и, возможно, точности, тернарная версия становится частью вычислительного базиса для программ, имеющих дело с трехмерным пространством.

Процедура является *частичной*, если имеет область определения, представляющую собой подмножество прямого произведения типов ее входных параметров; процедура является *полной*, если имеет область определения, равную прямому произведению. Мы придерживаемся стандартного математического подхода, согласно которому частичная функция включает полную функцию. Мы называем *неполными* частичные процедуры, которые не являются полными. Реализации некоторых полных функций на компьютере неполны из-за конечности представления. Например, сложение, которое определено на 32-битовых целых числах со знаком, неполно.

Неполная процедура сопровождается предусловием, задающим область ее определения. Чтобы проверить правильность вызова этой процедуры, мы должны выяснить, удовлетворяют ли аргументы ее предусловию. Иногда частичную

процедуру передают в качестве параметра к алгоритму, который должен выявить во время выполнения область определения процедурного параметра. Чтобы иметь дело с такими случаями, мы задаем *предикат области определения* с теми же входными параметрами, что и в процедуре; предикат возвращает true, если и только если входные параметры находятся в пределах области определения процедуры. Перед вызовом неполной процедуры должно быть либо удовлетворено ее предусловие, либо вызов должен быть защищен вызовом ее предиката области определения.

Упражнение 2.1. Реализуйте предикат области определения для сложения на 32-битовых целых числах со знаком.

Эта глава имеет дело с унарными операциями, которые мы называем *преобразованиями:*

$$\begin{aligned} &\mathit{Transformation}(\mathsf{F}) \triangleq \\ &\qquad \mathit{Operation}(\mathsf{F}) \\ &\quad \wedge\ \mathit{UnaryFunction}(\mathsf{F}) \\ &\quad \wedge\ \mathsf{DistanceType} : \mathit{Transformation} \to \mathit{Integer} \end{aligned}$$

Мы обсуждаем DistanceType в следующем разделе.

Преобразования являются самокомпонующимися: $f(x), f(f(x)), f(f(f(x)))$ и т. д. Область определения $f(f(x))$ представляет собой пересечение области определения и области значений f. Эта способность к самокомпоновке, наряду со способностью проводить проверку на равенство, позволяет нам определять интересные алгоритмы.

Если f — преобразование, определяем его степени следующим образом:

$$f^n(x) = \begin{cases} x & \text{если } n = 0, \\ f^{n-1}(f(x)) & \text{если } n > 0 \end{cases}$$

Чтобы реализовать алгоритм для вычисления $f^n(x)$, мы должны определить требование для целочисленного типа. Мы изучаем различные концепции, описывающие целые числа, в главе 5. Пока мы полагаемся на интуитивное понимание целых чисел. Их модели включают целочисленные типы без знака и со знаком, а также целые числа произвольной точности со следующими операциями и литералами:

	Спецификации	C++
Сумма	$+$	`+`
Разность	$-$	`-`
Произведение	$\cdot$	`*`
Частное	$/$	`/`
Остаток	mod	`%`
Нуль	0	`I(0)`
Один	1	`I(1)`
Два	2	`I(2)`

где I — целочисленный тип.

Это приводит к следующему алгоритму:

```
template<typename F, typename N>
    requires(Transformation(F) && Integer(N))
Domain(F) power_unary(Domain(F) x, N n, F f)
{
    // Предусловие: n ⩾ 0 ∧ (∀i ∈ N) 0 < i ⩽ n ⇒ f^i(x) определено
    while (n != N(0)) {
        n = n - N(1);
        x = f(x);
    }
    return x;
}
```

2.2. Орбиты

Чтобы понять глобальное поведение преобразования, исследуем структуру его *орбит* — элементов, достижимых от начального элемента путем повторных применений преобразования. y является *достижимым* от x согласно преобразованию f, если для некоторого $n \geqslant 0$, $y = f^n(x)$. x является *циклическим* согласно f, если для некоторого $n \geqslant 1$, $x = f^n(x)$. x является *терминальным* согласно f, если и только если x не находится в области определения f. *Орбита* x согласно преобразованию f представляет собой множество всех элементов, достижимых от x согласно f.

Лемма 2.2. Орбита не содержит одновременно циклический и терминальный элементы.

Лемма 2.3. Орбита содержит не более одного терминального элемента.

Если y является достижимым от x согласно f, то *расстояние* от x до y — это наименьшее количество шагов преобразования от x до y. Очевидно, что расстояние не всегда определено.

Если задан тип преобразования F, то DistanceType(F) — это целочисленный тип, достаточно большой для кодирования максимального количества шагов при любом преобразовании $f \in F$ от одного элемента $T = \mathsf{Domain}(F)$ к другому. Если тип T занимает k битов, то количество значений может составлять 2^k, но может быть только $2^k - 1$ шагов между различными значениями. Таким образом, если T — тип фиксированного размера, то любой целочисленный тип того же размера является допустимым типом расстояния для любого преобразования на T. (Вместо того, чтобы использовать тип расстояния, мы допускаем использование любого целочисленного типа в power_unary, поскольку дополнительная общность, по-видимому, здесь не помешает.) Часто имеет место

такая ситуация, что все типы преобразования над доменом имеют один и тот же тип расстояния. В этом случае для типа домена определена функция типа DistanceType, которая определяет соответствующую функцию типа для типов преобразования.

Существование DistanceType приводит к следующей процедуре:

```
template<typename F>
    requires(Transformation(F))
DistanceType(F) distance(Domain(F) x, Domain(F) y, F f)
{
    // Предусловие: y достижим от x согласно f
    typedef DistanceType(F) N;
    N n(0);
    while (x != y) {
        x = f(x);
        n = n + N(1);
    }
    return n;
}
```

Орбиты имеют различные формы. Орбита x согласно преобразованию называется

бесконечной, если она не содержит циклического или терминального элемента;

завершающейся, если в ней имеется терминальный элемент;

циклической, если x является циклическим;

ρ-*образной*, если x не является циклическим, но его орбита содержит циклический элемент.

Орбита x называется *конечной*, если она не бесконечна. Различные случаи иллюстрируются на рис. 2.1.

Циклом орбиты называется множество циклических элементов в орбите, которое является пустым для бесконечных и завершающихся орбит. *Ручка орбиты*, представляющая собой дополнение цикла орбиты относительно орбиты, является пустой для круговой орбиты. *Точкой соединения* называется первый циклический элемент, который является первым элементом круговой орбиты и первым элементом после ручки для ρ-образной орбиты. *Размером орбиты* o некоторой орбиты называется количество в ней различных элементов. *Размером ручки* h некоторой орбиты называется количество элементов в ручке орбиты. *Размером цикла* c некоторой орбиты называется количество элементов в цикле орбиты.

Лемма 2.4. $o = h + c$

Лемма 2.5. Расстояние от любой точки в орбите до некоторой точки в цикле этой орбиты всегда определено.

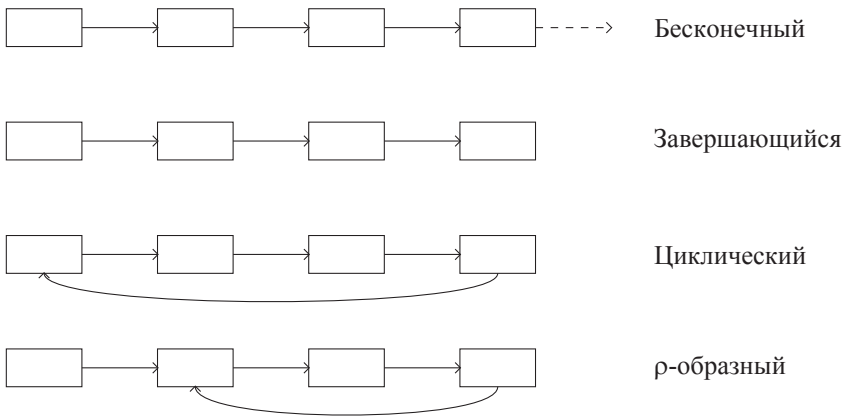


Рис. 2.1. Формы орбит

Лемма 2.6. Если x и y — различные точки в цикле размера c, то

$$c = \mathsf{distance}(x, y, f) + \mathsf{distance}(y, x, f)$$

Лемма 2.7. Если x и y — точки в цикле с размером c, то расстояние от x до y удовлетворяет условию

$$0 \leqslant \mathsf{distance}(x, y, f) < c$$

2.3. Точка столкновения

Если мы наблюдаем поведение преобразования, не имея доступа к его определению, то не можем определить, бесконечна ли конкретная орбита: она может закончиться или снова войти в цикл в любой точке. Если мы знаем, что орбита конечна, то можем использовать какой-то алгоритм, чтобы определить форму орбиты. Поэтому для всех алгоритмов в этой главе неявно применяется предусловие конечности орбиты.

Разумеется, есть такой примитивный алгоритм, который предусматривает сохранение в памяти каждого посещенного элемента и проверку при каждом шаге, не встречался ли ранее вновь полученный элемент. Даже если бы мы могли использовать хеширование для ускорения поиска, такой алгоритм все равно требовал бы линейно увеличивающегося объема памяти и не был бы практичным во многих приложениях. Однако есть и такой алгоритм, который требует только постоянного объема памяти.

Понять суть этого алгоритма поможет следующая аналогия. Если быстрый и медленный объекты одновременно начинают двигаться из одной точки по некоторому пути, то быстрый объект догонит медленный, если и только если

в пути есть цикл. Если цикл отсутствует, то быстрый объект достигнет конца пути прежде, чем медленный. Если же цикл имеется, то к тому времени, как медленный объект войдет в цикл, быстрый уже будет в цикле и в конечном счете встретится с медленным. Перенося это интуитивное представление из непрерывного домена в дискретный, необходимо следить за тем, чтобы быстрый объект не мог проскакивать мимо медленного[1].

Дискретная версия алгоритма основана на поиске точки, в которой быстрый объект встречается с медленным. *Точка столкновения* преобразования f и начальной точки x — это уникальное значение y, такое, что

$$y = f^n(x) = f^{2n+1}(x)$$

и $n \geqslant 0$ — наименьшее целое число, удовлетворяющее этому условию. Это определение приводит к такому алгоритму определения структуры орбиты, который нуждается в одном сравнении быстрого и медленного объектов в расчете на каждую итерацию. Чтобы можно было справиться с частичными преобразованиями, передаем в алгоритм предикат области определения:

```
template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
        Domain(F) == Domain(P))
Domain(F) collision_point(const Domain(F)& x, F f, P p)
{
    // Предусловие: p(x) ⇔ f(x) определено
    if (!p(x)) return x;
    Domain(F) slow = x;          // slow = f^0(x)
    Domain(F) fast = f(x);       // fast = f^1(x)
                                 // n ← 0 (законченные итерации)
    while (fast != slow) {       // slow = f^n(x) ∧ fast = f^(2n+1)(x)
        slow = f(slow);          // slow = f^(n+1)(x) ∧ fast = f^(2n+1)(x)
        if (!p(fast)) return fast;
        fast = f(fast);          // slow = f^(n+1)(x) ∧ fast = f^(2n+2)(x)
        if (!p(fast)) return fast;
        fast = f(fast);          // slow = f^(n+1)(x) ∧ fast = f^(2n+3)(x)
                                 // n ← n + 1
    }
    return fast;                 // slow = f^n(x) ∧ fast = f^(2n+1)(x)
    // Постусловие: возвращаемое значение представляет собой конечную точку
    // или точку столкновения
}
```

Устанавливаем правильность collision_point в три этапа: 1) проводим проверку того, что f никогда не применяется к аргументу вне области опреде-

[1] В [Knuth 1997, стр. 7] предположено, что этот алгоритм принадлежит Роберту В. Флойду.

ления; 2) проверяем, удовлетворено ли постусловие после завершения работы алгоритма; 3) подтверждаем, что работа алгоритма всегда заканчивается.

Несмотря на то, что f — частичная функция, ее использование в процедуре полностью определено, поскольку движение $fast$ защищено вызовом p. Движение $slow$ незащищено, поскольку в силу регулярности f объект $slow$ проходит по той же орбите, что и $fast$, поэтому f всегда определена, будучи применена к $slow$.

Из этого следует, что если после $n \geqslant 0$ итераций положение $fast$ становится равным $slow$, то имеют место $fast = f^{2n+1}(x)$ и $slow = f^n(x)$. Кроме того, n является именно таким наименьшим целым числом, поскольку мы проверяли условие для каждого $i < n$.

Если цикл отсутствует, то p в конечном счете возвратит false ввиду конечности орбиты. Если цикл имеется, то $slow$ в конечном счете достигнет точки соединения (первый элемент в цикле). Рассмотрим расстояние d от $fast$ до $slow$ в верхней части петли, когда $slow$ впервые входит в цикл: $0 \leqslant d < c$. Если $d = 0$, процедура заканчивается. В противном случае расстояние от $fast$ до $slow$ уменьшается на 1 при каждой итерации. Поэтому процедура всегда заканчивается; после ее завершения $slow$ перемещается в общей сложности на $h + d$ шагов.

Следующая процедура определяет, завершается ли орбита:

```
template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
        Domain(F) == Domain(P))
bool terminating(const Domain(F)& x, F f, P p)
{
    // Предусловие: p(x) ⇔ f(x) определено
    return !p(collision_point(x, f, p));
}
```

Иногда нам известно то, что преобразование является полным или орбита бесконечна применительно к определенному начальному элементу. Для этих ситуаций полезно иметь специализированную версию collision_point:

```
template<typename F>
    requires(Transformation(F))
Domain(F)
collision_point_nonterminating_orbit(const Domain(F)& x, F f)
{
    Domain(F) slow = x;          // slow = f^0(x)
    Domain(F) fast = f(x);       // fast = f^1(x)
                                 // n ← 0 (законченные итерации)
    while (fast != slow) {       // slow = f^n(x) ∧ fast = f^(2n+1)(x)
        slow = f(slow);          // slow = f^(n+1)(x) ∧ fast = f^(2n+1)(x)
        fast = f(fast);          // slow = f^(n+1)(x) ∧ fast = f^(2n+2)(x)
```

```
        fast = f(fast);          // slow = f^{n+1}(x) ∧ fast = f^{2n+3}(x)
                                 // n ← n + 1
    }
    return fast;                 // slow = f^n(x) ∧ fast = f^{2n+1}(x)
    // Постусловие: возвращаемое значение представляет собой
    // точку столкновения
}
```

Чтобы определить структуру цикла (размер ручки, точку соединения и размер цикла), мы должны проанализировать позицию точки столкновения.

Когда процедура возвращает точку столкновения

$$f^n(x) = f^{2n+1}(x)$$

n — количество шагов, проделанных $slow$, а $2n + 1$ — количество шагов, проделанных $fast$.

$$n = h + d$$

где h — размер ручки, а $0 \leqslant d < c$ — количество шагов, проделанных $slow$ в цикле. Количество шагов, проделанных $fast$, составляет

$$2n + 1 = h + d + qc$$

где $q > 0$ — количество полных циклов, законченных объектом $fast$ к тому времени, как он сталкивается со $slow$. В силу того, что $n = h+d$, имеет место следующее:

$$2(h + d) + 1 = h + d + qc$$

С помощью упрощения получаем

$$qc = h + d + 1$$

Представим h по модулю c:

$$h = mc + r$$

с условием $0 \leqslant r < c$. Подстановка дает

$$qc = mc + r + d + 1$$

или

$$d = (q - m)c - r - 1$$

из $0 \leqslant d < c$ следует

$$q - m = 1$$

поэтому

$$d = c - r - 1$$

и необходимы $r + 1$ шагов, чтобы закончить цикл.

Поэтому расстояние от точки столкновения до точки соединения составляет

$$e = r + 1$$

В случае круговой орбиты $h = 0$ имеет место $r = 0$, и расстояние от точки столкновения до начала орбиты равно

$$e = 1$$

Поэтому цикличность может быть проверена с помощью следующих процедур:

```
template<typename F>
    requires(Transformation(F))
bool circular_nonterminating_orbit(const Domain(F)& x, F f)
{
    return x == f(collision_point_nonterminating_orbit(x, f));
}

template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
        Domain(F) == Domain(P))
bool circular(const Domain(F)& x, F f, P p)
{
    // Предусловие: p(x) ⇔ f(x) определено
    Domain(F) y = collision_point(x, f, p);
    return p(y) && x == f(y);
}
```

Но мы все еще не знаем размер ручки h и размер цикла c. Определение последнего значения является несложным, поскольку точка столкновения известна: пройдем по циклу и подсчитаем количество шагов.

Чтобы понять, как определить h, рассмотрим позицию точки столкновения:

$$f^{h+d}(x) = f^{h+c-r-1}(x) = f^{mc+r+c-r-1}(x) = f^{(m+1)c-1}(x)$$

Прохождение $h + 1$ шагов от точки столкновения приводит нас к точке $f^{(m+1)c+h}(x)$, которая равна $f^h(x)$, поскольку $(m + 1)c$ соответствует движению по циклу $m + 1$ раз. Если мы одновременно проделаем h шагов от x и $h + 1$ шагов от точки столкновения, то окажемся в точке соединения. Другими словами, орбиты x и 1 шаг после точки столкновения сходятся точно в h шагах, что приводит к следующей последовательности алгоритмов:

```
template<typename F>
    requires(Transformation(F))
Domain(F) convergent_point(Domain(F) x0, Domain(F) x1, F f)
```

```
{
// Предусловие: (∃n ∈ DistanceType(F)) n ⩾ 0 ∧ f^n(x0) = f^n(x1)
    while (x0 != x1) {

        x0 = f(x0);
        x1 = f(x1);
    }
    return x0;
}


template<typename F>
    requires(Transformation(F))
Domain(F)
connection_point_nonterminating_orbit(const Domain(F)& x, F f)
{
    return convergent_point(
        x,
        f(collision_point_nonterminating_orbit(x, f)),
        f);
}


template<typename F, typename P>
    requires(Transformation(F) && UnaryPredicate(P) &&
        Domain(F) == Domain(P))
Domain(F) connection_point(const Domain(F)& x, F f, P p)
{
    // Предусловие: p(x) ⇔ f(x) определено
    Domain(F) y = collision_point(x, f, p);
    if (!p(y)) return y;
    return convergent_point(x, f(y), f);
}
```

Лемма 2.8. Если орбиты двух элементов пересекаются, они имеют одни и те же циклические элементы.

Упражнение 2.2. Даны преобразование и его предикат области определения. Спроектируйте алгоритм, который определяет, пересекаются ли орбиты двух элементов.

Упражнение 2.3. Предусловие convergent_point гарантирует завершение. Реализуйте алгоритм convergent_point_guarded, предназначенный для использования, когда неизвестно, соблюдается ли это предусловие, но орбиты x0 and x1 включают общий элемент.

2.4. Измерение размеров орбиты

Типом, который естественным образом подходит для представления размеров o, h и c орбит типа T, может служить целочисленный счетный тип, достаточно большой, чтобы с его помощью можно было подсчитать все различные значения типа T. Если тип T занимает k битов, то количество значений может составлять 2^k, поэтому счетный тип, занимающий k битов, не позволяет представить все результаты подсчета от 0 до 2^k. Однако есть возможность представить эти размеры с использованием типа расстояния.

Орбита потенциально способна содержать все значения типа, и в этом случае o может не поместиться во внутреннем представлении типа расстояния. В зависимости от формы такой орбиты могут также не поместиться h и c. Но если орбита ρ-образная, помещаются и h, и c. Во всех случаях помещается каждое из следующих: $o - 1$ (максимальное расстояние в орбите), $h - 1$ (максимальное расстояние в ручке) и $c - 1$ (максимальное расстояние в цикле). Это позволяет нам реализовать процедуры, возвращающие тройку, которая представляет полную структуру орбиты, а элементами тройки является следующее:

Случай	$m0$	$m1$	$m2$
Завершающаяся	$h - 1$	0	терминальный элемент
Круговая	0	$c - 1$	x
ρ-образная	h	$c - 1$	точка соединения

```
template<typename F>
    requires(Transformation(F))
triple<DistanceType(F), DistanceType(F), Domain(F)>
orbit_structure_nonterminating_orbit(const Domain(F)& x, F f)
{
    typedef DistanceType(F) N;
    Domain(F) y = connection_point_nonterminating_orbit(x, f);
    return triple<N, N, Domain(F)>(distance(x, y, f),
                                   distance(f(y), y, f),
                                   y);
}

template<typename F, typename P>
    requires(Transformation(F) &&
        UnaryPredicate(P) && Domain(F) == Domain(P))
triple<DistanceType(F), DistanceType(F), Domain(F)>
orbit_structure(const Domain(F)& x, F f, P p)
{
    // Предусловие: p(x) ⇔ f(x) определено
    typedef DistanceType(F) N;
```

```
    Domain(F) y = connection_point(x, f, p);
    N m = distance(x, y, f);
    N n(0);
    if (p(y)) n = distance(f(y), y, f);
    // Завершающаяся: m = h − 1 ∧ n = 0
    // Другая: m = h ∧ n = c − 1
    return triple<N, N, Domain(F)>(m, n, y);
}
```

Упражнение 2.4. Выведите формулы для подсчета различных операций (f, p, проверка на равенство) для алгоритмов в этой главе.

Упражнение 2.5. Используйте orbit_structure_nonterminating_orbit, чтобы определить средний размер ручки и размер цикла какого-нибудь конкретного генератора случайных чисел для различных начальных значений.

2.5. Действия

В алгоритмах часто используется преобразование f в виде инструкции, подобной следующей:

```
x = f(x);
```

Изменение состояния объекта путем применения к нему преобразования определяет *действие* над объектом. Преобразования и соответствующие действия являются дуалистичными по отношению друг к другу: действие определимо в терминах преобразования, и наоборот:

```
void a(T& x) { x = f(x); }    // действие из преобразования
```

и

```
T f(T x) { a(x); return x; } // преобразование из действия
```

Несмотря на эту дуалистичность, иногда более эффективны независимые реализации, когда необходимо обеспечить и действие, и преобразование. Например, если преобразование определено на большом объекте и изменяет только часть его полного состояния, то действие может оказаться значительно быстрее.

Упражнение 2.6. Перезапишите все алгоритмы в этой главе в терминах действий.

Проект 2.1. Еще один способ обнаружения цикла состоит в том, чтобы неоднократно проверять единственный продвигающийся элемент на равенство с сохраненным элементом, заменяя сохраненный элемент через постоянно увеличивающиеся интервалы. Эта и другие идеи описаны в [Sedgewick, et al. 1982],

[Brent 1980] и [Levy 1982]. Реализуйте другие алгоритмы для анализа орбиты, сравните их производительность применительно к различным приложениям и разработайте набор рекомендаций для выбора наиболее подходящего алгоритма.

2.6. Резюме

Абстракция позволила нам определить абстрактные процедуры, которые могут использоваться в различных областях. Регулярность типов и функций является необходимой для обеспечения работы алгоритмов: `fast` и `slow` следуют по одной и той же орбите благодаря регулярности. Необходимым требованием является разработка терминологии (например, касающейся разновидностей и размеров орбит). Должны быть точно определены типы, относящиеся к этой проблематике, такие как типы расстояния.

Глава 3
Ассоциативные операции

В этой главе обсуждаются ассоциативные бинарные операции. Ассоциативность позволяет перегруппировывать смежные операции. Эта возможность перегруппирования приводит к эффективному алгоритму вычисления степеней бинарной операции. Регулярность обеспечивает применение различных преобразований программы в целях оптимизации алгоритма. Затем мы используем алгоритм для вычисления линейных рекуррентных соотношений, таких как числа Фибоначчи, за логарифмическое время.

3.1. Ассоциативность

Бинарная операция представляет собой операцию с двумя аргументами:

$$\begin{array}{l} \mathit{BinaryOperation}(\mathrm{Op}) \triangleq \\ \qquad \mathit{Operation}(\mathrm{Op}) \\ \quad \wedge\ \mathsf{Arity}(\mathrm{Op}) = 2 \end{array}$$

Бинарные операции сложения и умножения занимают центральное место в математике. Широко используются также многие другие операции, такие как вычисление минимума и максимума, конъюнкция, дизъюнкция, объединение множеств, пересечение множеств и т. д. Все эти операции являются *ассоциативными*:

$$\begin{array}{l} \textbf{property}(\mathrm{Op} : \mathit{BinaryOperation}) \\ \mathsf{associative} : \mathrm{Op} \\ \quad \mathrm{op} \mapsto (\forall a, b, c \in \mathsf{Domain}(\mathrm{op}))\, \mathrm{op}(\mathrm{op}(a, b), c) = \mathrm{op}(a, \mathrm{op}(b, c)) \end{array}$$

Разумеется, есть и неассоциативные бинарные операции, такие как вычитание и деление.

Если из контекста ясно, что определенная бинарная операция op является ассоциативной, мы часто используем подразумеваемую мультипликативную систему обозначений и пишем ab вместо $\mathrm{op}(a, b)$. Поскольку операция op является ассоциативной, мы не обязаны вводить круглые скобки в выражение,

предусматривающее два или большее количество ее применений, учитывая то, что способы группирования операций эквивалентны: $(\cdots(a_0a_1)\cdots)a_{n-1} = \cdots = a_0(\cdots(a_{n-2}a_{n-1})\cdots) = a_0a_1\cdots a_{n-1}$. Если $a_0 = a_1 = \cdots = a_{n-1} = a$, пишем a^n: n–я *степень* от a.

Лемма 3.1. $a^n a^m = a^m a^n = a^{n+m}$ (степени одного и того же элемента коммутативны)

Лемма 3.2. $(a^n)^m = a^{nm}$

Но соотношение $(ab)^n = a^n b^n$ не всегда является истинным. Это условие соблюдается, только если операция коммутативна.

Если f и g представляют собой преобразования на одном и том же домене, то их *композицией*, $g \circ f$, называется преобразование, отображающее x в $g(f(x))$.

Лемма 3.3. Бинарная операция композиции является ассоциативной.

Если мы выберем некоторый элемент a домена ассоциативной операции op и будем рассматривать выражение $op(a, x)$ как унарную операцию с формальным параметром x, то можем трактовать a как преобразование "умножением на a". Это оправдывает использование одной и той же системы обозначений для степеней преобразования, f^n, и степеней элемента при ассоциативной бинарной операции, a^n. Эта дуалистичность позволяет нам использовать алгоритм из предыдущей главы для доказательства интересной теоремы о степенях ассоциативной операции. Элемент x имеет *конечный порядок* при ассоциативной операции, если существуют целые числа $0 < n < m$, такие, что $x^n = x^m$. Элемент x называется *идемпотентным элементом* при ассоциативной операции, если имеет место $x = x^2$.

Теорема 3.1. Элемент конечного порядка имеет идемпотентную степень (см. [Frobenius 1895]).

Доказательство. Предположим, что x — элемент конечного порядка при ассоциативной операции op. Пусть $g(z) = op(x, z)$. Поскольку x представляет собой элемент конечного порядка, его орбита при g имеет цикл. В соответствии с постусловием преобразования,

$$\mathsf{collision\_point}(x, g) = g^n(x) = g^{2n+1}(x)$$

для некоторого $n \geqslant 0$. Таким образом,

$$g^n(x) = x^{n+1}$$
$$g^{2n+1}(x) = x^{2n+2} = x^{2(n+1)} = (x^{n+1})^2$$

и x^{n+1} — идемпотентная степень x. □

Лемма 3.4. collision_point_nonterminating_orbit может использоваться в доказательстве.

3.2. Вычисление степеней

Алгоритм вычисления a^n для ассоциативной операции op принимает в качестве параметров a, n и op. Тип a представляет собой домен op; значение n должно иметь целочисленный тип. Без предположения об ассоциативности следующие два алгоритма вычисляют степень соответственно слева направо и справа налево:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_left_associated(Domain(Op) a, I n, Op op)
{
    // Предусловие: n > 0
    if (n == I(1)) return a;
    return op(power_left_associated(a, n - I(1), op), a);
}


template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_right_associated(Domain(Op) a, I n, Op op)
{
    // Предусловие: n > 0
    if (n == I(1)) return a;
    return op(a, power_right_associated(a, n - I(1), op));
}
```

Алгоритмы выполняют $n-1$ операций. Они возвращают другие результаты для неассоциативной операции. Рассмотрим, например, возведение 1 в 3-ю степень с операцией вычитания.

Если и a, и n — целые числа, а операция представляет собой умножение, оба алгоритма выполняют возведение в степень; если же операцией является сложение, оба они выполняют умножение. Еще в Древнем Египте был открыт быстрый алгоритм умножения, который может быть обобщен на вычисление степеней любой ассоциативной операции[1].

[1]Оригинал приведен в [Robins and Shute 1987, стр. 16-17]; папирус датируется примерно 1650 годом до н. э., но его составитель указал, что это — копия другого папируса с датой около 1850 года до н. э.

Ассоциативность позволяет нам свободно перегруппировывать операции, поэтому имеем

$$a^n = \begin{cases} a & \text{if } n = 1 \\ (a^2)^{n/2} & \text{если } n \text{ четно} \\ (a^2)^{\lfloor n/2 \rfloor} a & \text{если } n \text{ нечетно} \end{cases}$$

что соответствует следующему:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_0(Domain(Op) a, I n, Op op)
{
    // Предусловие: associative(op) ∧ n > 0
    if (n == I(1)) return a;
    if (n % I(2) == I(0))
        return power_0(op(a, a), n / I(2), op);
    return op(power_0(op(a, a), n / I(2), op), a);
}
```

Подсчитаем количество операций, выполяемых функцией power_0 для показателя степени n. Количество рекурсивных вызовов равно $\lfloor \log_2 n \rfloor$. Предположим, что ν представляет собой количество единиц в двоичном представлении n. Каждый рекурсивный вызов выполняет операцию по возведению a в квадрат. Кроме того, в $\nu - 1$ вызовах выполняется лишняя операция. Таким образом, количество операций составляет

$$\lfloor \log_2 n \rfloor + (\nu - 1) \leqslant 2 \lfloor \log_2 n \rfloor$$

Если $n = 15$, то $\lfloor \log_2 n \rfloor = 3$ и количество единиц составляет четыре, поэтому формула дает в результате шесть операций. Другое группирование приводит к получению $a^{15} = (a^3)^5$, где для a^3 требуются две операции, а для a^5 — три операции, что в общей сложности равно пяти. Имеются также более быстрые группирования для других показателей степени, таких как 23, 27, 39 и 43[2].

Реализация алгоритма power_left_associated выполняет $n - 1$ операций, а power_0 — самое большее $2\lfloor \log_2 n \rfloor$ операций, поэтому может показаться, что при очень больших значениях n алгоритм power_0 всегда будет намного быстрее. Но иногда дело обстоит иначе. Например, если операция представляет собой умножение многочленов с одной переменной с целочисленными коэффициентами произвольной точности, быстрее становится power_left_associated[3]. Даже применительно к этому простому алгоритму мы не знаем, как точно задать требования к сложности, которые определяют, какая из двух реализаций лучше.

[2]Исчерпывающее описание возведения в степень с применением минимального количества операций см. в [Knuth 1997, стр. 465–481].

[3]См. [McCarthy 1986].

Способность power_0 обрабатывать очень большие показатели степени, скажем, 10^{300}, выдвигает ее в число крайне важных для шифрования[4].

3.3. Преобразования программ

Функция power_0 является удовлетворительной реализацией алгоритма и применима, если стоимость выполнения операции значительно больше по сравнению с издержками вызовов функции, вызванных рекурсией. В этом разделе мы выводим итеративный алгоритм, который выполняет то же количество операций, что и power_0, используя последовательность преобразований программы, которые могут использоваться во многих контекстах[5]. В остальной части книги мы показываем только окончательные или почти окончательные версии.

power_0 содержит два идентичных рекурсивных вызова. Но в каждом конкретном вызове выполняется только один из них, поэтому можно сократить размер кода путем *устранения общего подвыражения*:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_1(Domain(Op) a, I n, Op op)
{
    // Предусловие: associative(op) ∧ n > 0
    if (n == I(1)) return a;
    Domain(Op) r = power_1(op(a, a), n / I(2), op);
    if (n % I(2) != I(0)) r = op(r, a);
    return r;
}
```

Наша цель состоит в том, чтобы устранить рекурсивный вызов. Первый шаг состоит в преобразовании процедуры в *форму с концевой рекурсией*, в которой выполнение процедуры заканчивается рекурсивным вызовом. Одним из методов, который допускает это преобразование, является *введение накопительной переменной*, для того, чтобы сохранение накопленного результата от одного рекурсивного вызова к другому происходило с помощью накопительной переменной:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_0(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
```

[4] См. работу по RSA в [Rivest, et al. 1978].

[5] Компиляторы выполняют аналогичные преобразования только для встроенных типов, когда известны семантика и сложность операций. Концепция регулярности позволяет создателю типа дать гарантии, что выполнение таких преобразований программистами и компиляторами безопасно.

```
{
    // Предусловие: associative(op) ∧ n ⩾ 0
    if (n == I(0)) return r;
    if (n % I(2) != I(0)) r = op(r, a);
    return power_accumulate_0(r, op(a, a), n / I(2), op);
}
```

Если r_0, a_0 и n_0 — исходные значения r, a и n, то при каждом рекурсивном вызове остается в силе следующий инвариант: $ra^n = r_0 a_0^{n_0}$. Эта версия вычисляет не только саму степень, но и степень, умноженную на коэффициент, что служит ее дополнительным преимуществом. Она также воспринимает нуль как значение показателя степени. Но в power_accumulate_0 выполняется ненужное возведение в квадрат после перехода от 1 к 0. Это можно устранить с помощью дополнительного условия:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_1(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // Предусловие: associative(op) ∧ n ⩾ 0
    if (n == I(0)) return r;
    if (n == I(1)) return op(r, a);
    if (n % I(2) != I(0)) r = op(r, a);
    return power_accumulate_1(r, op(a, a), n / I(2), op);
}
```

Результат введения этого дополнительного условия — одно дублирующееся подвыражение и три проверки, которые не являются независимыми. Анализ зависимостей между проверками и упорядочение проверок с учетом ожидаемой частоты приводят к следующему:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_2(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // Предусловие: associative(op) ∧ n ⩾ 0
    if (n % I(2) != I(0)) {
        r = op(r, a);
        if (n == I(1)) return r;
    } else if (n == I(0)) return r;
    return power_accumulate_2(r, op(a, a), n / I(2), op);
}
```

Процедура *со строго соблюдаемой концевой рекурсией* — это такая процедура, в которой все вызовы с концевой рекурсией выполняются с формаль-

ными параметрами процедуры, которые играют роль соответствующих аргументов:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_3(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // Предусловие: associative(op) ∧ n ⩾ 0
    if (n % I(2) != I(0)) {
        r = op(r, a);
        if (n == I(1)) return r;
    } else if (n == I(0)) return r;
    a = op(a, a);
    n = n / I(2);
    return power_accumulate_3(r, a, n, op);
}
```

Процедура со строго соблюдаемой концевой рекурсией может быть преобразована в итерационную процедуру путем замены каждого рекурсивного вызова переходом к началу процедуры по `goto` или путем использования эквивалентной итерационной конструкции:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_4(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // Предусловие: associative(op) ∧ n ⩾ 0
    while (true) {
        if (n % I(2) != I(0)) {
            r = op(r, a);
            if (n == I(1)) return r;
        } else if (n == I(0)) return r;
        a = op(a, a);
        n = n / I(2);
    }
}
```

Этот инвариант рекурсии становится *циклическим инвариантом*.

Если с самого начала $n > 0$, то показатель степени, прежде чем стать равным 0, принимает значение 1. Мы воспользуемся этим, устранив проверку на 0 и усилив предусловие:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_positive_0(Domain(Op) r,
                                       Domain(Op) a, I n,
                                       Op op)
```

```
{
    // Предусловие: associative(op) ∧ n > 0
    while (true) {
        if (n % I(2) != I(0)) {
            r = op(r, a);
            if (n == I(1)) return r;
        }
        a = op(a, a);
        n = n / I(2);
    }
}
```

Это удобно, если известно, что $n > 0$. Разрабатывая компонент, мы часто обнаруживаем новые интерфейсы.

Теперь снова ослабляем предусловие:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_5(Domain(Op) r, Domain(Op) a, I n,
                              Op op)
{
    // Предусловие: associative(op) ∧ n ⩾ 0
    if (n == I(0)) return r;
    return power_accumulate_positive_0(r, a, n, op);
}
```

Мы можем реализовать power исходя из power_accumulate, используя простое тождество:

$$a^n = aa^{n-1}$$

Преобразование состоит в *устранении накопительной переменной*:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_2(Domain(Op) a, I n, Op op)
{
    // Предусловие: associative(op) ∧ n > 0
    return power_accumulate_5(a, a, n - I(1), op);
}
```

Этот алгоритм выполняет больше операций, чем необходимо. Например, если n равно 16, он выполняет семь операций, тогда как требуется только четыре. При нечетном значении n этот алгоритм превосходен. Поэтому мы можем избежать возникновения проблемы путем повторного возведения a в квадрат и деления показателя степени на два, пока он не станет нечетным:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
```

```
Domain(Op) power_3(Domain(Op) a, I n, Op op)
{
    // Предусловие: associative(op) ∧ n > 0
    while (n % I(2) == I(0)) {
        a = op(a, a);
        n = n / I(2);
    }
    n = n / I(2);
    if (n == I(0)) return a;
    return power_accumulate_positive_0(a, op(a, a), n, op);
}
```

Упражнение 3.1. Убедитесь в том, что последние три строки программы являются правильными.

3.4. Процедуры для специального случая

В окончательных версиях мы использовали следующие операции:

```
n / I(2)
n % I(2) == I(0)
n % I(2) != I(0)
n == I(0)
n == I(1)
```

И `/`, и `%` являются дорогостоящими. Мы можем применять операции сдвига и наложения маски к неотрицательным значениям целых чисел со знаком и без знака.

Зачастую бывает удобно выявлять обычно встречающиеся выражения, в которые входят процедуры и постоянные некоторого типа, определяя *специализированные* процедуры. Такие специализированные процедуры нередко могут быть реализованы более эффективно по сравнению с универсальными и поэтому включены в вычислительный базис типа. Если речь идет о встроенных типах, то для специализированных вычислений могут быть предусмотрены машинные команды. А что касается определяемых пользователем типов, то зачастую открываются еще более существенные возможности по оптимизации специализированных вычислений. Например, реализовать деление двух произвольных многочленов труднее, чем деление многочлена на x. Аналогично, деление двух комплексных целых чисел (чисел в форме $a + bi$, где a и b — целые числа, а $i = \sqrt{-1}$) является более трудоемким по сравнению с делением комплексного целого числа на $1 + i$.

Для любого целочисленного типа должны быть предусмотрены следующие специализированные процедуры:

$$
\begin{aligned}
&\mathit{Integer}(\mathrm{I}) \triangleq \\
&\quad \mathsf{successor} : \mathrm{I} \to \mathrm{I} \\
&\qquad n \mapsto n + 1 \\
&\quad \wedge\ \mathsf{predecessor} : \mathrm{I} \to \mathrm{I} \\
&\qquad n \mapsto n - 1 \\
&\quad \wedge\ \mathsf{twice} : \mathrm{I} \to \mathrm{I} \\
&\qquad n \mapsto n + n \\
&\quad \wedge\ \mathsf{half\_nonnegative} : \mathrm{I} \to \mathrm{I} \\
&\qquad n \mapsto \lfloor n/2 \rfloor, \text{ where } n \geqslant 0 \\
&\quad \wedge\ \mathsf{binary\_scale\_down\_nonnegative} : \mathrm{I} \times \mathrm{I} \to \mathrm{I} \\
&\qquad (n, k) \mapsto \lfloor n/2^k \rfloor, \text{ where } n, k \geqslant 0 \\
&\quad \wedge\ \mathsf{binary\_scale\_up\_nonnegative} : \mathrm{I} \times \mathrm{I} \to \mathrm{I} \\
&\qquad (n, k) \mapsto 2^k n, \text{ where } n, k \geqslant 0 \\
&\quad \wedge\ \mathsf{positive} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto n > 0 \\
&\quad \wedge\ \mathsf{negative} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto n < 0 \\
&\quad \wedge\ \mathsf{zero} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto n = 0 \\
&\quad \wedge\ \mathsf{one} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto n = 1 \\
&\quad \wedge\ \mathsf{even} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto (n \bmod 2) = 0 \\
&\quad \wedge\ \mathsf{odd} : \mathrm{I} \to \mathsf{bool} \\
&\qquad n \mapsto (n \bmod 2) \neq 0
\end{aligned}
$$

Упражнение 3.2. Реализуйте эти процедуры для целочисленных типов C++.

Теперь можем привести окончательные реализации процедур возведения в степень, в которых используются специализированные процедуры:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate_positive(Domain(Op) r,
                                     Domain(Op) a, I n,
                                     Op op)
{
    // Предусловие: associative(op) ∧ positive(n)
    while (true) {
      if (odd(n)) {
          r = op(r, a);
          if (one(n)) return r;
      }
      a = op(a, a);
```

```
      n = half_nonnegative(n);
    }
}

template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power_accumulate(Domain(Op) r, Domain(Op) a, I n,
                            Op op)
{
    // Предусловие: associative(op) ∧ ¬negative(n)
    if (zero(n)) return r;
    return power_accumulate_positive(r, a, n, op);
}

template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power(Domain(Op) a, I n, Op op)
{
    // Предусловие: associative(op) ∧ positive(n)
    while (even(n)) {
        a = op(a, a);
        n = half_nonnegative(n);
    }
    n = half_nonnegative(n);
    if (zero(n)) return a;
    return power_accumulate_positive(a, op(a, a), n, op);
}
```

Поскольку известно, что $a^{n+m} = a^n a^m$, вычисление a^0 должно привести к получению единичного элемента для операции op. Мы можем распространить power на нулевые показатели степени, передавая единичный элемент в качестве еще одного параметра[6]:

```
template<typename I, typename Op>
    requires(Integer(I) && BinaryOperation(Op))
Domain(Op) power(Domain(Op) a, I n, Op op, Domain(Op) id)
{
    // Предусловие: associative(op) ∧ ¬negative(n)
    if (zero(n)) return id;
    return power(a, n, op);
}
```

Проект 3.1. Операции умножения с плавающей точкой и сложения не ассоциативны, поэтому могут давать различные результаты при использовании

[6]Еще один способ предусматривает определение функции identity_element, такой, что identity_element(op) возвращает единичный элемент для op.

в качестве операций для power и power_left_associated; установите, позволяет ли power или power_left_ associated получить более точный результат при возведении числа с плавающей точкой в целочисленную степень.

3.5. Параметризация алгоритмов

В power мы используем два различных способа предоставления операций для абстрактного алгоритма.

1. Ассоциативная операция передается в качестве параметра. Это позволяет использовать power применительно к одному и тому же типу с различными операциями, такими как умножение по модулю n.

2. Операции с показателем степени предоставляются как часть вычислительного базиса для типа показателя степени. Мы не выбрали, например, вариант с передачей half_nonnegative в качестве параметра для power, поскольку нам неизвестен случай, в котором имеются конкурирующие реализации half_nonnegative, относящиеся к тому же типу.

Вообще говоря, мы передаем операцию в качестве параметра, если алгоритм может использоваться с различными операциями применительно к одному и тому же типу. Если процедура определена с операцией в качестве параметра, то следует всегда стремиться указывать подходящее стандартное значение. Например, вполне естественным является выбор умножения как стандартной операции, передаваемой в power.

Использование знака операции или имени процедуры с одной и той же семантикой применительно к различным типам называется *перегрузкой*, и мы говорим, что знак операции или имя процедуры являются *перегруженными* по отношению к типу. Например, знак + используется для обозначения операции с натуральными, целыми и рациональными числами, многочленами и матрицами. В математике знак + всегда используется для обозначения ассоциативной и коммутативной операции, поэтому использование + для конкатенации строк было бы непоследовательным. Аналогично, если присутствуют и +, и $\times$, операция $\times$ должна быть дистрибутивной по отношению к +. В функции power функция half_nonnegative является перегруженной применительно к типу показателя степени.

Создавая экземпляры абстрактной процедуры, такой как collision_point или power, мы создаем перегруженные процедуры. Если фактические параметры типа удовлетворяют этим требованиям, то все экземпляры абстрактной процедуры имеют одну и ту же семантику.

3.6. Линейные рекуррентные соотношения

Линейная рекуррентная функция порядка k — это функция f, такая, что

$$f(y_0, \ldots, y_{k-1}) = \sum_{i=0}^{k-1} a_i y_i$$

где коэффициенты $a_0, a_{k-1} \neq 0$. Последовательность $\{x_0, x_1, \cdots\}$ — это *линейная рекуррентная последовательность порядка* k, если есть линейная рекуррентная функция порядка k, скажем, f, и

$$(\forall n \geqslant k)\, x_n = f(x_{n-1}, \ldots, x_{n-k})$$

Следует учитывать, что индексы при x уменьшаются. При условии, что заданы k *начальных значений* $x_0, \ldots, x_{k-1}$ и линейная рекуррентная функция порядка k, мы можем сформировать линейную рекуррентную последовательность с помощью простого итеративного алгоритма. Этот алгоритм требует $n - k + 1$ применений функции для вычисления x_n при $n \geqslant k$. Как показано ниже, мы можем вычислить x_n за $O(\log_2 n)$ шагов, используя power[7]. Если $f(y_0, \ldots, y_{k-1}) = \sum_{i=0}^{k-1} a_i y_i$ — линейная рекуррентная функция порядка k, мы можем рассматривать f как выполнение векторного внутреннего произведения[8]:

$$\begin{bmatrix} a_0 & \cdots & a_{k-1} \end{bmatrix} \begin{bmatrix} y_0 \\ \vdots \\ y_{k-1} \end{bmatrix}$$

Если мы расширим вектор коэффициентов до *сопровождающей матрицы* с единицами на поддиагонали, то сможем одновременно вычислять новое значение x_n и сдвигать старые значения $x_{n-1}, \ldots, x_{n-k+1}$ в правильные позиции для следующей итерации:

$$\begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{k-2} & a_{k-1} \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{n-1} \\ x_{n-2} \\ x_{n-3} \\ \vdots \\ x_{n-k} \end{bmatrix} = \begin{bmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \\ x_{n-k+1} \end{bmatrix}$$

[7]Первый алгоритм $O(\log n)$ для линейных рекуррентных последовательностей описан в [Miller and Brown 1966].

[8]Краткий обзор линейной алгебры приведен в [Kwak and Hong 2004], где линейные рекуррентные последовательности обсуждаются, начиная со стр. 214.

Из ассоциативности умножения матриц следует, что мы можем получить x_n, умножая вектор начальных значений k на сопровождающую матрицу, возведенную в степень $n-k+1$:

$$\begin{bmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \\ x_{n-k+1} \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{k-2} & a_{k-1} \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix}^{n-k+1} \begin{bmatrix} x_{k-1} \\ x_{k-2} \\ x_{k-3} \\ \vdots \\ x_0 \end{bmatrix}$$

Использование power позволяет находить x_n с применением не больше $2\log_2(n-k+1)$ операций умножения матриц. Простой алгоритм умножения матриц требует k^3 операций умножения и k^3-k^2 операций сложения коэффициентов. Поэтому для вычисления x_n требуется не больше $2k^3\log_2(n-k+1)$ раз выполнить умножение и $2(k^3-k^2)\log_2(n-k+1)$ — сложение коэффициентов. Напомним, что k — порядок линейного рекуррентного соотношения, который является постоянным значением[9].

Мы еще не дали определение домена элементов линейной рекуррентной последовательности. Это могут быть целые, рациональные, вещественные или комплексные числа. Единственным требованием является наличие ассоциативной и коммутативной операции сложения и ассоциативной операции умножения, а также дистрибутивность умножения по отношению к сложению[10].

Последовательность f_i, сформированная с помощью линейной рекуррентной функции

$$\mathsf{fib}(y_0, y_1) = y_0 + y_1$$

порядка 2 с начальными значениями $f_0 = 0$ и $f_1 = 1$, называется последовательностью Фибоначчи[11]. Вычислить n-е число Фибоначчи f_n с помощью функции power с умножением матриц 2×2 несложно. Воспользуемся последовательностью Фибоначчи для иллюстрации того, как обеспечить применение меньшего количества операций умножения, чем k^3, в данном конкретном случае. Допустим, что

$$F = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

представляет собой сопровождающую матрицу для линейного рекуррентного соотношения, вырабатывающего последовательность Фибоначчи. Мы можем

[9] В [Fiduccia 1985] показано, как уменьшить этот постоянный коэффициент с помощью модульного умножения многочленов.

[10] Это может быть любой тип, моделирующий полукольцо, определение которого мы дадим в главе 5.

[11] Leonardo Pisano, *Liber Abaci*, первое издание, 1202. Перевод на английский см. в [Sigler 2002]. Сама последовательность впервые упоминается на стр. 404.

показать по индукции следующее

$$F^n = \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix}$$

Действительно:

$$\begin{aligned} F^1 &= \begin{bmatrix} f_2 & f_1 \\ f_1 & f_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \\ F^{n+1} &= FF^n \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} f_{n+1} + f_n & f_n + f_{n-1} \\ f_{n+1} & f_n \end{bmatrix} = \begin{bmatrix} f_{n+2} & f_{n+1} \\ f_{n+1} & f_n \end{bmatrix} \end{aligned}$$

Это позволяет выразить матричное произведение F^m и F^n как

$$\begin{aligned} F^m F^n &= \begin{bmatrix} f_{m+1} & f_m \\ f_m & f_{m-1} \end{bmatrix} \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} f_{m+1}f_{n+1} + f_m f_n & f_{m+1}f_n + f_m f_{n-1} \\ f_m f_{n+1} + f_{m-1} f_n & f_m f_n + f_{m-1} f_{n-1} \end{bmatrix} \end{aligned}$$

Мы можем представить матрицу F^n с помощью пары pair, соответствующей ее нижней строке, (f_n, f_{n-1}), поскольку верхняя строка могла быть вычислена как $(f_{n-1} + f_n, f_n)$, что приводит к получению следующего кода:

```
template<typename I>
    requires(Integer(I))
pair<I, I> fibonacci_matrix_multiply(const pair<I, I>& x,
                                     const pair<I, I>& y)
{
    return pair<I, I>(
        x.m0 * (y.m1 + y.m0) + x.m1 * y.m0,
        x.m0 * y.m0 + x.m1 * y.m1);
}
```

Эта процедура выполняет только четыре умножения вместо восьми, которые требовались при использовании универсальной процедуры умножения матриц 2×2. Первым элементом нижней строки F^n является f_n, поэтому для вычисления f_n служит следующая процедура:

```
template<typename I>
    requires(Integer(I))
I fibonacci(I n)
{
```

```
    // Предусловие: n ⩾ 0
    if (n == I(0)) return I(0);
    return power(pair<I, I>(I(1), I(0)),
                 n,
                 fibonacci_matrix_multiply<I>).m0;
}
```

3.7. Процедуры накопления

В предыдущей главе определено действие как дуалистичное по отношению к преобразованию. Можно определить и процедуру, дуалистичную по отношению к бинарной операции, если последняя используется в инструкции, подобной следующей:

```
x = op(x, y);
```

Изменение состояния объекта путем комбинирования его с другим объектом с помощью бинарной операции определяет *процедуру накопления* на объекте. Процедура накопления является определимой в терминах бинарной операции, и наоборот:

```
void op_accumulate(T& x, const T& y) { x = op(x, y); }
  // процедура накопления из бинарной операции
```

и

```
T op(T x, const T& y) { op_accumulate(x, y); return x; }
  // бинарная операция из процедуры накопления
```

Как и применительно к действиям, иногда более эффективны независимые реализации, если нужно предусмотреть и операцию, и процедуры накопления.

Упражнение 3.3. Перепишите все алгоритмы в этой главе в терминах процедур накопления.

Проект 3.2. Создайте библиотеку для выработки линейных рекуррентных последовательностей на основе результатов [Miller and Brown 1966] и [Fiduccia 1985].

3.8. Резюме

Алгоритмы рассматриваются как *абстрактные*, если они могут использоваться с различными моделями, удовлетворяющими одним и тем же требованиям, таким как ассоциативность. Оптимизация кода зависит от того, можно

ли применять рассуждения, основанные на равенствах; если неизвестно, являются ли типы регулярными, то возможности оптимизации довольно ограничены. Применение специализированных процедур позволяет сделать код более эффективным и даже более абстрактным. Совместное использование математики и абстрактных алгоритмов приводит к созданию алгоритмов с удивительными свойствами, например, вырабатывающих n-й элемент линейного рекуррентного соотношения за логарифмическое время.

Глава 4
Линейные упорядочения

В настоящей главе описаны свойства бинарных отношений, такие как транзитивность и симметричность. В частности, мы вводим понятия полного и слабого линейных упорядочений, а также концепцию стабильности функций на основе линейного упорядочения: сохранения порядка, присутствующего в аргументах, для эквивалентных элементов. Мы обобщаем min и max до функций выбора порядка, таких как медиана трех элементов, и вводим способ управления сложностью их реализации путем приведения к ограниченным подзадачам.

4.1. Классификация отношений

Отношение — это предикат, принимающий два параметра одного и того же типа:

$$\begin{aligned} &\mathit{Relation}(\mathrm{Op}) \triangleq \\ &\qquad \mathit{HomogeneousPredicate}(\mathrm{Op}) \\ &\quad \wedge\ \mathsf{Arity}(\mathrm{Op}) = 2 \end{aligned}$$

Отношение *транзитивно*, если из того, что оно выполняется между a и b и между b и c, следует, что оно выполняется между a и c:

$$\begin{aligned} &\textbf{property}(\mathrm{R} : \mathit{Relation}) \\ &\mathsf{transitive} : \mathrm{R} \\ &\quad \mathrm{r} \mapsto (\forall \mathrm{a}, \mathrm{b}, \mathrm{c} \in \mathsf{Domain}(\mathrm{R}))\,(\mathrm{r}(\mathrm{a}, \mathrm{b}) \wedge \mathrm{r}(\mathrm{b}, \mathrm{c}) \Rightarrow \mathrm{r}(\mathrm{a}, \mathrm{c})) \end{aligned}$$

Примеры транзитивных отношений — равенство, равенство первых элементов пар, достижимость в орбите и делимость.

Отношение *строго*, если оно никогда не выполняется между элементом и им самим; отношение *рефлексивно*, если это всегда выполняется между элементом и им самим:

property(R : *Relation*)
strict : R
$r \mapsto (\forall a \in \mathsf{Domain}(R))\, \neg r(a, a)$

property(R : *Relation*)
reflexive : R
$r \mapsto (\forall a \in \mathsf{Domain}(R))\, r(a, a)$

Транзитивные отношения во всех предыдущих примерах рефлексивны; отношение "собственный делитель" является строгим.

Упражнение 4.1. Приведите пример отношения, которое не является ни строгим, ни рефлексивным.

Отношение *симметрично*, если из того, что оно выполняется в одном направлении, следует, что оно выполняется в другом; отношение *асимметрично*, если оно никогда не выполняется в обоих направлениях одновременно:

property(R : *Relation*)
symmetric : R
$r \mapsto (\forall a, b \in \mathsf{Domain}(R))\, (r(a, b) \Rightarrow r(b, a))$

property(R : *Relation*)
asymmetric : R
$r \mapsto (\forall a, b \in \mathsf{Domain}(R))\, (r(a, b) \Rightarrow \neg r(b, a))$

Пример симметричного транзитивного отношения — "брат"; пример асимметричного транзитивного отношения — "предок".

Упражнение 4.2. Приведите пример симметричного отношения, которое не является транзитивным.

Упражнение 4.3. Приведите пример симметричного отношения, которое не является рефлексивным.

Если дано отношение $r(a, b)$, то существуют *производные отношения* с тем же доменом:

$$\begin{aligned} \mathsf{complement}_r(a, b) &\Leftrightarrow \neg r(a, b) \\ \mathsf{converse}_r(a, b) &\Leftrightarrow r(b, a) \\ \mathsf{complement\_of\_converse}_r(a, b) &\Leftrightarrow \neg r(b, a) \end{aligned}$$

Что касается симметричного отношения, то единственным интересным отношением, которое может быть получено на его основе, является дополнение, поскольку обратное отношение эквивалентно исходному.

Отношение *эквивалентно*, если оно транзитивно, рефлексивно и симметрично:

property(R : *Relation*)
equivalence : R
$r \mapsto \mathsf{transitive}(r) \wedge \mathsf{reflexive}(r) \wedge \mathsf{symmetric}(r)$

Примеры отношений эквивалентности — равенство, геометрическая конгруэнтность и целочисленная конгруэнтность по модулю n.

Лемма 4.1. Если r — отношение эквивалентности, то $a = b \Rightarrow r(a, b)$.

Отношение эквивалентности разделяет свой домен на множество *классов эквивалентности*: подмножеств, содержащих все элементы, эквивалентные данному элементу. Мы часто можем реализовать отношение эквивалентности, определяя *ключевую функцию*, которая возвращает уникальное значение для всех элементов в каждом классе эквивалентности. Применение равенства к результатам ключевой функции определяет эквивалентность:

property(F : *UnaryFunction*, R : *Relation*)
requires(Domain(F) = Domain(R))
key_function : F × R
$(f, r) \mapsto (\forall a, b \in \mathsf{Domain}(F))\,(r(a, b) \Leftrightarrow f(a) = f(b))$

Лемма 4.2. $\mathsf{key\_function}(f, r) \Rightarrow \mathsf{equivalence}(r)$

4.2. Полные и слабые упорядочения

Отношение является *полным упорядочением*, если оно транзитивно и подчиняется *закону трихотомии*, согласно которому для каждой пары элементов выполняется одно и только одно из следующего: отношение, его обращение или равенство:

property(R : *Relation*)
total_ordering : R
$r \mapsto \mathsf{transitive}(r) \wedge$
$(\forall a, b \in \mathsf{Domain}(R))$ выполняется одно и только одно из следующего:
$r(a, b), r(b, a),$ или $a = b$

Отношение является *слабым упорядочением*, если оно транзитивно и существует отношение эквивалентности на том же домене, такое, что исходное отношение подчиняется *закону слабой трихотомии*, согласно которому для каждой пары элементов выполняется одно и только одно из следующего: отношение, его обращение или эквивалентность:

property(R : *Relation*, E : *Relation*) **requires**(Domain(R) = Domain(E))
weak_ordering : R
$r \mapsto \mathsf{transitive}(r) \wedge (\exists e \in E)\,\mathsf{equivalence}(e) \wedge$
$(\forall a, b \in \mathsf{Domain}(R))$ выполняется одно и только одно из следующего: $r(a, b), r(b, a)$ или $e(a, b)$

Если дано отношение r, то отношение $\neg r(a,b) \wedge \neg r(b,a)$ называется *симметричным дополнением* для r.

Лемма 4.3. Симметричное дополнение слабого упорядочения — это отношение эквивалентности.

Примеры слабого упорядочения — пары, упорядоченные по их первым элементам, и служащие, упорядоченные по зарплате.

Лемма 4.4. Полное упорядочение — слабое упорядочение.

Лемма 4.5. Слабое упорядочение асимметрично.

Лемма 4.6. Слабое упорядочение строго.

Ключевая функция f на множестве T, наряду с полным упорядочением r на кодомене f, определяет слабое упорядочение $\widetilde{r}(x,y) \Leftrightarrow r(f(x),f(y))$.

Мы называем полные и слабые упорядочения *линейными* упорядочениями, поскольку на них распространяются соответствующие законы трихотомии.

4.3. Выбор порядка

Если дано слабое упорядочение r и два объекта, a и b, из домена r, то приобретает смысл вопрос: какой из них является минимумом? Очевидно, как определить минимум, когда выполняется r или его обращение между a и b, но не столь очевидно, как это сделать, если элементы эквивалентны. Аналогичная проблема возникает, если вопрос состоит в том, какой из элементов является максимумом.

Свойство, позволяющее справиться с этой проблемой, известно как *стабильность*. Неформально алгоритм является *стабильным*, если он сохраняет исходный порядок эквивалентных объектов. Итак, если рассматривать минимум и максимум как выбор в списке из двух аргументов, соответственно наименьшего и второго наименьшего по счету, то стабильность требует, чтобы при вызове с эквивалентными элементами операция минимума возвратила первый элемент, а максимума — второй[1].

Мы можем обобщить минимум и максимум на выбор (j,k)-го порядка, где $k > 0$ указывает количество аргументов, а $0 \leqslant j < k$ указывает, что должен быть выбран j-й наименьший аргумент. Чтобы формализовать применяемое нами понятие стабильности, предположим, что каждый из аргументов k связан с уникальным натуральным числом, называемым его *индексом стабильности*.

[1] В следующих главах мы распространим понятие стабильности на другие категории алгоритмов.

При условии, что дано исходное слабое упорядочение r, определяем *усиленное* отношение $\widehat{r}$ на парах (объект, индекс стабильности):

$$\widehat{r}((a, i_a), (b, i_b)) \Leftrightarrow r(a, b) \vee (\neg r(b, a) \wedge i_a < i_b)$$

Если мы реализуем алгоритм выбора порядка в терминах $\widehat{r}$, то неоднозначные случаи, вызванные эквивалентными аргументами, исключаются. Естественным стандартным значением для индекса стабильности аргумента является его порядковая позиция в списке аргументов.

Разумеется, это усиленное отношение $\widehat{r}$ предоставляет нам мощное инструментальное средство проведения рассуждений о стабильности, но несложно определить простые процедуры выбора порядка, не прибегая явно к использованию индексов стабильности. Следующая реализация минимума возвращает a, если a и b эквивалентны, удовлетворяя принятому нами определению стабильности[2]:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_0_2(const Domain(R)& a,
                            const Domain(R)& b, R r)
{
    // Предусловие: weak_ordering(r)
    if (r(b, a)) return b;
    return a;
}
```

Аналогично следующая реализация максимума возвращает b, если a и b эквивалентны, также в соответствии с принятым определением стабильности[3]:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_2(const Domain(R)& a,
                            const Domain(R)& b, R r)
{
    // Предусловие: weak_ordering(r)

    if (r(b, a)) return a;
    return b;
}
```

В оставшейся части главы подразумевается выполнение предусловия weak_ordering(r).

[2]Описание принятого нами соглашения об именовании приведено ниже в этом разделе.

[3]STL необоснованно требует, чтобы функция $\max(a, b)$ возвращала a, если a и b эквивалентны.

Несомненно, было бы полезно иметь другие процедуры выбора порядка для k аргументов, но сложность составления подобной процедуры выбора порядка быстро возрастает с увеличением k, а количество различных процедур, которые могли бы потребоваться, весьма велико. Поэтому мы применяем подход, названный нами *сведением к ограниченным подзадачам*, который позволяет разрешить обе проблемы. Мы разрабатываем семейство процедур, которые принимают определенный объем информации об относительном упорядочении их аргументов.

При этом важно определить систему обозначения этих процедур. Каждое имя начинается с select_j_k, где $0 \leqslant j < k$, указывая на выбор j-го элемента из из k аргументов в соответствии с заданным упорядочением. Затем добавляется последовательность символов, указывающая предусловие упорядочения параметров, которое выражено в виде наращиваемых цепочек. Например, суффикс _ab означает, что по порядку расположены первые два параметра, а _abd показывает, что по порядку расположены первый, второй и четвертый параметры. Больше чем один подобный суффикс появляется, когда имеются предусловия, относящиеся к различным цепочкам параметров.

Например, несложно реализовать минимум и максимум для трех элементов:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_0_3(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c, R r)
{
    return select_0_2(select_0_2(a, b, r), c, r);
}

template<typename R>
    requires(Relation(R))
const Domain(R)& select_2_3(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c, R r)
{
    return select_1_2(select_1_2(a, b, r), c, r);
}
```

Легко также найти медиану трех элементов, если известно, что первые два элемента находятся в порядке по возрастанию:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_3_ab(const Domain(R)& a,
                               const Domain(R)& b,
```

```
                                const Domain(R)& c, R r)
{
    if (!r(c, b)) return b;       // a, b, c отсортированы
    return select_1_2(a, c, r);   // b не является медианой
}
```

Определение предусловия для select_1_3_ab требует только одного сравнения. Параметры передаются по постоянной ссылке, поэтому перемещение данных не происходит:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_3(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c, R r)
{
    if (r(b, a)) return select_1_3_ab(b, a, c, r);
    return              select_1_3_ab(a, b, c, r);
}
```

В наихудшем случае в select_1_3 выполняются три сравнения. В этой функции производятся два сравнения, только если c является максимумом среди a, b, c, а поскольку это происходит в одной трети случаев, среднее количество сравнений равно $2\frac{2}{3}$, при условии, что распределение входных данных равномерно.

Для поиска второго из наименьших среди n элементов требуется по крайней мере $n+\lceil\log_2 n\rceil-2$ сравнений[4]. В частности, для поиска второго элемента из четырех требуются четыре сравнения.

Второй аргумент из четырех выбрать несложно, если известно, что обе пары аргументов, и первая, и вторая, заданы в порядке возрастания:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_4_ab_cd(const Domain(R)& a,
                                  const Domain(R)& b,
                                  const Domain(R)& c,
                                  const Domain(R)& d, R r) {
    if (r(c, a)) return select_0_2(a, d, r);
    return              select_0_2(b, c, r);
}
```

Предусловие для select_1_4_ab_cd может быть установлено с применением одного сравнения, если известно, что первая пара аргументов приведена в порядке возрастания:

[4]Этот результат был предсказан Йозефом Шрейером и доказан Сергеем Кислицыным [Knuth 1998, теорема S, стр. 209].

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_4_ab(const Domain(R)& a,
                               const Domain(R)& b,
                               const Domain(R)& c,
                               const Domain(R)& d, R r) {
    if (r(d, c)) return select_1_4_ab_cd(a, b, d, c, r);
    return              select_1_4_ab_cd(a, b, c, d, r);
}
```

Предусловие для select_1_4_ab может быть установлено с помощью одного сравнения:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& select_1_4(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c,
                            const Domain(R)& d, R r) {
    if (r(b, a)) return select_1_4_ab(b, a, c, d, r);
    return              select_1_4_ab(a, b, c, d, r);
}
```

Упражнение 4.4. Реализуйте select_2_4.

Поддержка стабильности сетей выбора порядка вплоть до порядка 4 оказалась не слишком затруднительной. Но начиная с порядка 5 возникают ситуации, в котором процедура, соответствующая ограниченной подзадаче, вызывается из исходной вызывающей программы с аргументами, не приведенными в требуемом порядке, что приводит к нарушению стабильности. Успешно справляться с такими ситуациями позволяет систематический подход, который состоит в том, что наряду с фактическими параметрами передаются индексы стабильности и используется усиленное отношение $\widehat{\mathrm{r}}$. Мы избегаем дополнительного увеличения времени выполнения благодаря использованию параметров в виде целочисленных шаблонов.

Мы именуем индексы стабильности как ia, ib, ... в соответствии с параметрами a, b и т. д. Для получения усиленного отношения $\widehat{\mathrm{r}}$ используется шаблон функционального объекта compare_strict_or_reflexive, принимающий параметр шаблона bool, значение true которого показывает, что индексы стабильности его аргументов находятся в порядке возрастания:

```
template<bool strict, typename R>
    requires(Relation(R))
struct compare_strict_or_reflexive;
```

При создании экземпляра compare_strict_or_reflexive мы предоставляем соответствующий булев аргумент шаблона:

```
template<int ia, int ib, typename R>
    requires(Relation(R))
const Domain(R)& select_0_2(const Domain(R)& a,
                            const Domain(R)& b, R r)
{
    compare_strict_or_reflexive<(ia < ib), R> cmp;
    if (cmp(b, a, r)) return b;
    return a;
}
```

Мы специализируем compare_strict_or_reflexive для следующих двух случаев: 1) индексы стабильности находятся в порядке возрастания, и в этом случае используется исходное строгое отношение r; и 2) имеет место порядок убывания; при этом используется соответствующая рефлексивная версия r:

```
template<typename R>
    requires(Relation(R))
struct compare_strict_or_reflexive<true, R> // строгая
{
    bool operator()(const Domain(R)& a,
                    const Domain(R)& b, R r)
    {
        return r(a, b);
    }
};

template<typename R>
    requires(Relation(R))
struct compare_strict_or_reflexive<false, R> // рефлексивная
{
    bool operator()(const Domain(R)& a,
                    const Domain(R)& b, R r)
    {
        return !r(b, a); // complement_of_converse_r(a, b)
    }
};
```

Когда процедура выбора порядка с индексами стабильности вызывает другую такую процедуру, тогда передаются индексы стабильности, соответствующие параметрам, в том же порядке, в каком они появляются в вызове:

```
template<int ia, int ib, int ic, int id, typename R>
    requires(Relation(R))
const Domain(R)& select_1_4_ab_cd(const Domain(R)& a,
                                  const Domain(R)& b,
                                  const Domain(R)& c,
                                  const Domain(R)& d, R r)
```

```
{
    compare_strict_or_reflexive<(ia < ic), R> cmp;
    if (cmp(c, a, r)) return
        select_0_2<ia,id>(a, d, r);
    return
        select_0_2<ib,ic>(b, c, r);
}

template<int ia, int ib, int ic, int id, typename R>
    requires(Relation(R))
const Domain(R)& select_1_4_ab(const Domain(R)& a,
                               const Domain(R)& b,
                               const Domain(R)& c,
                               const Domain(R)& d, R r)
{
    compare_strict_or_reflexive<(ic < id), R> cmp;
    if (cmp(d, c, r)) return
        select_1_4_ab_cd<ia,ib,id,ic>(a, b, d, c, r);
    return
        select_1_4_ab_cd<ia,ib,ic,id>(a, b, c, d, r);
}

template<int ia, int ib, int ic, int id, typename R>
    requires(Relation(R))
const Domain(R)& select_1_4(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c,
                            const Domain(R)& d, R r)
{
    compare_strict_or_reflexive<(ia < ib), R> cmp;
    if (cmp(b, a, r)) return
        select_1_4_ab<ib,ia,ic,id>(b, a, c, d, r);
    return
        select_1_4_ab<ia,ib,ic,id>(a, b, c, d, r);
}
```

Теперь мы готовы реализовать процедуры выбора порядка 5:

```
template<int ia, int ib, int ic, int id, int ie, typename R>
    requires(Relation(R))
const Domain(R)& select_2_5_ab_cd(const Domain(R)& a,
                                  const Domain(R)& b,
                                  const Domain(R)& c,
                                  const Domain(R)& d,
                                  const Domain(R)& e, R r)
{
    compare_strict_or_reflexive<(ia < ic), R> cmp;
```

```
    if (cmp(c, a, r)) return
        select_1_4_ab<ia,ib,id,ie>(a, b, d, e, r);
    return
        select_1_4_ab<ic,id,ib,ie>(c, d, b, e, r);
}

template<int ia, int ib, int ic, int id, int ie, typename R>
    requires(Relation(R))

const Domain(R)& select_2_5_ab(const Domain(R)& a,
                               const Domain(R)& b,
                               const Domain(R)& c,
                               const Domain(R)& d,
                               const Domain(R)& e, R r)
{
    compare_strict_or_reflexive<(ic < id), R> cmp;
    if (cmp(d, c, r)) return
        select_2_5_ab_cd<ia,ib,id,ic,ie>(
                        a, b, d, c, e, r);
    return
        select_2_5_ab_cd<ia,ib,ic,id,ie>(
                        a, b, c, d, e, r);
}

template<int ia, int ib, int ic, int id, int ie, typename R>
    requires(Relation(R))
const Domain(R)& select_2_5(const Domain(R)& a,
                            const Domain(R)& b,
                            const Domain(R)& c,
                            const Domain(R)& d,
                            const Domain(R)& e, R r)
{
    compare_strict_or_reflexive<(ia < ib), R> cmp;
    if (cmp(b, a, r)) return
        select_2_5_ab<ib,ia,ic,id,ie>(b, a, c, d, e, r);
    return
        select_2_5_ab<ia,ib,ic,id,ie>(a, b, c, d, e, r);
}
```

Лемма 4.7. select_2_5 выполняет шесть сравнений.

Упражнение 4.5. Найдите алгоритм определения медианы для 5 аргументов, который в среднем выполняет немного меньше сравнений.

Мы можем оформить процедуру выбора порядка с внешней процедурой, которая предоставляет в качестве индексов стабильности любой строго возрастающий ряд целочисленных постоянных; обычно используются подряд идущие целые числа начиная с 0:

```
template<typename R>
    requires(Relation(R))
const Domain(R)& median_5(const Domain(R)& a,
                          const Domain(R)& b,
                          const Domain(R)& c,
                          const Domain(R)& d,
                          const Domain(R)& e, R r)
{
    return select_2_5<0,1,2,3,4>(a, b, c, d, e, r);
}
```

Упражнение 4.6. Докажите стабильность каждой процедуры выбора порядка, приведенной в этом разделе.

Упражнение 4.7. Проверьте правильность и стабильность каждой процедуры выбора порядка в этом разделе с помощью исчерпывающего тестирования.

Проект 4.1. Спроектируйте набор необходимых и достаточных условий сохранения стабильности при составлении процедур выбора порядка.

Проект 4.2. Создайте библиотеку процедур проверки на минимум для стабильной сортировки и слияния[5]. Минимизируйте не только количество сравнений, но и количество перемещений данных.

4.4. Естественное полное упорядочение

Равенство применительно к типу определяется уникальным образом, поскольку равенство значений конкретного типа показывает, что эти значения представляют одну и ту же сущность. Зачастую нельзя уникально определить естественное полное упорядочение применительно к некоторому типу. Что касается конкретных видов, то часто можно определить много полных и слабых упорядочений, причем ни одно из них не играет какую-то особую роль. Применительно к абстрактным видам может быть задано одно специальное полное упорядочение, в котором поддерживаются все фундаментальные операции для конкретного вида. Такое упорядочение именуется *естественным полным упорядочением* и обозначается символом $<$, как показано ниже.

$$\begin{aligned} &\mathit{TotallyOrdered}(\mathsf{T}) \triangleq \\ &\quad \mathit{Regular}(\mathsf{T}) \\ &\quad \wedge\ <\colon \mathsf{T} \times \mathsf{T} \to \mathsf{bool} \\ &\quad \wedge\ \mathsf{total\_ordering}(<) \end{aligned}$$

Например, фундаментальные операции поддерживаются в естественном полном упорядочении на целых числах:

[5] См. [Knuth 1998, Section 5.3: Optimum Sorting].

$$a < \mathsf{successor}(a)$$
$$a < b \Rightarrow \mathsf{successor}(a) < \mathsf{successor}(b)$$
$$a < b \Rightarrow a + c < b + c$$
$$a < b \wedge 0 < c \Rightarrow ca < cb$$

Иногда тип не имеет естественного полного упорядочения. Например, комплексные числа и личные дела служащих не имеют естественных полных упорядочений. Мы требуем, чтобы регулярные типы обеспечивали *стандартное полное упорядочение* (иногда сокращенно называемое *стандартным упорядочением*) для предоставления возможности логарифмического поиска. Примером стандартного полного упорядочения значений в условиях отсутствия естественного полного упорядочения может служить лексикографическое упорядочение для комплексных чисел. Если естественное полное упорядочение существует, оно совпадает со стандартным упорядочением. Мы используем следующую систему обозначений:

	Определения	C++
Стандартное упорядочение для `T`	less_T	`less<T>`

4.5. Семейства производных процедур

Некоторые процедуры естественным образом объединяются в семейства. Если некоторые процедуры в семействе уже определены, то на этой основе могут быть легко получены определения других процедур. Всякий раз, когда определено равенство, становится определенным неравенство — дополнение равенства; операторы $=$ и $\neq$ должны быть определены непротиворечивым способом. Для каждого полностью упорядоченного типа должны быть определены одновременно все четыре оператора, $<$, $>$, $\leqslant$ и $\geqslant$, таким образом, чтобы имело место следующее:

$$a > b \Leftrightarrow b < a$$
$$a \leqslant b \Leftrightarrow \neg(b < a)$$
$$a \geqslant b \Leftrightarrow \neg(a < b)$$

4.6. Расширение процедур выбора порядка

Процедуры выбора порядка в этой главе не возвращают объект, который может быть изменен, поскольку они работают с постоянными ссылками. Но удобнее и проще, если предусмотрены версии, которые возвращают изменяемый объект, что позволяет использовать их в левой стороне операции присваивания или в качестве изменяемого аргумента для действия или процедуры накопления. Перегруженная изменяемая версия процедуры выбора порядка может быть реализована путем удаления в неизменяемой версии ключевого

слова const в объявлении каждого типа параметра и типа результата. Например, применяемая нами версия select_0_2 может быть дополнена следующим образом:

```
template<typename R>
    requires(Relation(R))
Domain(R)& select_0_2(Domain(R)& a, Domain(R)& b, R r)
{
    if (r(b, a)) return b;
    return a;
}
```

Кроме того, в любой библиотеке должны быть предусмотрены версии для полностью упорядоченных типов (с операцией $<$), так как это — обычный случай. Это означает, что имеются четыре версии каждой процедуры.

Согласно законам трихотомии и слабой трихотомии, которым подчиняется полное и слабое упорядочение, вместо двузначного отношения можно использовать трехзначную процедуру сравнения, в связи с тем, что в некоторых ситуациях это позволяет избежать дополнительного вызова процедуры.

Упражнение 4.8. Перепишите алгоритмы в этой главе с использованием трехзначного сравнения.

4.7. Резюме

Аксиомы полного и слабого упорядочения обеспечивают интерфейс для подключения определенных упорядочений к универсальным алгоритмам. Систематический поиск решений небольших задач позволяет облегчить декомпозицию крупных задач. Процедуры объединяются в семейства со взаимосвязанной семантикой.

Глава 5

Упорядоченные алгебраические структуры

В настоящей главе представлена иерархия концепций из абстрактной алгебры, начиная с полугрупп и заканчивая кольцами и модулями. Затем мы объединяем алгебраические концепции с понятием полного упорядочения. Если упорядоченные алгебраические структуры являются архимедовыми, мы можем определить эффективный алгоритм вычисления частного и остатка. Частное и остаток, в свою очередь, приводят к обобщенной версии алгоритма Евклида для наибольшего общего делителя. Мы кратко рассматриваем связанные с концепциями логические понятия, такие как непротиворечивость и независимость. В заключение мы обсуждаем компьютерную целочисленную арифметику.

5.1. Основные алгебраические структуры

Элемент называется *нейтральным элементом* бинарной операции, если при его совместном применении с любым другим элементом в качестве первого или второго аргумента операция возвращает этот другой элемент:

property$(\mathsf{T} : \mathit{Regular}, \mathsf{Op} : \mathit{BinaryOperation})$
  requires$(\mathsf{T} = \mathsf{Domain}(\mathsf{Op}))$
$\mathsf{identity\_element} : \mathsf{T} \times \mathsf{Op}$
  $(e, \mathrm{op}) \mapsto (\forall a \in \mathsf{T})\, \mathrm{op}(a, e) = \mathrm{op}(e, a) = a$

Лемма 5.1. Нейтральный элемент уникален:

$$\mathsf{identity\_element}(e, \mathrm{op}) \wedge \mathsf{identity\_element}(e', \mathrm{op}) \Rightarrow e = e'$$

Пустая строка — это нейтральный элемент для конкатенации строк. Матрица $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ является нейтральным элементом умножения для матриц 2×2, тогда как $\left(\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ служит для них нейтральным элементом сложения.

Преобразование называется *обратной операцией* бинарной операции относительно заданного элемента (обычно нейтрального элемента бинарной операции), если оно удовлетворяет следующему:

property(F : *Transformation*, T : *Regular*, Op : *BinaryOperation*)
  requires(Domain(F) = T = Domain(Op))
inverse_operation : F × T × Op
  $(inv, e, op) \mapsto (\forall a \in T)\, op(a, inv(a)) = op(inv(a), a) = e$

Лемма 5.2. n^3 является мультипликативным обратным элементом по модулю 5 положительного целого числа $n \neq 0$.

Бинарная операция *коммутативна*, если ее результат остается тем же после перемены местами ее аргументов:

property(Op : *BinaryOperation*)
commutative : Op
  $op \mapsto (\forall a, b \in \mathsf{Domain}(Op))\, op(a, b) = op(b, a)$

Композиция преобразований ассоциативна, но не коммутативна.

Множество с ассоциативной операцией называется *полугруппой*. Как было отмечено в главе 3, знак + всегда используется для обозначения ассоциативной, коммутативной операции, поэтому тип с операцией + называется *аддитивной полугруппой*:

AdditiveSemigroup(T) $\triangleq$
    Regular(T)
  $\wedge$ + : T × T → T
  $\wedge$ associative(+)
  $\wedge$ commutative(+)

Умножение иногда не коммутативно. В качестве примера можно указать умножение матриц.

MultiplicativeSemigroup(T) $\triangleq$
    Regular(T)
  $\wedge$ · : T × T → T
  $\wedge$ associative(·)

Мы используем следующую систему обозначений:

	Определения	C++
Умножение	·	*

Полугруппа с нейтральным элементом называется *моноидом*. Нейтральный элемент сложения обозначается как 0, что приводит к определению *аддитивного моноида*:

AdditiveMonoid(T) $\triangleq$
    AdditiveSemigroup(T)
  $\wedge$ 0 ∈ T
  $\wedge$ identity_element(0, +)

Мы используем следующую систему обозначений:

	Определения	C++
Нейтральный элемент сложения	0	`T(0)`

Неотрицательные вещественные числа составляют аддитивный моноид, как и матрицы с натуральными числами в качестве их коэффициентов.

Нейтральный элемент умножения обозначается как 1, что приводит к определению *мультипликативного моноида*:

$$\begin{aligned}&\mathit{MultiplicativeMonoid}(\mathsf{T}) \triangleq \\ &\quad \mathit{MultiplicativeSemigroup}(\mathsf{T}) \\ &\wedge\ 1 \in \mathsf{T} \\ &\wedge\ \mathsf{identity\_element}(1, \cdot)\end{aligned}$$

Мы используем следующую систему обозначений:

	Определения	C++
Нейтральный элемент умножения	1	`T(1)`

Матрицы с целочисленными коэффициентами составляют мультипликативный моноид.

Моноид с обратной операцией называется *группой*. Если аддитивный моноид имеет обратную операцию, последняя обозначается унарным знаком $-$ и существует производная операция, называемая *вычитанием*, которая обозначается бинарным знаком $-$. Это приводит к определению *аддитивной группы*:

$$\begin{aligned}&\mathit{AdditiveGroup}(\mathsf{T}) \triangleq \\ &\quad \mathit{AdditiveMonoid}(\mathsf{T}) \\ &\wedge\ - : \mathsf{T} \to \mathsf{T} \\ &\wedge\ \mathsf{inverse\_operation}(\text{unary } -, 0, +) \\ &\wedge\ - : \mathsf{T} \times \mathsf{T} \to \mathsf{T} \\ &\qquad (a, b) \mapsto a + (-b)\end{aligned}$$

Матрицы с целочисленными коэффициентами составляют аддитивную группу.

Лемма 5.3. В аддитивной группе $-0 = 0$.

Наряду с концепцией аддитивной группы есть соответствующая концепция *мультипликативной группы*. В этой концепции обратная операция называется *мультипликативной инверсией*, а также имеется производная операция, именуемая *делением*, которая обозначается бинарным знаком $/$:

$$\begin{aligned}&\mathit{MultiplicativeGroup}(\mathsf{T}) \triangleq \\ &\quad \mathit{MultiplicativeMonoid}(\mathsf{T}) \\ &\wedge\ \mathsf{multiplicative\_inverse} : \mathsf{T} \to \mathsf{T} \\ &\wedge\ \mathsf{inverse\_operation}(\mathsf{multiplicative\_inverse}, 1, \cdot) \\ &\wedge\ / : \mathsf{T} \times \mathsf{T} \to \mathsf{T} \\ &\qquad (a, b) \mapsto a \cdot \mathsf{multiplicative\_inverse}(b) \\ &\mathsf{multiplicative\_inverse}(x) \text{ is written as } x^{-1}.\end{aligned}$$

Множество $\{\cos\theta + i\sin\theta\}$ комплексных чисел на единичной окружности составляет коммутативную мультипликативную группу. Унимодулярная группа $GL_n(\mathbb{Z})$ (матрицы $n \times n$ с целочисленными коэффициентами и определителем, равным ± 1) составляет некоммутативную мультипликативную группу.

Две концепции могут быть объединены применительно к одному и тому же типу с помощью аксиом, соединяющих их операции. Если применительно к некоторому типу заданы обе операции, и $+$, и $\cdot$, то их взаимодействие устанавливается с помощью аксиом, определяющих *полукольцо*:

$$
\begin{aligned}
&\mathit{Semiring}(T) \triangleq \\
&\qquad \mathit{AdditiveMonoid}(T) \\
&\quad \wedge\ \mathit{MultiplicativeMonoid}(T) \\
&\quad \wedge\ 0 \neq 1 \\
&\quad \wedge\ (\forall a \in T)\, 0 \cdot a = a \cdot 0 = 0 \\
&\quad \wedge\ (\forall a, b, c \in T) \\
&\qquad\qquad a \cdot (b + c) = a \cdot b + a \cdot c \\
&\qquad\quad \wedge\ (b + c) \cdot a = b \cdot a + c \cdot a
\end{aligned}
$$

Аксиома об умножении на 0 называется *свойством аннигиляции*. Последняя аксиома, соединяющая $+$ и $\cdot$, называется *дистрибутивностью*.

Матрицы с неотрицательными целочисленными коэффициентами составляют полукольцо.

$$
\begin{aligned}
&\mathit{CommutativeSemiring}(T) \triangleq \\
&\qquad \mathit{Semiring}(T) \\
&\quad \wedge\ \mathsf{commutative}(\cdot)
\end{aligned}
$$

Неотрицательные целые числа составляют коммутативное полукольцо.

$$
\begin{aligned}
&\mathit{Ring}(T) \triangleq \\
&\qquad \mathit{AdditiveGroup}(T) \\
&\quad \wedge\ \mathit{Semiring}(T)
\end{aligned}
$$

Матрицы с целочисленными коэффициентами составляют кольцо.

$$
\begin{aligned}
&\mathit{CommutativeRing}(T) \triangleq \\
&\qquad \mathit{AdditiveGroup}(T) \\
&\quad \wedge\ \mathit{CommutativeSemiring}(T)
\end{aligned}
$$

Целые числа составляют коммутативное кольцо; многочлены с целочисленными коэффициентами составляют коммутативное кольцо.

Реляционная концепция — это концепция, определенная на двух типах. *Полумодуль* — это реляционная концепция, которая соединяет аддитивный моноид и коммутативное полукольцо:

$$
\begin{aligned}
&\mathit{Semimodule}(T, S) \triangleq \\
&\qquad \mathit{AdditiveMonoid}(T) \\
&\quad \wedge\ \mathit{CommutativeSemiring}(S) \\
&\quad \wedge\ \cdot : S \times T \to T
\end{aligned}
$$

$$
\begin{array}{ll}
\wedge & (\forall \alpha, \beta \in S)(\forall a, b \in T) \\
& \quad \alpha \cdot (\beta \cdot a) = (\alpha \cdot \beta) \cdot a \\
& \quad (\alpha + \beta) \cdot a = \alpha \cdot a + \beta \cdot a \\
& \quad \alpha \cdot (a + b) = \alpha \cdot a + \alpha \cdot b \\
& \quad 1 \cdot a = a
\end{array}
$$

Если определена концепция $Semimodule(T, S)$, мы говорим, что T — полумодуль *над* S. Мы заимствуем терминологию из теории векторных пространств и называем элементы T *векторами* и элементы S — *скалярами*. Например, многочлены с неотрицательными целочисленными коэффициентами составляют полумодуль над неотрицательными целыми числами.

Теорема 5.1. $AdditiveMonoid(T) \Rightarrow Semimodule(T, \mathbb{N})$, где скалярное умножение определено как $n \cdot x = \underbrace{x + \cdots + x}_{n \text{ раз}}$.

Доказательство. Это следует тривиально из определения скалярного умножения, наряду с ассоциативностью и коммутативностью операции моноида. Например,

$$
\begin{aligned}
n \cdot a + n \cdot b &= (a + \cdots + a) + (b + \cdots + b) \\
&= (a + b) + \cdots + (a + b) \\
&= n \cdot (a + b)
\end{aligned}
$$

□

Использование процедуры power из главы 3 позволяет нам реализовать умножение на целое число за $\log_2 n$ шагов.

Усиление этих требований путем замены аддитивного моноида аддитивной группой и замены полукольца кольцом приводит к преобразованию полумодуля в модуль:

$$
\begin{array}{ll}
Module(T, S) \triangleq & \\
& Semimodule(T, S) \\
\wedge & AdditiveGroup(T) \\
\wedge & Ring(S)
\end{array}
$$

Лемма 5.4. Каждая аддитивная группа — это модуль над целыми числами с соответственно определенным скалярным умножением.

Компьютерные типы часто являются частичными моделями концепций. Модель называется *частичной*, когда ее операции удовлетворяют аксиомам там, где они определены, но последние определены не везде. Например, результат конкатенации строк может оказаться непредставимым из-за ограничений по памяти, но конкатенация ассоциативна всякий раз, когда она определена.

5.2. Упорядоченные алгебраические структуры

Если на элементах структуры определено полное упорядочение, так, что это упорядочение соответствует алгебраическим свойствам структуры, оно является *естественным полным упорядочением* для структуры:

$$\begin{aligned}
&\mathit{OrderedAdditiveSemigroup}(\mathsf{T}) \triangleq \\
&\qquad \mathit{AdditiveSemigroup}(\mathsf{T}) \\
&\quad \wedge\ \mathit{TotallyOrdered}(\mathsf{T}) \\
&\quad \wedge\ (\forall a, b, c \in \mathsf{T})\ a < b \Rightarrow a + c < b + c \\
&\mathit{OrderedAdditiveMonoid}(\mathsf{T}) \triangleq \\
&\qquad \mathit{OrderedAdditiveSemigroup}(\mathsf{T}) \\
&\quad \wedge\ \mathit{AdditiveMonoid}(\mathsf{T}) \\
&\mathit{OrderedAdditiveGroup}(\mathsf{T}) \triangleq \\
&\qquad \mathit{OrderedAdditiveMonoid}(\mathsf{T}) \\
&\quad \wedge\ \mathit{AdditiveGroup}(\mathsf{T})
\end{aligned}$$

Лемма 5.5. В упорядоченной аддитивной полугруппе $a < b \wedge c < d \Rightarrow a + c < b + d$.

Лемма 5.6. В упорядоченном аддитивном моноиде, рассматриваемом как полумодуль над натуральными числами, $a > 0 \wedge n > 0 \Rightarrow na > 0$.

Лемма 5.7. В упорядоченной аддитивной группе $a < b \Rightarrow -b < -a$.

Полное упорядочение и отрицание позволяют определить абсолютное значение:

```
template<typename T>
    requires(OrderedAdditiveGroup(T))
T abs(const T& a)
{
    if (a < T(0)) return -a;
    else          return  a;
}
```

Следующая лемма отражает важное свойство abs.

Лемма 5.8. В упорядоченной аддитивной группе $a < 0 \Rightarrow 0 < -a$.

Мы используем обозначение $|a|$ для абсолютного значения a. Абсолютное значение удовлетворяет следующим свойствам.

Лемма 5.9.
$$\begin{aligned}
|a - b| &= |b - a| \\
|a + b| &\leqslant |a| + |b| \\
|a - b| &\geqslant |a| - |b| \\
|a| = 0 &\Rightarrow a = 0 \\
a \neq 0 &\Rightarrow |a| > 0
\end{aligned}$$

5.3. Остаток

Выше было показано, что повторяющееся сложение в аддитивном моноиде по индукции приводит к умножению на неотрицательное целое число. В аддитивной группе можно осуществить обращение этого алгоритма, получая деление путем повторного вычитания на элементах в форме $a = nb$, где b делит a. Чтобы распространить это определение на деление с остатком для произвольной пары элементов, мы должны воспользоваться упорядочением. Упорядочение позволяет завершить работу алгоритма после того, как дальнейшее вычитание станет невозможным. Как будет показано ниже, это делает возможным выполнение алгоритма за логарифмическое количество шагов. Операция вычитания не должна быть определена везде; достаточно иметь частичное вычитание, называемое *сокращением*, где $a - b$ определено, только если b не превышает a:

$$\begin{aligned} &\mathit{CancellableMonoid}(\mathsf{T}) \triangleq \\ &\qquad \mathit{OrderedAdditiveMonoid}(\mathsf{T}) \\ &\quad \wedge\ - : \mathsf{T} \times \mathsf{T} \to \mathsf{T} \\ &\quad \wedge\ (\forall a, b \in \mathsf{T})\, b \leqslant a \Rightarrow a - b \text{ is defined} \wedge (a - b) + b = a \end{aligned}$$

Мы записываем аксиому как $(a - b) + b = a$ вместо $(a + b) - b = a$, чтобы избежать переполнения в частичных моделях *CancellableMonoid*:

```
template<typename T>
    requires(CancellableMonoid(T))
T slow_remainder(T a, T b)
{
    // Предусловие: a ⩾ 0 ∧ b > 0
    while (b <= a) a = a - b;
    return a;
}
```

Концепция *CancellableMonoid* недостаточно сильна, чтобы с ее помощью можно было проверять, завершается ли slow_remainder. Например, slow_remainder не всегда завершается применительно к многочленам с целочисленными коэффициентами, упорядоченными лексикографически.

Упражнение 5.1. Приведите пример двух многочленов с целочисленными коэффициентами, для которых работа этого алгоритма не завершается.

Чтобы можно было гарантировать, что работа алгоритм завершится, нам требуется еще одно свойство, называемое *аксиомой Архимеда*[1]:

[1] "... разность, на которую бóльшая из (двух) неравных площадей превышает меньшую, путем сложения с самой собой может быть сделана большей по величине любой заданной конечной площади". См. [Heath 1912, стр. 234].

$$\begin{aligned}&\mathit{ArchimedeanMonoid}(\mathsf{T}) \triangleq\\ &\quad \mathit{CancellableMonoid}(\mathsf{T})\\ &\wedge\ (\forall a, b \in \mathsf{T})\,(a \geqslant 0 \wedge b > 0) \Rightarrow \mathsf{slow\_remainder}(a, b) \text{ terminates}\\ &\wedge\ \mathsf{QuotientType} : \mathit{ArchimedeanMonoid} \to \mathit{Integer}\end{aligned}$$

Следует отметить, что применение аксиомы завершения алгоритма вполне обосновано; в этом случае она эквивалентна следующему:

$$(\exists n \in \mathsf{QuotientType}(\mathsf{T}))\, a - n \cdot b < b$$

Аксиома Архимеда обычно приводится в формулировке "существует такое целое число n, что $a < n \cdot b$", но наша версия применима для частичных архимедовых моноидов, в условиях, когда может произойти переполнение $n \cdot b$. Функция типа QuotientType возвращает тип, достаточно большой для представления количества итераций, выполненных slow_remainder.

Лемма 5.10. Следующие примеры относятся к архимедовым моноидам: целые числа, рациональные числа, двоичные дроби $\{\frac{n}{2^k}\}$, троичные дроби $\{\frac{n}{3^k}\}$ и вещественные числа.

Мы можем тривиально приспособить код slow_remainder для возврата частного:

```
template<typename T>
    requires(ArchimedeanMonoid(T))
QuotientType(T) slow_quotient(T a, T b)
{
    // Предусловие: a ⩾ 0 ∧ b > 0
    QuotientType(T) n(0);
    while (b <= a) {
        a = a - b;
        n = successor(n);
    }
    return n;
}
```

Повторное удвоение приводит к алгоритму power логарифмической сложности. Связанный с этим алгоритм возможен для остатка[2]. Выведем выражение для остатка u от деления a на b в терминах остатка v от деления a на $2b$:

$$a = n(2b) + v$$

Поскольку остаток v должен быть меньше, чем делитель $2b$, из этого следует, что

$$u = \begin{cases} v & \text{if } v < b \\ v - b & \text{if } v \geqslant b \end{cases}$$

[2]В Древнем Египте этот алгоритм использовался для деления с остатком, а степенной алгоритм — для умножения. См. [Robins and Shute 1987, стр. 18].

Это приводит к следующей рекурсивной процедуре:

```
template<typename T>
    requires(ArchimedeanMonoid(T))
T remainder_recursive(T a, T b)
{
    // Предусловие: a ⩾ b > 0
    if (a - b >= b) {
        a = remainder_recursive(a, b + b);
        if (a < b) return a;
    }
    return a - b;
}
```

Проверка $a - b \geqslant b$, а не $a \geqslant b + b$ позволяет избежать переполнения $b + b$.

```
template<typename T>
    requires(ArchimedeanMonoid(T))
T remainder_nonnegative(T a, T b)
{
    // Предусловие: a ⩾ 0 ∧ b > 0
    if (a < b) return a;
    return remainder_recursive(a, b);
}
```

Упражнение 5.2. Проанализируйте сложность remainder_nonnegative.

В [Floyd and Knuth 1990] Floyd and Knuth [1990] приведен алгоритм для остатка на архимедовых моноидах с постоянной потребностью в пространстве, который выполняет примерно на 31% больше операций по сравнению с remainder_nonnegative, но при условии, что возможно деление на два, существует алгоритм, не требующий увеличения количества операций[3]. Очевидно, что такая возможность обнаруживается во многих ситуациях. Например, общая задача k-секции угла с помощью линейки и циркуля не может быть решена, но задача бисекции тривиальна.

$$\begin{aligned} &\mathit{HalvableMonoid}(\mathsf{T}) \triangleq \\ &\quad \mathit{ArchimedeanMonoid}(\mathsf{T}) \\ &\quad \wedge\ \mathsf{half} : \mathsf{T} \to \mathsf{T} \\ &\quad \wedge\ (\forall a, b \in \mathsf{T})\,(b > 0 \wedge a = b + b) \Rightarrow \mathsf{half}(a) = b \end{aligned}$$

Отметим, что процедура half должна быть определена только для “четных” элементов.

```
template<typename T>
    requires(HalvableMonoid(T))
T remainder_nonnegative_iterative(T a, T b)
```

[3] В [Dijkstra 1972, стр. 13] утверждается, что этот алгоритм принадлежит Н. Г. де Брюйну.

```
{
    // Предусловие: a ⩾ 0 ∧ b > 0
    if (a < b) return a;
    T c = largest_doubling(a, b);
    a = a - c;
    while (c != b) {
        c = half(c);
        if (c <= a) a = a - c;
    }
    return a;
}
```

где largest_doubling определена следующей процедурой:

```
template<typename T>
    requires(ArchimedeanMonoid(T))
T largest_doubling(T a, T b)
{
    // Предусловие: a ⩾ b > 0
    while (b <= a - b) b = b + b;
    return b;
}
```

Правильность remainder_nonnegative_iterative зависит от следующей леммы.

Лемма 5.11. Результат удвоения положительного элемента делимого на два моноида k раз может быть подвергнут делению на два k раз.

Процедура remainder_nonnegative нам потребовалась бы, только если бы рассматриваемый архимедов моноид не был делимым на два. Во всех приведенных нами примерах, включая линейные сегменты в евклидовой геометрии, рациональные числа, двоичные и троичные дроби, моноиды являются делимыми на два.

Проект 5.1. Есть ли полезные модели архимедовых моноидов, не являющихся моноидами, делимыми на два?

5.4. Наибольший общий делитель

Для $a \geqslant 0$ и $b > 0$ в архимедовом моноиде T определим *делимость* следующим образом:

$$b \text{ делит } a \Leftrightarrow (\exists n \in \mathsf{QuotientType}(T))\, a = nb$$

Лемма 5.12. В архимедовом моноиде T с положительными x, a, b:

- b делит $a \Leftrightarrow \mathsf{remainder\_nonnegative}(a, b) = 0$

- b делит $a \Rightarrow b \leqslant a$
- $a > b \wedge x$ делит $a \wedge x$ делит $b \Rightarrow x$ делит $(a - b)$
- x делит $a \wedge x$ делит $b \Rightarrow x$ делит $\mathsf{remainder\_nonnegative}(a, b)$

Наибольший общий делитель a и b, обозначаемый как $\gcd(a, b)$, является делителем a и b, делимым на любой другой общий делитель a и b[4].

Лемма 5.13. В архимедовом моноиде имеет место следующее при положительных x, a, b:

- $\gcd$ коммутативна
- $\gcd$ ассоциативна
- x делит $a \wedge x$ делит $b \Rightarrow x \leqslant \gcd(a, b)$
- $\gcd(a, b)$ уникален
- $\gcd(a, a) = a$
- $a > b \Rightarrow \gcd(a, b) = \gcd(a - b, b)$

Из приведенных выше лемм непосредственно следует, что если следующий алгоритм завершает свою работу, то возвращает $\gcd$ своих аргументов[5]:

```
template<typename T>
    requires(ArchimedeanMonoid(T))
T subtractive_gcd_nonzero(T a, T b)
{
    // Предусловие: a > 0 ∧ b > 0
    while (true) {
        if (b < a)      a = a - b;
        else if (a < b) b = b - a;
        else            return a;
    }
}
```

Лемма 5.14. Он всегда завершается для целых и рациональных чисел.

Есть и такие типы, для которых алгоритм не всегда завершается. В частности, он не всегда завершается для вещественных чисел; особо следует отметить, что он не завершается для входных данных $\sqrt{2}$ и 1. Доказательство этого факта зависит от следующих двух лемм:

Лемма 5.15. $\gcd(\frac{a}{\gcd(a,b)}, \frac{b}{\gcd(a,b)}) = 1$

[4]Хотя это определение применяется для архимедовых моноидов, оно не зависит от упорядочения и может быть распространено на другие структуры с отношениями делимости, такие как кольца.

[5]Он известен как алгоритм Евклида [Heath 1925, том 3, стр. 14–22].

Лемма 5.16. Если квадрат целого числа n является четным, n четно.

Теорема 5.2. subtractive_gcd_nonzero$(\sqrt{2}, 1)$ не завершается.

Доказательство. Предположим, что subtractive_gcd_nonzero$(\sqrt{2}, 1)$ завершается, возвращая d. Пусть $m = \frac{\sqrt{2}}{d}$ и $n = \frac{1}{d}$; согласно лемме 5.15, m и n не имеют общих множителей больше 1. $\frac{m}{n} = \frac{\sqrt{2}}{1} = \sqrt{2}$, поэтому $m^2 = 2n^2$; m — четно; для некоторого целого числа u, $m = 2u$. $4u^2 = 2n^2$, поэтому $n^2 = 2u^2$; n — четно. И m, и n делятся на 2; налицо противоречие[6]. □

Евклидов моноид — это архимедов моноид, где subtractive_gcd_nonzero всегда завершается:

$$\begin{aligned} &\mathit{EuclideanMonoid}(\mathsf{T}) \triangleq \\ &\qquad \mathit{ArchimedeanMonoid}(\mathsf{T}) \\ &\quad \wedge\ (\forall a, b \in \mathsf{T})\,(a > 0 \wedge b > 0) \Rightarrow \text{subtractive\_gcd\_nonzero}(a, b) \text{ завершается} \end{aligned}$$

Лемма 5.17. Каждый архимедов моноид с наименьшим положительным элементом является евклидовым.

Лемма 5.18. Рациональные числа — это евклидов моноид.

Несложно распространить subtractive_gcd_nonzero на случай, в котором один из аргументов этой процедуры — нуль, поскольку любой аргумент $b \neq 0$ является делителем для нуля моноида:

```
template<typename T>
    requires(EuclideanMonoid(T))
T subtractive_gcd(T a, T b)
{
    // Предусловие: a ⩾ 0 ∧ b ⩾ 0 ∧ ¬(a = 0 ∧ b = 0)
    while (true) {
        if (b == T(0)) return a;
        while (b <= a) a = a - b;
        if (a == T(0)) return b;
        while (a <= b) b = b - a;
    }
}
```

Каждая из внутренних инструкций `while` в процедуре subtractive_gcd эквивалентна вызову slow_remainder. Используя наш логарифмический алгоритм вычисления остатка, ускорим вычисления в том случае, когда a и b значительно отличаются по величине, продолжая полагаться только на примитивное вычитание на типе T:

[6] Доказательство несоизмеримости стороны и диагонали квадрата было одним из первых математических доказательств, полученных в Древней Греции. Аристотель упоминает его в *Первой аналитике* (I. 23) как канонический пример доказательства от противного *(reductio ad absurdum)*.

```
template<typename T>
    requires(EuclideanMonoid(T))
T fast_subtractive_gcd(T a, T b)
{
    // Предусловие: a ⩾ 0 ∧ b ⩾ 0 ∧ ¬(a = 0 ∧ b = 0)
    while (true) {
        if (b == T(0)) return a;
        a = remainder_nonnegative(a, b);
        if (a == T(0)) return b;
        b = remainder_nonnegative(b, a);
    }
}
```

Концепция евклидового моноида позволила нам получить более абстрактную версию оригинального алгоритма Евклида, который был основан на повторном вычитании.

5.5. Обобщение НОД

Мы можем использовать fast_subtractive_gcd с целыми числами, потому что они составляют евклидов моноид. Для целых чисел можно было бы также использовать тот же алгоритм со встроенной процедурой вычисления остатка вместо remainder_nonnegative. Кроме того, алгоритм является применимым в определенных неархимедовых доменах, при условии, что для них предусмотрена подходящая функция вычисления остатка. Например, стандартный алгоритм деления в столбик легко распространяется с десятичных целых чисел на многочлены над вещественными числами[7]. Используя такой остаток, мы можем вычислить НОД двух многочленов.

В абстрактной алгебре определено понятие евклидового кольца (известного также как евклидов домен), позволяющее обеспечить указанное применение алгоритма Евклида[8]. Однако достаточно выполнить требования к полукольцу:

$$
\begin{aligned}
&\mathit{EuclideanSemiring}(\mathsf{T}) \triangleq \\
&\qquad \mathit{CommutativeSemiring}(\mathsf{T}) \\
&\quad \wedge\ \mathsf{NormType} : \mathit{EuclideanSemiring} \to \mathit{Integer} \\
&\quad \wedge\ \mathsf{w} : \mathsf{T} \to \mathsf{NormType}(\mathsf{T}) \\
&\quad \wedge\ (\forall \mathfrak{a} \in \mathsf{T})\, \mathsf{w}(\mathfrak{a}) \geqslant 0 \\
&\quad \wedge\ (\forall \mathfrak{a} \in \mathsf{T})\, \mathsf{w}(\mathfrak{a}) = 0 \Leftrightarrow \mathfrak{a} = 0 \\
&\quad \wedge\ (\forall \mathfrak{a}, \mathfrak{b} \in \mathsf{T})\, \mathfrak{b} \neq 0 \Rightarrow \mathsf{w}(\mathfrak{a} \cdot \mathfrak{b}) \geqslant \mathsf{w}(\mathfrak{a}) \\
&\quad \wedge\ \mathsf{remainder} : \mathsf{T} \times \mathsf{T} \to \mathsf{T} \\
&\quad \wedge\ \mathsf{quotient} : \mathsf{T} \times \mathsf{T} \to \mathsf{T}
\end{aligned}
$$

[7] См. [Chrystal 1904, глава 5].

[8] См. [van der Waerden 1930, глава 3, раздел 18].

$\wedge\ (\forall a, b \in T)\, b \neq 0 \Rightarrow a = \mathsf{quotient}(a, b) \cdot b + \mathsf{remainder}(a, b)$
$\wedge\ (\forall a, b \in T)\, b \neq 0 \Rightarrow \mathsf{w}(\mathsf{remainder}(a, b)) < \mathsf{w}(b)$

Функция w называется *евклидовой функцией*.

Лемма 5.19. В евклидовом полукольце $a \cdot b = 0 \Rightarrow a = 0 \vee b = 0$.

```
template<typename T>
    requires(EuclideanSemiring(T))
T gcd(T a, T b)
{
    // Предусловие: ¬(a = 0 ∧ b = 0)
    while (true) {
        if (b == T(0)) return a;
        a = remainder(a, b);
        if (a == T(0)) return b;
        b = remainder(b, a);
    }
}
```

Необходимо учитывать, что вместо использования remainder_nonnegative мы используем функцию remainder, определяемую типом. Завершение гарантируется тем фактом, что w уменьшается после каждого применения remainder.

Лемма 5.20. gcd завершается на евклидовом полукольце.

В евклидовом полукольце quotient возвращает элемент полукольца. Это препятствует его использованию по первоначальному назначению алгоритма Евклида: определение общей меры любых двух соизмеримых количеств. Например, $\gcd(\frac{1}{2}, \frac{3}{4}) = \frac{1}{4}$. Мы можем унифицировать исходную и современную трактовку с помощью концепции *евклидова полумодуля*, который позволяет использовать quotient для возврата другого типа и принять условие завершения gcd как аксиому:

$EuclideanSemimodule(T, S) \triangleq$
$\quad Semimodule(T, S)$
$\wedge\ \mathsf{remainder} : T \times T \to T$
$\wedge\ \mathsf{quotient} : T \times T \to S$
$\wedge\ (\forall a, b \in T)\, b \neq 0 \Rightarrow a = \mathsf{quotient}(a, b) \cdot b + \mathsf{remainder}(a, b)$
$\wedge\ (\forall a, b \in T)\, (a \neq 0 \vee b \neq 0) \Rightarrow \mathsf{gcd}(a, b)$ terminates

где процедура gcd определена следующим образом:

```
template<typename T, typename S>
    requires(EuclideanSemimodule(T, S))
T gcd(T a, T b)
{
    // Предусловие: ¬(a = 0 ∧ b = 0)
    while (true) {
```

```
        if (b == T(0)) return a;
        a = remainder(a, b);
        if (a == T(0)) return b;
        b = remainder(b, a);
    }
}
```

Каждое коммутативное полукольцо — это полумодуль над самим собой, поэтому данный алгоритм может использоваться, даже если quotient возвращает тот же тип, как в случае с многочленами над вещественными числами.

5.6. Алгоритм gcd по Штейну

В 1961 году Джозеф Штейн открыл новый алгоритм gcd для целых чисел, который часто оказывается более быстродействующим, чем алгоритм Евклида [Stein 1967]. Алгоритм Штейна зависит от таких двух известных свойств:

$$\gcd(a, b) = \gcd(b, a)$$
$$\gcd(a, a) = a$$

вместе со следующими дополнительными свойствами, которые выполняются для всех $a > b > 0$:

$$\gcd(2a, 2b) = 2\gcd(a, b)$$
$$\gcd(2a, 2b + 1) = \gcd(a, 2b + 1)$$
$$\gcd(2a + 1, 2b) = \gcd(2a + 1, b)$$
$$\gcd(2a + 1, 2b + 1) = \gcd(2b + 1, a - b)$$

Упражнение 5.3. Реализуйте процедуру gcd по Штейну для целых чисел и докажите, что она завершается.

Разумеется, может показаться, что процедура gcd по Штейну зависит от двоичного представления целых чисел, но догадка, согласно которой 2 является наименьшим простым целым числом, позволяет обобщать эту процедуру на другие домены с использованием наименьших простых чисел в этих доменах; в качестве примеров можно указать одночлен x для многочленов[9] или $1+i$ для комплексных целых чисел[10]. Процедура gcd по Штейну могла использоваться в кольцах, которые не являются евклидовыми[11].

Проект 5.2. Найдите правильную общую трактовку для gcd по Штейну.

[9] См. [Knuth 1997, упражнение 4.6.1.6 (стр. 435) и решение (стр. 673)].
[10] См. [Weilert 2000].
[11] См. [Agarwal and Frandsen 2004].

5.7. Частное

Ход обоснования быстрой процедуры вычисления частного и остатка полностью повторяет приведенное нами ранее обоснование быстрой процедуры вычисления остатка. Мы выводим выражение для частного $\mathfrak{m}$ и остатка $\mathfrak{u}$ из выражения для деления $\mathfrak{a}$ на $\mathfrak{b}$ в терминах частного $\mathfrak{n}$ и остатка ν от деления $\mathfrak{a}$ на $2\mathfrak{b}$:

$$\mathfrak{a} = \mathfrak{n}(2\mathfrak{b}) + \nu$$

Поскольку остаток ν должен быть меньше, чем делитель $2\mathfrak{b}$, из этого следует, что

$$\mathfrak{u} = \begin{cases} \nu & \text{if } \nu < \mathfrak{b} \\ \nu - \mathfrak{b} & \text{if } \nu \geqslant \mathfrak{b} \end{cases}$$

и

$$\mathfrak{m} = \begin{cases} 2\mathfrak{n} & \text{if } \nu < \mathfrak{b} \\ 2\mathfrak{n} + 1 & \text{if } \nu \geqslant \mathfrak{b} \end{cases}$$

Это приводит к получению следующего кода:

```
template<typename T>
    requires(ArchimedeanMonoid(T))
pair<QuotientType(T), T>
quotient_remainder_nonnegative(T a, T b)
{
    // Предусловие: a ⩾ 0 ∧ b > 0
    typedef QuotientType(T) N;
    if (a < b) return pair<N, T>(N(0), a);
    if (a - b < b) return pair<N, T>(N(1), a - b);
    pair<N, T> q = quotient_remainder_nonnegative(a, b + b);
    N m = twice(q.m0);
    a = q.m1;
    if (a < b) return pair<N, T>(m, a);
    else       return pair<N, T>(successor(m), a - b);
}
```

Если применимо “деление на два”, мы получаем следующее:

```
template<typename T>
    requires(HalvableMonoid(T))
pair<QuotientType(T), T>
quotient_remainder_nonnegative_iterative(T a, T b)
{
    // Предусловие: a ⩾ 0 ∧ b > 0
    typedef QuotientType(T) N;
    if (a < b) return pair<N, T>(N(0), a);
    T c = largest_doubling(a, b);
```

```
    a = a - c;
    N n(1);
    while (c != b) {
        n = twice(n);
        c = half(c);
        if (c <= a) {
            a = a - c;
            n = successor(n);
        }
    }
    return pair<N, T>(n, a);
}
```

5.8. Частное и остаток для отрицательных величин

В определениях частного и остатка, используемых во многих компьютерных процессорах и языках программирования, понятие отрицательной величины интерпретируется неправильно. Расширение наших определений для архимедова моноида на архимедову группу T должно удовлетворять следующим свойствам, где $b \neq 0$:

$$a = \mathsf{quotient}(a, b) \cdot b + \mathsf{remainder}(a, b)$$
$$|\mathsf{remainder}(a, b)| < |b|$$
$$\mathsf{remainder}(a + b, b) = \mathsf{remainder}(a - b, b) = \mathsf{remainder}(a, b)$$

Последнее свойство эквивалентно классическому математическому определению конгруэнтности[12]. В книгах по теории чисел обычно предполагается, что $b > 0$, но мы можем, не вступая в противоречие, распространить remainder на $b < 0$. Эти требования не удовлетворяются в реализациях, которые предусматривают усечение частного в сторону нуля, приводя таким образом к нарушению нашего третьего требования[13]. Усечение не только нарушает третье требование, но и является худшим способом округления, поскольку при его применении ноль возвращается вдвое чаще чем любое другое число, что приводит к созданию неоднородного распределения.

При наличии процедуры вычисления остатка rem и процедуры вычисления остатка и частного $\mathrm{quo\_rem}$, удовлетворяющих нашим трем требованиям для

[12] "Если два числа a и b имеют один и тот же остаток r относительно одного и того же модуля k, то они называются *конгруэнтными* относительно модуля k (согласно Гауссу)" [Dirichlet 1863].

[13] Превосходное обсуждение темы частного и остатка см. в [Boute 1992]. Бут обозначает два приемлемых расширения как E и F; мы следуем Кнуту и предпочитаем то, которое Бут называет F.

неотрицательных входных данных, мы можем написать процедуры сопряжения, которые дают правильные результаты для положительных или отрицательных входных данных. Эти процедуры сопряжения будут применяться на *архимедовой группе*:

$$ArchimedeanGroup(\mathsf{T}) \triangleq ArchimedeanMonoid(\mathsf{T}) \wedge AdditiveGroup(\mathsf{T})$$

```
template<typename Op>
    requires(BinaryOperation(Op) &&
        ArchimedeanGroup(Domain(Op)))
Domain(Op) remainder(Domain(Op) a, Domain(Op) b, Op rem)
{
    // Предусловие: b ≠ 0
    typedef Domain(Op) T;
    T r;
    if (a < T(0))
        if (b < T(0)) {
            r = -rem(-a, -b);
        } else {
            r =  rem(-a,  b); if (r != T(0)) r = b - r;
        }
    else
        if (b < T(0)) {
            r =  rem(a, -b);  if (r != T(0)) r = b + r;
        } else {
            r =  rem(a,  b);
        }
    return r;
}

template<typename F>
    requires(HomogeneousFunction(F) && Arity(F) == 2 &&
        ArchimedeanGroup(Domain(F)) &&
        Codomain(F) == pair<QuotientType(Domain(F)),
                            Domain(F)>)
pair<QuotientType(Domain(F)), Domain(F)>
quotient_remainder(Domain(F) a, Domain(F) b, F quo_rem)
{
    // Предусловие: b ≠ 0
    typedef Domain(F) T;
    pair<QuotientType(T), T> q_r;
    if (a < T(0)) {
        if (b < T(0)) {
            q_r = quo_rem(-a, -b); q_r.m1 = -q_r.m1;
        } else {
            q_r = quo_rem(-a,  b);
```

```
            if (q_r.m1 != T(0)) {
                q_r.m1 = b - q_r.m1; q_r.m0 = successor(q_r.m0);
            }
            q_r.m0 = -q_r.m0;
        }
    } else {
        if (b < T(0)) {
            q_r = quo_rem( a, -b);
            if (q_r.m1 != T(0)) {
                q_r.m1 = b + q_r.m1; q_r.m0 = successor(q_r.m0);
            }
            q_r.m0 = -q_r.m0;
        }
        else
            q_r = quo_rem( a,  b);
    }
    return q_r;
}
```

Лемма 5.21. remainder и quotient_remainder удовлетворяют нашим требованиям, если их функциональные параметры удовлетворяют требованиям для положительных аргументов.

5.9. Концепции и их модели

Мы использовали целочисленные типы начиная с главы 2, формально не определяя эту концепцию. Основываясь на упорядоченных алгебраических структурах, определенных ранее в этой главе, мы можем формализовать нашу трактовку целых чисел. Сначала определим *дискретное архимедово полукольцо*:

$$\begin{aligned}&\mathit{DiscreteArchimedeanSemiring}(\mathsf{T}) \triangleq\\ &\quad \mathit{CommutativeSemiring}(\mathsf{T})\\ &\quad \wedge\ \mathit{ArchimedeanMonoid}(\mathsf{T})\\ &\quad \wedge\ (\forall a, b, c \in \mathsf{T})\, a < b \wedge 0 < c \Rightarrow a \cdot c < b \cdot c\\ &\quad \wedge\ \neg(\exists a \in \mathsf{T})\, 0 < a < 1\end{aligned}$$

Дискретность относится к последнему свойству: между 0 и 1 нет элементов.

Дискретное архимедово полукольцо может иметь отрицательные элементы. Связанной концепцией, которая не предусматривает наличие отрицательных элементов, является следующая:

$$\begin{aligned}&\mathit{NonnegativeDiscreteArchimedeanSemiring}(\mathsf{T}) \triangleq\\ &\quad \mathit{DiscreteArchimedeanSemiring}(\mathsf{T})\\ &\quad \wedge\ (\forall a \in \mathsf{T})\, 0 \leqslant a\end{aligned}$$

В дискретном архимедовом полукольце отсутствуют аддитивные обратные операции; связанной концепцией с аддитивными обратными операциями является следующая:

$$\begin{aligned} &\mathit{DiscreteArchimedeanRing}(\mathsf{T}) \triangleq \\ &\quad \mathit{DiscreteArchimedeanSemiring}(\mathsf{T}) \\ &\quad \wedge\ \mathit{AdditiveGroup}(\mathsf{T}) \end{aligned}$$

Два типа, T и T', являются *изоморфными*, если возможно написать функции преобразования из T в T' и из T' в T, которые сохраняют процедуры и их аксиомы.

Концепция является *однозначной*, если изоморфны все удовлетворяющие ей типы. Концепция $\mathit{NonnegativeDiscreteArchimedeanSemiring}$ однозначна; удовлетворяющие ей типы изоморфны $\mathbb{N}$ — натуральным числам[14]. Концепция $\mathit{DiscreteArchimedeanRing}$ однозначна; удовлетворяющие ей типы изоморфны $\mathbb{Z}$ — целым числам. Как было показано выше, по мере добавления аксиом количество моделей в концепции уменьшается, поэтому она быстро достигает точки однозначности.

Изложение в настоящей главе развивается дедуктивно, от более общих концепций к более конкретным, в ходе чего количество представленных операций и аксиом увеличивается. Дедуктивный подход представляет в готовом виде таксономию концепций, а также связанных с ними теорем и алгоритмов. Но фактический процесс открытия новых фактов происходит индуктивно, начиная с конкретных моделей, таких как числа или вещественные числа, после чего исключаются операции и аксиомы в целях поиска самой слабой концепции, к которой можно применить интересные алгоритмы.

Определяя концепцию, мы должны проверить независимость и непротиворечивость ее аксиом и продемонстрировать ее полезность.

Суждение *независимо* от набора аксиом, если есть модель, в которой все аксиомы являются истинными, а это суждение — ложным. Например, ассоциативность и коммутативность независимы: конкатенация строк ассоциативна, но не коммутативна, тогда как среднее двух значений ($\frac{x+y}{2}$) коммутативно, но не ассоциативно. Суждение является *зависимым*, или *доказуемым* из набора аксиом, если может быть выведено из них.

Концепция является *непротиворечивой*, если она имеет модель. Продолжая наш пример, отметим, что сложение натуральных чисел ассоциативно и коммутативно. Концепция *противоречива*, если из ее аксиом можно вывести и суждение, и его отрицание. Иными словами, чтобы продемонстрировать непротиворечивость, мы создаем модель; чтобы продемонстрировать несогласованность, мы выводим противоречие.

[14] Мы следуем [Peano 1908, стр. 27] и включаем 0 в состав натуральных чисел.

Концепция *полезна*, если есть полезные алгоритмы, для которых она является наиболее абстрактным основанием. Например, параллельное приведение с изменением последовательности применимо к любой ассоциативной, коммутативной операции.

5.10. Компьютерные целочисленные типы

Компьютерные системы команд, как правило, предусматривают лишь частичные представления натуральных и целых чисел. Например, *ограниченный целочисленный двоичный тип без знака*, $\mathsf{U}_\mathtt{n}$, где $\mathtt{n} = 8, 16, 32, 64, \ldots$, — это целочисленный тип без знака, способный представлять значения в интервале $[0, 2^\mathtt{n})$; *ограниченный целочисленный двоичный тип со знаком*, $\mathsf{S}_\mathtt{n}$, где $\mathtt{n} = 8, 16, 32, 64, \ldots$, — это целочисленный тип со знаком, способный представлять значения в интервале $[-2^{\mathtt{n}-1}, 2^{\mathtt{n}-1})$. Хотя эти типы ограничены, типичные компьютерные команды предусматривают выполнение с ними полных операций, поскольку результаты кодируются как кортежи ограниченных значений.

Обычно существуют команды для ограниченных типов без знака, имеющие сигнатуры, подобные следующим:

$$\begin{aligned}
\mathsf{sum_extended} &: \mathsf{U}_\mathtt{n} \times \mathsf{U}_\mathtt{n} \times \mathsf{U}_1 \to \mathsf{U}_1 \times \mathsf{U}_\mathtt{n} \\
\mathsf{difference_extended} &: \mathsf{U}_\mathtt{n} \times \mathsf{U}_\mathtt{n} \times \mathsf{U}_1 \to \mathsf{U}_1 \times \mathsf{U}_\mathtt{n} \\
\mathsf{product_extended} &: \mathsf{U}_\mathtt{n} \times \mathsf{U}_\mathtt{n} \to \mathsf{U}_{2\mathtt{n}} \\
\mathsf{quotient_remainder_extended} &: \mathsf{U}_\mathtt{n} \times \mathsf{U}_\mathtt{n} \to \mathsf{U}_\mathtt{n} \times \mathsf{U}_\mathtt{n}
\end{aligned}$$

Следует отметить, что $\mathsf{U}_{2\mathtt{n}}$ можно представить как $\mathsf{U}_\mathtt{n} \times \mathsf{U}_\mathtt{n}$ (пара $\mathsf{U}_\mathtt{n}$). Языки программирования, которые обеспечивают полный доступ к этим аппаратным операциям, позволяют разрабатывать эффективные и абстрактные программные компоненты, основанные на использовании целочисленных типов.

Проект 5.3. Спроектируйте семейство концепций для ограниченных двоичных целых чисел без знака и со знаком. Системы команд для современных архитектур вычислительных систем должны предоставлять определенные функциональные возможности, для определения которых проводятся специальные исследования. Хорошее абстрактное описание таких систем команд представлено в `MMIX` [Knuth 2005].

5.11. Резюме

Мы можем объединять алгоритмы и математические структуры в единое целое, описывая алгоритмы в абстрактных терминах и подстраивая теории, чтобы они соответствовали алгоритмическим требованиям. В математических выкладках и алгоритмах, представленных в этой главе, в абстрактном виде переформулированы результаты, полученные больше двух тысяч лет тому назад.

Глава 6
Итераторы

В настоящей главе вводится концепция *итератора*: интерфейса между алгоритмами и последовательными структурами данных. Иерархия концепций итератора соответствует различным разновидностям последовательных обходов: однопроходный прямой, многопроходный прямой, двунаправленный и с произвольным доступом[1]. Мы исследуем разнообразие интерфейсов к общим алгоритмам, таких как линейный и дихотомический поиск. Ограниченные и счетные интервалы предоставляют гибкий способ определения интерфейсов для всевозможных разновидностей последовательного алгоритма.

6.1. Читаемость

Каждый объект имеет адрес: целочисленный индекс в машинной памяти. Адреса позволяют нам обращаться или изменять объекты и создавать широкое разнообразие структур данных, многие из которых опираются на тот факт, что адреса, по существу, являются целыми числами и позволяют выполнять с ними операции, подобные операциям с целыми числами.

Итераторы — это семейство концепций, которые абстрагируют различные аспекты применения адресов, предоставляя нам возможность разрабатывать алгоритмы, которые работают не только с адресами, но и с любыми подобными адресам объектами, удовлетворяющими минимальному набору требований. В главе 7 мы вводим еще более широкое концептуальное семейство: *координатные структуры*.

Предусмотрены две разновидности операций на итераторах: доступ к значениям или обход. Имеются три разновидности доступа: чтение, запись или и чтение, и запись. Существуют четыре разновидности линейного обхода: однопроходный прямой (входной поток), многопроходный прямой (односвязный

[1]Наша трактовка итераторов является дальнейшим уточнением таковой, приведенной в [Stepanov and Lee 1995], но отличается от нее в нескольких аспектах.

список), двунаправленный (двухсвязный список) и с произвольным доступом (массив).

В настоящей главе рассматривается первая разновидность доступа: читаемость, т. е. способность получить значение одного объекта, обозначенного другим объектом. Тип T является *читаемым*, если унарная функция source, определенная на нем, возвращает объект типа ValueType(T):

$$\begin{aligned} &\mathit{Readable}(\mathsf{T}) \triangleq \\ &\qquad \mathit{Regular}(\mathsf{T}) \\ &\quad \wedge\ \mathsf{ValueType} : \mathit{Readable} \to \mathit{Regular} \\ &\quad \wedge\ \mathsf{source} : \mathsf{T} \to \mathsf{ValueType}(\mathsf{T}) \end{aligned}$$

source используется только в контекстах, в которых необходимо значение; ее результат можно передать в процедуру по значению или по постоянной ссылке.

Не исключено существование объектов читаемого типа, на которых source не определена; функция source не обязана быть всеохватывающей. Эта концепция не предоставляет предикат области определения, который позволил бы выяснять, определена ли функция source для конкретного объекта. Например, если задан указатель на тип T, то невозможно определить, указывает ли он на объект, созданный допустимым способом. Допустимость использования source в алгоритме должна быть выводимой из предусловий.

Доступ к данным путем вызова source применительно к объекту читаемого типа является столь же быстродействующим, как и любой другой способ доступа к этим данным. В частности, мы предполагаем, что для объекта читаемого типа со значением типа T, находящегося в оперативной памяти, стоимость вызова source будет приблизительно равна стоимости разадресации обычного указателя на T. Как и в случае обычных указателей, может иметь место неоднородность вследствие иерархической организации памяти. Другими словами, нет необходимости хранить указатели вместо итераторов, чтобы ускорить работу алгоритма.

Было бы полезно распространить source на типы, объекты которых не указывают на другие объекты. Мы осуществим это, предусмотрев возврат функцией source ее аргумента в случае применения к объекту такого типа. Это позволяет программе определять свое требование к значению типа T таким образом, чтобы требование могло быть удовлетворено значением типа T, указателем на тип T или, вообще говоря, любым читаемым типом со значением типа T. Поэтому мы предполагаем, что если не определено иное, $\mathsf{ValueType}(\mathsf{T}) = \mathsf{T}$, и функция source возвращает объект, к которому она применена.

6.2. Итераторы

Для обхода требуется способность генерировать новые итераторы. Как было показано в главе 2, один из способов генерировать новые значения некоторого типа состоит в использовании преобразования. В то время как преобразования являются регулярными, для некоторых однопроходных алгоритмов не требуется регулярность обхода, а некоторые модели, такие как входные потоки, не обеспечивают регулярность обхода. Таким образом, для самой нестрогой концепции итератора требуются только *псевдотрансформации*[2] successor и функция типа DistanceType:

$Iterator(\mathsf{T}) \triangleq$
$\quad Regular(\mathsf{T})$
$\quad \wedge\ \mathsf{DistanceType} : Iterator \to Integer$
$\quad \wedge\ \mathsf{successor} : \mathsf{T} \to \mathsf{T}$
$\quad \wedge$ successor не обязательно является регулярной

DistanceType возвращает целочисленный тип, достаточно большой для измерения любой последовательности применений successor, допустимых для типа. Регулярность подразумевается по умолчанию, поэтому мы обязаны явно указать, что регулярность не является требованием к функции successor.

Как и в случае применения функции source к читаемым типам, функция successor не должна быть всеохватывающей; могут быть такие объекты типа итератора, на которых successor не определена. Эта концепция не предоставляет предикат области определения, который позволил бы выяснять, определена ли функция successor для конкретного объекта. Например, указатель в массиве не содержит сведения, указывающие на то, сколько раз может произойти его наращивание. Допустимость использования successor в алгоритме должна быть выводимой из предусловий.

Действие, соответствующее successor, определено следующим образом:

```
template<typename I>
    requires(Iterator(I))
void increment(I& x)
{
    // Предусловие: successor(x) определено
    x = successor(x);
}
```

Многие важные алгоритмы, такие как линейный поиск и копирование, являются *однопроходными*, т. е. применяют successor к значению каждого итератора лишь единожды. Поэтому они могут использоваться со входными потоками, и именно по этой причине мы опускаем требование к successor, со-

[2] Псевдотрансформация имеет сигнатуру трансформации, но не является регулярной.

гласно которому она должна быть регулярной: из равенства $i = j$ не следует $\mathsf{successor}(i) = \mathsf{successor}(j)$, даже если функция successor определена. Кроме того, после вызова $\mathsf{successor}(i)$ может оказаться, что итератор i и любой равный ему итератор больше не являются полностью сформированными. Итераторы остаются частично сформированными и могут быть уничтожены или использованы в левой части операции присваивания; к ним не должны применяться функции successor или source и операция сравнения на равенство, $=$.

Следует учитывать, что из $\mathsf{successor}(i) = \mathsf{successor}(j)$ не следует, что $i = j$. В качестве примера можно указать два односвязных списка, завершающихся нулем.

Итератор обеспечивает столь же быстрый линейный обход через всю коллекцию данных, как и любой другой способ обхода по этим данным.

Чтобы можно было моделировать *Iterator* с помощью целочисленного типа, этот тип должен иметь тип расстояния. Целочисленный тип без знака является своим собственным типом расстояния; типом расстояния для любого ограниченного двоичного целочисленного типа S_n со знаком служит соответствующий тип U_n без знака.

6.3. Интервалы

Предположим, что f — объект типа итератора, а n — объект соответствующего типа расстояния; мы должны иметь возможность определять алгоритмы, оперирующие в *нестрогом интервале* $[\![f, n)\!)$ итераторов n, начиная с f, с применением кода в форме

```
while (!zero(n)) { n = predecessor(n); ... f = successor(f); }
```

Это свойство обеспечивает такую итерацию:

property(I : *Iterator*)
weak_range : $I \times \mathsf{DistanceType}(I)$
$(f, n) \mapsto (\forall i \in \mathsf{DistanceType}(I))$
$(0 \leqslant i \leqslant n) \Rightarrow \mathsf{successor}^i(f)$ определено

Лемма 6.1. $0 \leqslant j \leqslant i \wedge \mathsf{weak\_range}(f, i) \Rightarrow \mathsf{weak\_range}(f, j)$

В нестрогом интервале можем продвигаться вплоть до его размера:

```
template<typename I>
    requires(Iterator(I))
I operator+(I f, DistanceType(I) n)
{
    // Предусловие: n ⩾ 0 ∧ weak_range(f, n)
    while (!zero(n)) {
        n = predecessor(n);
```

```
        f = successor(f);
    }
    return f;
}
```

Добавление следующей аксиомы гарантирует, что в интервале не будет циклов:

property(I : *Iterator*, N : *Integer*)
counted_range : I × N
$(f, n) \mapsto \mathsf{weak\_range}(f, n) \wedge$
$(\forall i, j \in \mathbb{N})\,(0 \leqslant i < j \leqslant n) \Rightarrow$
$\mathsf{successor}^i(f) \neq \mathsf{successor}^j(f)$

Предположим, что f и l — объекты типа итератора; мы должны иметь возможность определять алгоритмы, действующие на *ограниченном интервале* $[f, l)$ итераторов, начиная с f и ограничиваясь l, с применением кода в форме

```
while (f != l) { ... f = successor(f); }
```

Это свойство обеспечивает такую итерацию:

property(I : *Iterator*)
bounded_range : I × I
$(f, l) \mapsto (\exists k \in \mathsf{DistanceType}(I))\,\mathsf{counted\_range}(f, k) \wedge \mathsf{successor}^k(f) = l$

Структура итерации, использующей ограниченный интервал, заканчивается сразу после обнаружения l; поэтому, в отличие от нестрогого интервала, она не может иметь циклов.

В ограниченном интервале мы можем реализовать[3] частичное вычитание на итераторах:

```
template<typename I>
    requires(Iterator(I))
DistanceType(I) operator-(I l, I f)
{
    // Предусловие: bounded_range(f, l)
    DistanceType(I) n(0);
    while (f != l) {
        n = successor(n);
        f = successor(f);
    }
    return n;
}
```

Поскольку функция successor может оказаться нерегулярной, вычитание должно использоваться только в предусловиях или в ситуациях, в которых необходимо вычислить только размер ограниченного интервала.

[3]Обратите внимание на аналогию с функцией distance из главы 2.

Наши определения операций + и − между итераторами и целыми числами не противоречат их трактовке в математике, где + и − всегда определяются на одном и том же типе. Как и в математике, и операция + между итераторами и целыми числами, и операция − между итераторами определены индуктивно в терминах successor. В стандартном индуктивном определении сложения на натуральных числах используется функция successor[4]:

$$a + 0 = a$$
$$a + \mathsf{successor}(b) = \mathsf{successor}(a + b)$$

Наше итерационное определение $f + n$ для итераторов эквивалентно, даже притом, что f и n имеют различные типы. Как и в случае с натуральными числами, доказуемым по индукции является определенный вариант ассоциативности.

Лемма 6.2. $(f + n) + m = f + (n + m)$

В предусловиях мы должны определить принадлежность в пределах интервала. Мы заимствуем соглашения, касающиеся интервалов (см. приложение А), для введения *полуоткрытых* и *замкнутых* интервалов. Мы используем разные обозначения для нестрогих или счетных интервалов и для ограниченных интервалов.

Полуоткрытый нестрогий или *счетный* интервал $[\![f, n)\!)$, где $n \geqslant 0$ — целое число, обозначает последовательность итераторов $\{\mathsf{successor}^k(f) \mid 0 \leqslant k < n\}$. *Закрытый нестрогий* или *счетный* интервал $[\![f, n]\!]$, где $n \geqslant 0$ — целое число, обозначает последовательность итераторов $\{\mathsf{successor}^k(f) \mid 0 \leqslant k \leqslant n\}$.

Полуоткрытый ограниченный интервал $[f, l)$ эквивалентен полуоткрытому счетному интервалу $[\![f, l - f)\!)$. *Замкнутый* ограниченный интервал $[f, l]$ эквивалентен замкнутому счетному интервалу $[\![f, l - f]\!]$.

Размер интервала — это количество итераторов в последовательности, которую он обозначает.

Лемма 6.3. Функция successor определена для каждого итератора в полуоткрытом интервале и для каждого итератора, кроме последнего, в замкнутом интервале.

Если r — интервал, а i — итератор, указываем, что $i \in r$, если i — элемент соответствующего множества итераторов.

Лемма 6.4. Если $i \in [f, l)$, то $[f, i)$ и $[i, l)$ — ограниченные интервалы.

Пустые полуоткрытые интервалы обозначаются с помощью $[\![i, 0)\!)$ или $[i, i)$ для некоторого итератора i. Пустые замкнутые интервалы не существуют.

[4] Впервые представлена в [Grassmann 1861]; определение Грассмана было популяризировано в [Peano 1908].

Лемма 6.5. $\mathtt{i} \notin [\![\mathtt{i}, 0[\![\wedge \mathtt{i} \notin [\mathtt{i}, \mathtt{i})$

Лемма 6.6. Пустые интервалы не имеют ни первого, ни последнего элемента.

Полезно описать пустую последовательность итераторов, начинающуюся с определенного итератора. Например, допустим, что в дихотомическом поиске отыскивается последовательность итераторов, значения которых равны заданному значению. Эта последовательность пуста, если такие значения отсутствуют, но ее позиция определяется там, где они появились бы в случае их вставки.

Итератор $\mathtt{l}$ называется *пределом* полуоткрытого $[\mathtt{f}, \mathtt{l})$ ограниченного интервала. Итератор $\mathtt{f} + \mathtt{n}$ — предел полуоткрытого нестрогого интервала $[\![\mathtt{f}, \mathtt{n}[\![$. Необходимо учитывать, что пустой интервал имеет предел, даже притом, что он не имеет первого или последнего элемента.

Лемма 6.7. Размер полуоткрытого нестрогого интервала $[\![\mathtt{f}, \mathtt{n}[\![$ равен $\mathtt{n}$. Размер замкнутого нестрогого интервала $[\![\mathtt{f}, \mathtt{n}]\!]$ равен $\mathtt{n}+1$. Размер полуоткрытого ограниченного интервала $[\mathtt{f}, \mathtt{l})$ равен $\mathtt{l} - \mathtt{f}$. Размер замкнутого ограниченного интервала $[\mathtt{f}, \mathtt{l}]$ равен $(\mathtt{l} - \mathtt{f}) + 1$.

Если $\mathtt{i}$ и $\mathtt{j}$ — итераторы в счетном или ограниченном интервале, определяем отношение $\mathtt{i} \prec \mathtt{j}$ как означающее, что $\mathtt{i} \neq \mathtt{j} \wedge \mathsf{bounded\_range}(\mathtt{i}, \mathtt{j})$, иными словами, что одно или несколько применений successor приводят от $\mathtt{i}$ к $\mathtt{j}$. Отношение $\prec$ (“предшествует”) и соответствующее рефлексивное отношение $\preceq$ (“предшествует или равно”) используются в спецификациях, таких как пред- и постусловия алгоритмов. Для многих пар значений типа итератора отношение $\prec$ не определено, поэтому часто отсутствует эффективный способ написания кода, реализующего $\prec$. Например, не существует эффективного способа определения того, предшествует ли один узел другому в связной структуре; узлы могут даже не быть соединены.

6.4. Читаемые интервалы

Интервал итераторов, которые относятся к типу, моделирующему *Readable* и *Iterator*, является *читаемым*, если функция source определена на всех итераторах в интервале:

property(I : *Readable*)
  requires(Iterator(I))
readable_bounded_range : I × I
  $(\mathtt{f}, \mathtt{l}) \mapsto \mathsf{bounded\_range}(\mathtt{f}, \mathtt{l}) \wedge (\forall \mathtt{i} \in [\mathtt{f}, \mathtt{l}))\, \mathsf{source}(\mathtt{i})$ определено

Необходимо учитывать, что source не обязательно должна быть определена на пределе интервала. Кроме того, итератор может оказаться не полностью

сформированным после применения successor, поэтому не гарантируется, что source можно будет применить к итератору после получения его последующего элемента. Функции readable_weak_range и readable_counted_range определены подобным образом.

При наличии читаемого интервала появляется возможность применить процедуру к каждому значению в интервале:

```
template<typename I, typename Proc>
    requires(Readable(I) && Iterator(I) &&
        Procedure(Proc) && Arity(Proc) == 1 &&
        ValueType(I) == InputType(Proc, 0))
Proc for_each(I f, I l, Proc proc)
{
    // Предусловие: readable_bounded_range(f,l)
    while (f != l) {
        proc(source(f));
        f = successor(f);
    }
    return proc;
}
```

Мы возвращаем процедуру, поскольку она могла собрать полезные данные в течение обхода[5].

Реализуем линейный поиск с помощью следующей процедуры:

```
template<typename I>
    requires(Readable(I) && Iterator(I))
I find(I f, I l, const ValueType(I)& x)
{
    // Предусловие: readable_bounded_range(f,l)
    while (f != l && source(f) != x) f = successor(f);
    return f;
}
```

При этом либо возвращенный итератор равен пределу интервала, либо его значение равно x. Возврат предела указывает на неудачное завершение поиска. Поскольку количество результатов достигает $n + 1$ при поиске в интервале с размером n, предел в данном алгоритме и во многих других алгоритмах выполняет полезное назначение. Поиск, в котором применяется функция find, можно перезапустить, выполнив обход вслед за возвращенным итератором, а затем снова вызвав find.

Изменение оператора сравнения с x для использования равенства вместо неравенства позволяет получить функцию find_not.

[5] Таким образом может использоваться функциональный объект.

Мы можем обобщить поиск равного значения до поиска первого значения, удовлетворяющего унарному предикату:

```
template<typename I, typename P>
    requires(Readable(I) && Iterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I find_if(I f, I l, P p)
{
    // Предусловие: readable_bounded_range(f, l)
    while (f != l && !p(source(f))) f = successor(f);
    return f;
}
```

Применение предиката вместо его дополнения позволяет получить следующее: find_if_not.

Упражнение 6.1. Используйте find_if и find_if_not для реализации кванторных функций all, none, not_all и some, каждая из которых принимает ограниченный интервал и предикат.

Функция find и кванторные функции позволяют искать значения, удовлетворяющие условию; мы можем также подсчитывать количество удовлетворяющих значений:

```
template<typename I, typename P, typename J>
    requires(Readable(I) && Iterator(I) &&
        UnaryPredicate(P) && Iterator(J) &&
        ValueType(I) == Domain(P))
J count_if(I f, I l, P p, J j)
{
    // Предусловие: readable_bounded_range(f, l)
    while (f != l) {
        if (p(source(f))) j = successor(j);
        f = successor(f);
    }
    return j;
}
```

Явная передача j становится полезной, если сложение целого числа с j требует линейного времени. Тип J может быть любым целочисленным типом или типом итератора, включая I.

Упражнение 6.2. Реализуйте count_if, передавая соответствующий функциональный объект в for_each и извлекая результат накопления из возвращенного функционального объекта.

Естественный стандартный подход состоит в том, чтобы начинать отсчет с нуля и использовать тип расстояния для итераторов:

```
template<typename I, typename P>
    requires(Readable(I) && Iterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
DistanceType(I) count_if(I f, I l, P p) {
    // Предусловие: readable_bounded_range(f, l)
    return count_if(f, l, p, DistanceType(I)(0));
}
```

Замена предиката с проверкой равенства дает нам count; отрицание проверок позволяет получить count_not и count_if_not.

Обозначение $\sum_{i=0}^{n} a_i$ для суммы a_i часто обобщается на другие бинарные операции; например, $\prod_{i=0}^{n} a_i$ используется для произведений и $\bigwedge_{i=0}^{n} a_i$ — для конъюнкций. В любом случае операция является ассоциативной, а это означает, что группирование не важно. Кеннет Айверсон объединил эту систему обозначений языка программирования APL с *оператором приведения* /, который принимает бинарную операцию и последовательность и сводит элементы в единственный результат[6]. Например, `+/1 2 3` равняется 6.

Айверсон не ограничивает приведение ассоциативными операциями. Мы расширяем приведение Айверсона для работы с интервалами итераторов, но ограничиваем его *частично ассоциативными* операциями: если операция определена между смежными элементами, то может быть повторно связана:

property(Op : *BinaryOperation*)
partially_associative : Op
$op \mapsto (\forall a, b, c \in \mathrm{Domain}(op))$
Если $op(a, b)$ и $op(b, c)$ определены,
$op(op(a, b), c)$ и $op(a, op(b, c))$) определены
и равны.130g

В качестве примера операции, которая является частично, но не полностью ассоциативной, рассмотрим конкатенацию двух интервалов, $[f_0, l_0)$ и $[f_1, l_1)$, которая определена, только если $l_0 = f_1$.

Мы допускаем возможность применения унарной функции к каждому итератору до выполнения бинарной операции, что приводит к получению a_i из i. Произвольная частично ассоциативная операция может не иметь единицы, поэтому предусматриваем такую версию приведения, которая требует наличия непустого интервала:

```
template<typename I, typename Op, typename F>
    requires(Iterator(I) && BinaryOperation(Op) &&
        UnaryFunction(F) &&
        I == Domain(F) && Codomain(F) == Domain(Op))
Domain(Op) reduce_nonempty(I f, I l, Op op, F fun)
{
```

[6] См. [Iverson 1962].

```
    // Предусловие: bounded_range(f, l) ∧ f ≠ l
    // Предусловие: partially_associative(op)
    // Предусловие: (∀x ∈ [f, l)) fun(x) определено
    Domain(Op) r = fun(f);
    f = successor(f);
    while (f != l) {
        r = op(r, fun(f));
        f = successor(f);
    }
    return r;
}
```

Естественным стандартным значением для fun является source. Может быть передан единичный элемент для возврата применительно к пустому интервалу:

```
template<typename I, typename Op, typename F>
    requires(Iterator(I) && BinaryOperation(Op) &&
        UnaryFunction(F) &&
        I == Domain(F) && Codomain(F) == Domain(Op))
Domain(Op) reduce(I f, I l, Op op, F fun, const Domain(Op)& z)
{
    // Предусловие: bounded_range(f, l)
    // Предусловие: partially_associative(op)
    // Предусловие: (∀x ∈ [f, l)) fun(x) определено
    if (f == l) return z;
    return reduce_nonempty(f, l, op, fun);
}
```

Если операции, в которых участвует единичный элемент, являются медленными или требуют для реализации дополнительных вычислений, то становится полезной следующая процедура:

```
template<typename I, typename Op, typename F>
    requires(Iterator(I) && BinaryOperation(Op) &&
        UnaryFunction(F) &&
        I == Domain(F) && Codomain(F) == Domain(Op))
Domain(Op) reduce_nonzeroes(I f, I l,
                            Op op, F fun, const Domain(Op)& z)
{
    // Предусловие: bounded_range(f, l)
    // Предусловие: partially_associative(op)
    // Предусловие: (∀x ∈ [f, l)) fun(x) определено
    Domain(Op) x;
    do {
        if (f == l) return z;
        x = fun(f);
```

```
        f = successor(f);
    } while (x == z);
    while (f != l) {
        Domain(Op) y = fun(f);
        if (y != z) x = op(x, y);
        f = successor(f);
    }
    return x;
}
```

Алгоритмы, принимающие ограниченный интервал, имеют соответствующую версию, принимающую нестрогий или счетный интервал; однако должна быть возвращена дополнительная информация:

```
template<typename I, typename Proc>
    requires(Readable(I) && Iterator(I) &&
        Procedure(Proc) && Arity(Proc) == 1 &&
        ValueType(I) == InputType(Proc, 0))
pair<Proc, I> for_each_n(I f, DistanceType(I) n, Proc proc)
{
    // Предусловие: readable_weak_range(f, n)
    while (!zero(n)) {
        n = predecessor(n);
        proc(source(f));
        f = successor(f);
    }
    return pair<Proc, I>(proc, f);
}
```

Необходимо обеспечить возврат окончательного значения итератора, поскольку из отсутствия регулярности successor следует, что возможность его повторного вычисления может отсутствовать. Даже если итераторы являются таковыми, что функция successor для них регулярна, ее повторное вычисление может потребовать затрат времени, линейно зависящих от размера интервала.

```
template<typename I>
    requires(Readable(I) && Iterator(I))
pair<I, DistanceType(I)> find_n(I f, DistanceType(I) n,
                                const ValueType(I)& x)
{
    // Предусловие: readable_weak_range(f, n)
    while (!zero(n) && source(f) != x) {
        n = predecessor(n);
        f = successor(f);
    }
    return pair<I, DistanceType(I)>(f, n);
}
```

find_n возвращает окончательное значение итератора и количество, поскольку оба эти значения необходимы для перезапуска поиска.

Упражнение 6.3. Реализуйте разновидности алгоритмов, принимающие нестрогий интервал вместо ограниченного интервала, для всех версий find, кванторов, count и reduce.

Мы можем устранить одну из двух проверок в цикле find_if, если есть уверенность в том, что элемент в интервале удовлетворяет предикату; такой элемент называется *защитным*:

```
template<typename I, typename P>
    requires(Readable(I) && Iterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I find_if_unguarded(I f, P p) {
    // Предусловие: (∃l) readable_bounded_range(f, l) ∧ some(f, l, p)
    while (!p(source(f))) f = successor(f);
    return f;
    // Постусловие: p(source(f))
}
```

Применение предиката вместо его дополнения лежит в основе функции find_if_not_unguarded.

Если имеются два интервала с одним и тем же типом значений и отношение на этом типе значений, то можно обеспечить поиск несогласованной пары значений:

```
template<typename I0, typename I1, typename R>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) && Relation(R) &&
        ValueType(I0) == ValueType(I1) &&
        ValueType(I0) == Domain(R))
pair<I0, I1> find_mismatch(I0 f0, I0 l0, I1 f1, I1 l1, R r)
{
    // Предусловие: readable_bounded_range(f0, l0)
    // Предусловие: readable_bounded_range(f1, l1)
    while (f0 != l0 && f1 != l1 && r(source(f0), source(f1))) {
        f0 = successor(f0);
        f1 = successor(f1);
    }
    return pair<I0, I1>(f0, f1);
}
```

Упражнение 6.4. Сформулируйте постусловие для find_mismatch и объясните, почему происходит возврат окончательных значений обоих итераторов.

Естественным стандартным значением для отношения в find_mismatch является равенство на типе значений.

Упражнение 6.5. Разработайте разновидности функции find_mismatch для всех четырех комбинаций счетных и ограниченных интервалов.

Иногда бывает важно найти рассогласование не между интервалами, а между смежными элементами одного и того же интервала:

```
template<typename I, typename R>
    requires(Readable(I) && Iterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
I find_adjacent_mismatch(I f, I l, R r) {
    // Предусловие: readable_bounded_range(f, l)
    if (f == l) return l;
    ValueType(I) x = source(f);
    f = successor(f);
    while (f != l && r(x, source(f))) {
        x = source(f);
        f = successor(f);
    }
    return f;
}
```

Мы должны скопировать предыдущее значение, поскольку не имеем возможности применить source к итератору после применения к нему successor. Из нестрогих требований к *Iterator* также следует, что возврат первого итератора в несогласованной паре может привести к возврату не полностью сформированного значения.

6.5. Увеличение интервалов

Если дано некоторое отношение на типе значений определенного итератора, то интервал, охватывающий этот тип итератора, называется *сохраняющим отношение*, если отношение соблюдается для каждой смежной пары значений в интервале. Иными словами, find_adjacent_mismatch возвращает предел, будучи вызванной с этим интервалом и отношением:

```
template<typename I, typename R>
    requires(Readable(I) && Iterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
bool relation_preserving(I f, I l, R r)
{
    // Предусловие: readable_bounded_range(f, l)
    return l == find_adjacent_mismatch(f, l, r);
}
```

Если дано нестрогое упорядочение r, мы говорим, что интервал является r-*возрастающим*, если он сохраняет отношение применительно к дополнению

обращения r. Если дано нестрогое упорядочение r, мы говорим, что интервал является *строго* r-*возрастающим*, если он сохраняет отношение применительно к r[7]. Несложно проверить, что интервал — строго возрастающий:

```
template<typename I, typename R>
    requires(Readable(I) && Iterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
bool strictly_increasing_range(I f, I l, R r)
{
    // Предусловие: readable_bounded_range(f, l) ∧ weak_ordering(r)
    return relation_preserving(f, l, r);
}
```

С помощью функционального объекта мы можем проверить, что интервал является возрастающим:

```
template<typename R>
    requires(Relation(R))
struct complement_of_converse
{
    typedef Domain(R) T;
    R r;
    complement_of_converse(const R& r) : r(r) { }
    bool operator()(const T& a, const T& b)
    {
        return !r(b, a);
    }
};

template<typename I, typename R>
    requires(Readable(I) && Iterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
bool increasing_range(I f, I l, R r)
{
    // Предусловие: readable_bounded_range(f, l) ∧ weak_ordering(r)
    return relation_preserving(
        f, l,
        complement_of_converse<R>(r));
}
```

Определить strictly_increasing_counted_range и increasing_counted_range несложно.

Если задан предикат p на типе значений некоторого итератора, то интервал, распространяющийся на этот тип итератора, называется p-*разделенным*, при условии, что все значения в интервале, удовлетворяющие предикату, следуют за всеми значениями в интервале, не удовлетворяющими предикату.

[7]Некоторые авторы используют неуменьшающиеся и возрастающие интервалы вместо возрастающих и строго возрастающих соответственно.

Несложно предусмотреть проверку, которая показывает, является ли интервал разделенным по p:

```
template<typename I, typename P>
    requires(Readable(I) && Iterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
bool partitioned(I f, I l, P p)
{
    // Предусловие: readable_bounded_range(f, l)
    return l == find_if_not(find_if(f, l, p), l, p);
}
```

Итератор, возвращенный вызовом find_if, называется *точкой раздела*; это — первый итератор, если таковые вообще имеются, значение которого удовлетворяет предикату.

Упражнение 6.6. Реализуйте предикат partitioned_n, который проверяет, является ли счетный интервал разделенным по p.

В ходе линейного поиска необходимо вызывать source после каждого применения successor, поскольку неудавшаяся проверка не предоставляет информацию о значении любого другого итератора в интервале. Но равномерность разделенного интервала позволяет получить дополнительную информацию.

Лемма 6.8. Если p — предикат и $[f, l)$ — p-разделенный интервал:

$$(\forall m \in [f, l))\, \neg p(\mathsf{source}(m)) \Rightarrow (\forall j \in [f, m])\, \neg p(\mathsf{source}(j))$$
$$(\forall m \in [f, l))\, p(\mathsf{source}(m)) \Rightarrow (\forall j \in [m, l))\, p(\mathsf{source}(j))$$

Это наталкивает на идею создания алгоритма дихотомического поиска для определения точки раздела: при наличии равномерного распределения проверка существования средней точки интервала позволяет вдвое уменьшить область поиска. Но такой алгоритм может потребовать обхода подынтервала, по которому уже выполнен обход, что выдвигает требование регулярности successor.

6.6. Прямые итераторы

Преобразование функции successor в регулярную позволяет неоднократно проходить через один и тот же интервал и поддерживать в интервале больше одного итератора:

$$\mathit{ForwardIterator}(\mathsf{T}) \triangleq$$
$$\quad \mathit{Iterator}(\mathsf{T})$$
$$\quad \wedge\ \mathsf{regular\_unary\_function}(\mathsf{successor})$$

Следует учитывать, что различия между *Iterator* и *ForwardIterator* состоят только в одной аксиоме; новые операции отсутствуют. В дополнение к successor все другие функциональные процедуры, определенные на уточнениях концепции прямого итератора, которые будут введены ниже в этой главе, являются регулярными. Регулярность successor позволяет реализовать find_adjacent_mismatch без сохранения значения перед продвижением итератора:

```
template<typename I, typename R>
    requires(Readable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
I find_adjacent_mismatch_forward(I f, I l, R r)
{
    // Предусловие: readable_bounded_range(f, l)
    if (f == l) return l;
    I t;
    do {
        t = f;
        f = successor(f);
    } while (f != l && r(source(t), source(f)));
    return f;
}
```

Следует учитывать, что значение t указывает на первый элемент этой несогласованной пары и также может быть возвращено.

В главе 10 будет показано, как использовать *планирование концепции* для создания перегруженных версий алгоритма, написанных для различных концепций итератора. Для обозначения различных версий применяются суффиксы наподобие _forward.

Регулярность successor позволяет также реализовать алгоритм дихотомического поиска для обнаружения точки раздела:

```
template<typename I, typename P>
    requires(Readable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_point_n(I f, DistanceType(I) n, P p) {
    // Предусловие: readable_counted_range(f, n) ∧ partitioned_n(f, n, p)
    while (!zero(n)) {
        DistanceType(I) h = half_nonnegative(n);
        I m = f + h;
        if (p(source(m))) {
            n = h;
        } else {
            n = n - successor(h); f = successor(m);
        }
    }
```

```
    return f;
}
```

Лемма 6.9. partition_point_n возвращает точку раздела p-разделенного интервала $[\![f, n)\!)$.

Для обнаружения точки раздела в ограниченном интервале с помощью бисекции[8] необходимо вначале определить размер интервала:

```
template<typename I,  typename P>
    requires(Readable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_point(I f, I l, P p)
{
    // Предусловие: readable_bounded_range(f, l) ∧ partitioned(f, l, p)
    return partition_point_n(f, l - f, p);
}
```

Это определение точки раздела непосредственно приводит к алгоритмам дихотомического поиска в интервале, возрастающем по r, для нестрогого упорядочения r. Любое значение a, независимо от того, появляется ли оно в возрастающем интервале, определяет два итератора в этом интервале, называемые *нижней* и *верхней границами*. Неформально можно отметить, что нижней границей является первая позиция, в которой значение, эквивалентное a, может появиться в возрастающей последовательности. Аналогично верхняя граница — это последователь последней позиции, где может появиться значение, эквивалентное a. Поэтому элементы, эквивалентные a, появляются только в полуоткрытом интервале от нижней до верхней границы. В качестве примера укажем, что при условии полного упорядочения последовательность с нижней границей l и верхней границей u для значения a выглядит следующим образом:

$$\underbrace{x_0, x_1, \ldots, x_{l-1}}_{x_i<a}, \underbrace{x_l, \ldots, x_{u-1}}_{x_i=a}, \underbrace{x_u, x_{u+1}, \ldots, x_{n-1}}_{x_i>a}$$

Следует учитывать, что любая из этих трех областей может быть пустой.

[8] Метод бисекции восходит к доказательству теоремы о среднем значении в [Bolzano 1817] и независимо описан в [Cauchy 1821]. Больцано и Коши использовали этот метод для наиболее общего случая непрерывных функций, а в [Lagrange 1795] он использовался еще раньше для решения частной задачи аппроксимации корня многочлена. Первое описание применения бисекции для поиска приведено Джоном В. Мочли в лекции “Сортировка и упорядочение” [Mauchly 1946].

Лемма 6.10. В возрастающем интервале $[f, l)$, для любого значения a из типа значений для интервала, интервал секционируется по следующим двум предикатам:

$$\mathsf{lower\_bound}_a(x) \Leftrightarrow \neg r(x, a)$$
$$\mathsf{upper\_bound}_a(x) \Leftrightarrow r(a, x)$$

Это позволяет формально определить нижнюю и верхнюю границы как точки раздела для соответствующих предикатов.

Лемма 6.11. Итератор нижней границы предшествует или равен итератору верхней границы.

Реализация функционального объекта, соответствующего этому предикату, ведет непосредственно к созданию алгоритма определения нижней границы:

```
template<typename R>
    requires(Relation(R))
struct lower_bound_predicate
{
    typedef Domain(R) T;
    const T& a;
    R r;
    lower_bound_predicate(const T& a, R r) : a(a), r(r) { }
    bool operator()(const T& x) { return !r(x, a); }
};

template<typename I, typename R>
    requires(Readable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
I lower_bound_n(I f, DistanceType(I) n,
                const ValueType(I)& a, R r)
{
    // Предусловие: weak_ordering(r) ∧ increasing_counted_range(f, n, r)
    lower_bound_predicate<R> p(a, r);
    return partition_point_n(f, n, p);
}
```

Аналогично для верхней границы:

```
template<typename R>
    requires(Relation(R))
struct upper_bound_predicate
{
    typedef Domain(R) T;
    const T& a;
    R r;
    upper_bound_predicate(const T& a, R r) : a(a), r(r) { }
```

```
    bool operator()(const T& x) { return r(a, x); }
};

template<typename I, typename R>
    requires(Readable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
I upper_bound_n(I f, DistanceType(I) n,
                const ValueType(I)& a, R r)
{
    // Предусловие: weak_ordering(r) ∧ increasing_counted_range(f, n, r)
    upper_bound_predicate<R> p(a, r);
    return partition_point_n(f, n, p);
}
```

Упражнение 6.7. Реализуйте процедуру, которая возвращает и нижнюю, и верхнюю границы и выполняет меньше сравнений, чем сумма сравнений, которые были бы выполнены при вызове и lower_bound_n, и upper_bound_n[9].

Предикат применяется в середине интервала, и это гарантирует, что даже в наихудшем случае количество применений предиката в алгоритме поиска точки раздела будет оптимальным. При любом другом выборе лучшим стал бы конкурирующий алгоритм, который гарантирует, что больший подынтервал содержит точку раздела. Если априори известна ожидаемая позиция точки раздела, то проверка должна осуществляться в этой точке.

В partition_point_n предикат применяется $\lfloor \log_2 n \rfloor + 1$ раз, поэтому длина интервала при каждом шаге сокращается вдвое. Количество сложений итераторов/целых чисел, выполняемых в алгоритме, определяется по логарифмическому закону.

Лемма 6.12. Что касается прямого итератора, то общее количество операций successor, выполняемых в алгоритме, меньше или равно размеру интервала.

partition_point вычисляет также $l - f$, что в случае прямых итераторов приводит к выполнению еще n вызовов successor. Этот алгоритм целесообразно использовать по отношению к прямым итераторам, таким как связные списки, когда применение предиката является более дорогостоящим, чем вызов successor.

Лемма 6.13. При условии, что ожидаемое расстояние до точки раздела равно половине размера интервала, partition_point работает быстрее по сравнению с find_if при обнаружении точки раздела для прямых итераторов в любом случае, если

$$\mathrm{cost}_{\mathrm{successor}} < \left(1 - 2\frac{\log_2 n}{n}\right) \mathrm{cost}_{\mathrm{predicate}}$$

[9] Аналогичная функция STL носит имя equal_range.

6.7. Индексированные итераторы

Для обеспечения лучших характеристик partition_point, lower_bound и upper_bound по сравнению с линейным поиском необходимо гарантировать, чтобы операции сложения целого числа с итератором и вычитания итератора из итератора выполнялись быстро:

$IndexedIterator(\mathsf{T}) \triangleq$
$\quad ForwardIterator(\mathsf{T})$
$\quad \wedge\ + : \mathsf{T} \times \mathsf{DistanceType}(\mathsf{T}) \to \mathsf{T}$
$\quad \wedge\ - : \mathsf{T} \times \mathsf{T} \to \mathsf{DistanceType}(\mathsf{T})$
$\quad \wedge\ +$ takes constant time
$\quad \wedge\ -$ takes constant time

Операции $+$ и $-$, которые были определены для *Iterator* в терминах successor, теперь должны стать примитивными и быстрыми. Эта концепция отличается от *ForwardIterator* только тем, что требования к сложности становятся более строгими. Мы ожидаем, что стоимость $+$ и $-$ на индексированных итераторах будет почти полностью идентична стоимости successor.

6.8. Двунаправленные итераторы

В некоторых ситуациях индексация неосуществима, но у нас есть возможность двигаться в обратном направлении:

$BidirectionalIterator(\mathsf{T}) \triangleq$
$\quad ForwardIterator(\mathsf{T})$
$\quad \wedge\ \mathsf{predecessor} : \mathsf{T} \to \mathsf{T}$
$\quad \wedge\ \mathsf{predecessor}$ takes constant time
$\quad \wedge\ (\forall \mathtt{i} \in \mathsf{T})\ \mathsf{successor}(\mathtt{i})$ определено $\Rightarrow$
$\qquad\qquad \mathsf{predecessor}(\mathsf{successor}(\mathtt{i}))$ определено и равно $\mathtt{i}$
$\quad \wedge\ (\forall \mathtt{i} \in \mathsf{T})\ \mathsf{predecessor}(\mathtt{i})$ определено $\Rightarrow$
$\qquad\qquad \mathsf{successor}(\mathsf{predecessor}(\mathtt{i}))$ определено и равно $\mathtt{i}$

Как и в отношении successor, функция predecessor не обязательно должна быть полной; аксиомы этой концепции связывают ее область определения с областью определения successor. Мы ожидаем, что стоимость predecessor будет почти полностью идентична стоимости successor.

Лемма 6.14. Если successor определена на двунаправленных итераторах $\mathtt{i}$ и $\mathtt{j}$,

$$\mathsf{successor}(\mathtt{i}) = \mathsf{successor}(\mathtt{j}) \Rightarrow \mathtt{i} = \mathtt{j}$$

В нестрогом интервале двунаправленных итераторов возможно движение в обратном направлении вплоть до начала интервала:

```
template<typename I>
    requires(BidirectionalIterator(I))
I operator-(I l, DistanceType(I) n)
{
    // Предусловие: n ⩾ 0 ∧ (∃f ∈ I) weak_range(f, n) ∧ l = f + n
    while (!zero(n)) {
        n = predecessor(n);
        l = predecessor(l);
    }
    return l;
}
```

С помощью двунаправленных итераторов можно осуществлять поиск в обратном направлении. Как было указано выше, при поиске в интервале итераторов n количество результатов составляет $n + 1$; это утверждение остается истинным, выполняется ли поиск в прямом или обратном направлении. Поэтому мы должны придерживаться такого соглашения, чтобы полуоткрытые левые интервалы обозначались в форме $(f, l]$. Для указания результата "не найдено" мы возвращаем f, что вынуждает нас возвращать successor(i), если найден удовлетворяющий условию элемент в итераторе i:

```
template<typename I, typename P>
    requires(Readable(I) && BidirectionalIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I find_backward_if(I f, I l, P p)
{
    // Предусловие: (f, l] — читаемый ограниченный полуоткрытый
    // левый интервал
    while (l != f && !p(source(predecessor(l))))
        l = predecessor(l);
    return l;
}
```

Сравнение этого варианта с find_if служит иллюстрацией к преобразованию программы: f и l меняются ролями, source(i) преобразуется в source(predecessor(i)), а successor(i) — в predecessor(i). При этом преобразовании в непустом интервале l допускает разадресацию, а f — нет.

Только что продемонстрированное преобразование программы может быть применено к любому алгоритму, который принимает интервал прямых итераторов. Поэтому возможно реализовать вспомогательный тип, который после получения типа двунаправленного итератора вырабатывает другой тип двунаправленного итератора, где successor преобразуется в predecessor, predecessor преобразуется в successor, а source — в source для predecessor[10]. Этот вспомогательный тип позволяет приспособить любой алгоритм для итераторов или

[10] В STL этот тип именуется вспомогательным типом обратного итератора.

прямых итераторов к работе в обратном направлении на двунаправленных итераторах, и это также позволяет в любом алгоритме на двунаправленных итераторах менять направления обхода.

Упражнение 6.8. Перепишите find_backward_if так, чтобы в этой функции осуществлялся только один вызов predecessor в цикле.

Упражнение 6.9. В качестве примера алгоритма, в котором используются и successor, и predecessor, реализуйте предикат, определяющий, является ли интервал палиндромом. Палиндром читается одинаково и в прямом, и в обратном направлениях.

6.9. Итераторы с произвольным доступом

Некоторые типы итераторов удовлетворяют требованиям и к индексированным, и к двунаправленным итераторам. Эти типы, называемые *итераторами с произвольным доступом*, позволяют воспользоваться всеми возможностями компьютерной адресации:

$$
\begin{aligned}
&\mathit{RandomAccessIterator}(\mathsf{T}) \triangleq \\
&\quad \mathit{IndexedIterator}(\mathsf{T}) \wedge \mathit{BidirectionalIterator}(\mathsf{T}) \\
&\wedge\ \mathit{TotallyOrdered}(\mathsf{T}) \\
&\wedge\ (\forall \mathsf{i}, \mathsf{j} \in \mathsf{T})\, \mathsf{i} < \mathsf{j} \Leftrightarrow \mathsf{i} \prec \mathsf{j} \\
&\wedge\ \mathsf{DifferenceType} : \mathit{RandomAccessIterator} \to \mathit{Integer} \\
&\wedge\ + : \mathsf{T} \times \mathsf{DifferenceType}(\mathsf{T}) \to \mathsf{T} \\
&\wedge\ - : \mathsf{T} \times \mathsf{DifferenceType}(\mathsf{T}) \to \mathsf{T} \\
&\wedge\ - : \mathsf{T} \times \mathsf{T} \to \mathsf{DifferenceType}(\mathsf{T}) \\
&\wedge\ < \text{ требует постоянных затрат времени} \\
&\wedge\ - \text{ применительно к итератору и целому числу требует постоянных} \\
&\quad \text{затрат времени}
\end{aligned}
$$

DifferenceType(T) является достаточно большим для включения расстояний и их аддитивных обратных значений; если i и j — итераторы из допустимого интервала, то разность i—j всегда определена. С итератором можно складывать отрицательное целое число или вычитать из него это число.

При менее строгих типах итераторов операции + и — определены только в пределах одного интервала. Что касается типов итераторов с произвольным доступом, то это утверждение относится к операции <, а также к операциям + и —. Вообще говоря, операция на двух итераторах определена, только если они принадлежат к одному и тому же интервалу.

Проект 6.1. Определите аксиомы, связывающие друг с другом операции на итераторах с произвольным доступом.

Мы не описываем более подробно итераторы с произвольным доступом по следующим причинам.

Теорема 6.1. Для любой процедуры, определенной на явно заданном интервале итераторов с произвольным доступом, можно найти другую процедуру с той же сложностью, которая определена на индексированных итераторах.

Доказательство. Поскольку операции на итераторах с произвольным доступом определены только на итераторах, принадлежащих к одному и тому же интервалу, возможно реализовать вспомогательный тип, который при наличии индексированного типа итератора производит тип итератора с произвольным доступом. Информация о состоянии такого итератора содержит сам итератор f и целое число i и представляет итератор f + i. Операции с итератором, такие как +, − и <, оперируют над i; source оперирует над f + i. Иными словами, итератор, указывающий на начало интервала, вместе с индексом в этом интервале ведет себя как итератор с произвольным доступом. □

Эта теорема показывает, что теоретически указанные концепции являются эквивалентными в любом контексте, в котором известны начала интервалов. На практике мы пришли к выводу, что применение менее строгой концепции не приводит к снижению производительности. Однако в некоторых случаях должна быть откорректирована сигнатура, чтобы в нее было включено начало интервала.

Проект 6.2. Реализуйте семейство абстрактных процедур поиска подпоследовательности в последовательности. Опишите компромиссы, к которым необходимо прийти при выборе соответствующего алгоритма[11].

6.10. Резюме

Алгебра предоставляет иерархию концепций, таких как полугруппы, моноиды и группы, которые позволяют формулировать алгоритмы в наиболее общем контексте. Подобным образом, концепции итератора (рис. 6.1) позволяют формулировать в наиболее общем контексте алгоритмы на последовательных структурах данных. При разработке этих концепций использовались три разновидности усовершенствований: добавление операции, уточнение семантики и ужесточение требований к сложности. В частности, применяемые в главе три концепции, *итератор*, *прямой итератор* и *индексированный итератор*, не только различаются своими операциями, но и имеют разную семантику и сложность. Широкий набор алгоритмов поиска для различных концепций

[11] Два из наиболее известных алгоритмов для этой задачи приведены в [Boyer and Moore 1977] и [Knuth, et al. 1977]. [Musser and Nishanov 1997] может служить хорошей основой для подготовки этих алгоритмов к работе на определенном уровне абстракции.

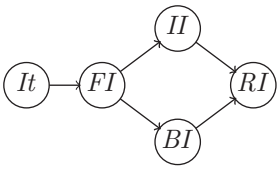


Рис. 6.1. Концепции итератора

итераторов, счетных и ограниченных интервалов и способов упорядочения интервалов может служить основой последовательного программирования.

Глава 7
Координатные структуры

В главе 6 представлено семейство концепций итераторов как интерфейса между алгоритмами и объектами в структурах данных с неизменяемой линейной формой. В настоящей главе мы выходим за рамки итераторов и приступаем к рассмотрению координатных структур с более сложной формой. В ней будут введены бифуркатные координаты и реализованы алгоритмы на бинарных деревьях с помощью машины для итерационного обхода дерева. После обсуждения схемы концепций для координатных структур изложение материала завершается алгоритмами для изоморфизма, эквивалентности и упорядочения.

7.1. Бифуркатные координаты

Итераторы позволяют осуществлять обход линейных структур, которые имеют единственного последователя в каждой позиции. Вместе с тем находят применение структуры данных с произвольным количеством последователей, поэтому в настоящей главе рассматривается важный случай структур с двумя и только с двумя последователями в каждой позиции, обозначенными как левый и правый. Чтобы можно было определять алгоритмы на этих структурах, сформулируем следующую концепцию:

$$
\begin{aligned}
&\mathit{BifurcateCoordinate}(\mathsf{T}) \triangleq \\
&\quad \mathit{Regular}(\mathsf{T}) \\
&\quad \wedge\ \mathsf{WeightType} : \mathit{BifurcateCoordinate} \to \mathit{Integer} \\
&\quad \wedge\ \mathsf{empty} : \mathsf{T} \to \mathsf{bool} \\
&\quad \wedge\ \mathsf{has\_left\_successor} : \mathsf{T} \to \mathsf{bool} \\
&\quad \wedge\ \mathsf{has\_right\_successor} : \mathsf{T} \to \mathsf{bool} \\
&\quad \wedge\ \mathsf{left\_successor} : \mathsf{T} \to \mathsf{T} \\
&\quad \wedge\ \mathsf{right\_successor} : \mathsf{T} \to \mathsf{T} \\
&\quad \wedge\ (\forall \mathsf{i}, \mathsf{j} \in \mathsf{T})\,(\mathsf{left\_successor}(\mathsf{i}) = \mathsf{j} \vee \mathsf{right\_successor}(\mathsf{i}) = \mathsf{j}) \Rightarrow \neg\mathsf{empty}(\mathsf{j})
\end{aligned}
$$

Функция типа WeightType возвращает тип, позволяющий подсчитывать все объекты в обходе, в котором используются бифуркатные координаты. Функция WeightType аналогична функции DistanceType для типа итератора.

Предикат empty всюду определен. Если он возвращает true, ни одна из прочих процедур не определена. empty является отрицанием предиката области определения и для has_left_successor, и для has_right_successor. has_left_successor — предикат области определения для left_successor, а has_right_successor — предикат области определения для right_successor. Иными словами, если бифуркатная координата не пуста, то has_left_successor и has_right_successor определены; если одна из этих функций возвращает true, то соответствующая функция определения последователя определена. В алгоритмах, предназначенных для работы с итераторами, используются данные о пределе или количестве для указания конца интервала. Что же касается бифуркатных координат, то на концах ветвей имеется много позиций. Поэтому более целесообразно ввести предикаты has_left_successor и has_right_successor для определения того, имеет ли координата последователей.

В данной книге мы описываем алгоритмы на концепции *BifurcateCoordinate*, где все операции являются регулярными. В этом состоит отличие от концепции *Iterator*, при использовании которой наиболее фундаментальные алгоритмы, такие как find, не требуют регулярности successor, и применяются нерегулярные модели наподобие входных потоков. Для структур, в которых применение left_successor и right_successor приводит к изменению формы основополагающего бинарного дерева, требуется концепция *WeakBifurcateCoordinate*, в рамках которой операции не являются регулярными.

Форма структуры, доступ к которой осуществляется с помощью итераторов, может быть циклической при нестрогом интервале и является линейно сегментированной при счетном или ограниченном интервале. Для обсуждения формы структуры, доступ к которой может осуществляться с помощью бифуркатных координат, необходимо сформулировать понятие достижимости.

Бифуркатная координата y является *подлинным потомком* другой координаты x, если y — левый или правый последователь x или подлинный потомок левого или правого последователя x. Бифуркатная координата y является *потомком* координаты x, если $y = x$ или y — подлинный потомок x.

Потомки x образуют *ориентированный ациклический граф*, если для всех y из потомков x узел y не является его собственным потомком. Иными словами, ни одна из последовательностей последователей любой координаты не должна вести к самой себе. x называется *корнем* ориентированного ациклического графа его потомков. Если потомки x образуют ориентированный ациклический граф и их количество является конечным, то они образуют *конечный ориентированный ациклический граф*. *Высота* конечного ориентированного ациклического графа — это увеличенная на единицу максимальная длина по-

следовательности последователей, начинающихся от корня графа, или нуль, если граф пуст.

Бифуркатная координата y является *достижимой слева* от x, если она — потомок левого последователя x; аналогичное определение относится к координате, *достижимой справа*.

Потомки x образуют *дерево*, если они входят в состав конечного ориентированного ациклического графа и для всех y, z среди потомков x узел z не является ни достижимым слева, ни достижимым справа от y. Иными словами, имеется уникальная последовательность последователей от некоторой координаты до любого из ее потомков. Свойство, которое определяет граф как дерево, имеет то же назначение для алгоритмов в этой главе, что и свойства, определяющие ограниченные или счетные интервалы, которые использовались в главе 6, — оно задает конечность, гарантирующую завершение:

property(C : *BifurcateCoordinate*)
tree : C
$x \mapsto$ потомки x образуют дерево

Ниже приведены рекурсивные алгоритмы вычисления веса и высоты дерева.

```
template<typename C>
    requires(BifurcateCoordinate(C))
WeightType(C) weight_recursive(C c)
{
    // Предусловие: tree(c)
    typedef WeightType(C) N;
    if (empty(c)) return N(0);
    N l(0);
    N r(0);
    if (has_left_successor(c))
        l = weight_recursive(left_successor(c));
    if (has_right_successor(c))
        r = weight_recursive(right_successor(c));
    return successor(l + r);
}

template<typename C>
    requires(BifurcateCoordinate(C))
WeightType(C) height_recursive(C c)
{
    // Предусловие: tree(c)
    typedef WeightType(C) N;
    if (empty(c)) return N(0);
    N l(0);
    N r(0);
    if (has_left_successor(c))
```

```
        l = height_recursive(left_successor(c));
    if (has_right_successor(c))
        r = height_recursive(right_successor(c));
    return successor(max(l, r));
}
```

Лемма 7.1. height_recursive(x) $\leqslant$ weight_recursive(x)

Функция height_recursive правильно вычисляет высоту ориентированного ациклического графа, но количество посещений ею каждой координаты равно количеству путей к этой координате; этот факт означает, что weight_recursive не вычисляет правильно вес ориентированного ациклического графа. Для алгоритмов обхода ориентированных ациклических графов и циклических структур требуется *маркирование*: способ запоминания того, какие координаты уже были посещены.

Могут быть предложены три основных порядка обхода дерева с поиском в глубину. При использовании всех этих трех порядков вначале полностью осуществляется обход левых потомков, а затем правых потомков. Посещения координаты по принципу *preorder* происходят перед обходом ее потомков; посещения *inorder* происходят между обходами левых и правых потомков; посещения *postorder* происходят после обхода всех потомков. Мы именуем эти три посещения с помощью следующего определения типа:

```
enum visit { pre, in, post };
```

Может быть выполнена любая комбинация обходов с помощью единственной процедуры, которая принимает в качестве параметра другую процедуру, принимающую посещение вместе с координатой:

```
template<typename C, typename Proc>
    requires(BifurcateCoordinate(C) &&
        Procedure(Proc) && Arity(Proc) == 2 &&
        visit == InputType(Proc, 0) &&
        C == InputType(Proc, 1))
Proc traverse_nonempty(C c, Proc proc)
{
    // Предусловие: tree(c) ∧ ¬empty(c)
    proc(pre, c);
    if (has_left_successor(c))
        proc = traverse_nonempty(left_successor(c), proc);
    proc(in, c);
    if (has_right_successor(c))
        proc = traverse_nonempty(right_successor(c), proc);
    proc(post, c);
    return proc;
}
```

7.2. Двунаправленные бифуркатные координаты

Для рекурсивного обхода требуется пространство стека, пропорциональное высоте дерева, которая может достигать значения веса; это часто недопустимо, если приходится иметь дело с большими несбалансированными деревьями. Кроме того, интерфейс к traverse_nonempty не позволяет осуществлять параллельный обход многочисленных деревьев. Вообще говоря, для обхода больше чем одного дерева одновременно необходимо предусмотреть отдельный стек для каждого дерева. Если бы мы объединили координату со стеком предыдущих координат, то создали бы новый тип координат с дополнительным преобразованием для получения предшественника. (Было бы более эффективно использовать действия, а не преобразования, чтобы избежать копирования каждый раз стека.) Такая координата моделировала бы концепцию *двунаправленной бифуркатной координаты*. Может быть предложена более простая и более гибкая модель этой концепции: деревья, которые включают связь предшественника в каждом узле. Такие деревья позволяют выполнять параллельные обходы с постоянной потребностью в пространстве и обеспечивают возможность реализовывать различные алгоритмы повторного балансирования. Издержки на сопровождение дополнительной связи обычно оправдываются.

$BidirectionalBifurcateCoordinate(\mathsf{T}) \triangleq$
  $BifurcateCoordinate(\mathsf{T})$
  $\wedge$ has_predecessor : $\mathsf{T} \to$ bool
  $\wedge$ $(\forall \mathtt{i} \in \mathsf{T})\, \neg$empty$(\mathtt{i}) \Rightarrow$ has_predecessor$(\mathtt{i})$ определено
  $\wedge$ predecessor : $\mathsf{T} \to \mathsf{T}$
  $\wedge$ $(\forall \mathtt{i} \in \mathsf{T})$ has_left_successor$(\mathtt{i}) \Rightarrow$
    predecessor(left_successor$(\mathtt{i})$) определено и равно $\mathtt{i}$
  $\wedge$ $(\forall \mathtt{i} \in \mathsf{T})$ has_right_successor$(\mathtt{i}) \Rightarrow$
    predecessor(right_successor$(\mathtt{i})$) определено и равно $\mathtt{i}$
  $\wedge$ $(\forall \mathtt{i} \in \mathsf{T})$ has_predecessor$(\mathtt{i}) \Rightarrow$
    is_left_successor$(\mathtt{i}) \vee$ is_right_successor$(\mathtt{i})$

где is_left_successor и is_right_successor определены следующим образом:

```
template<typename T>
    requires(BidirectionalBifurcateCoordinate(T))
bool is_left_successor(T j)
{
    // Предусловие: has_predecessor(j)
    T i = predecessor(j);
    return has_left_successor(i) && left_successor(i) == j;
}
```

```
template<typename T>
    requires(BidirectionalBifurcateCoordinate(T))
bool is_right_successor(T j) {
    // Предусловие: has_predecessor(j)
    T i = predecessor(j);
    return has_right_successor(i) && right_successor(i) == j;
}
```

Лемма 7.2. If x and y are bidirectional bifurcate coordinates,

$$\mathsf{left\_successor}(x) = \mathsf{left\_successor}(y) \Rightarrow x = y$$
$$\mathsf{left\_successor}(x) = \mathsf{right\_successor}(y) \Rightarrow x = y$$
$$\mathsf{right\_successor}(x) = \mathsf{right\_successor}(y) \Rightarrow x = y$$

Упражнение 7.1. Противоречит ли существование координаты x, такой, что

$$\mathsf{is\_left\_successor}(x) \wedge \mathsf{is\_right\_successor}(x)$$

аксиомам двунаправленных бифуркатных координат?

traverse_nonempty посещает каждую координату три раза, независимо от того, имеет ли она последователей; благодаря сохранению этого инварианта обход становится единообразным. Три посещения координаты всегда происходят в одном и том же порядке (pre, in, post), поэтому, если даны текущая координата и только что выполненное посещение этой координаты, можно определить следующую координату и следующее состояние с использованием лишь информации о координате и ее предшественнике. Эти соображения приводят к созданию итерационного алгоритма обхода дерева с двунаправленными бифуркатными координатами, характеризующегося постоянной потребностью в пространстве. Обход зависит от *машины* — последовательности инструкций, используемых как компонент многих алгоритмов:

```
template<typename C>
    requires(BidirectionalBifurcateCoordinate(C))
int traverse_step(visit& v, C& c)
{
    // Предусловие: has_predecessor(c) ∨ v ≠ post
    switch (v) {
    case pre:
        if (has_left_successor(c))  {
                    c = left_successor(c);  return 1;
        }   v = in;                         return 0;
    case in:
        if (has_right_successor(c)) {
            v = pre; c = right_successor(c); return 1;
```

```
        }   v = post;                          return 0;
    case post:
        if (is_left_successor(c))
            v = in;
                    c = predecessor(c);        return -1;
    }
}
```

Значением, возвращаемым процедурой, является изменение по высоте. В алгоритме, основанном на traverse_step, используется цикл, в котором определяется, достигнута ли исходная координата в последнем посещении (post):

```
template<typename C>
    requires(BidirectionalBifurcateCoordinate(C))
bool reachable(C x, C y)
{
    // Предусловие: tree(x)
    if (empty(x)) return false;
    C root = x;
    visit v = pre;
    do {
        if (x == y) return true;
        traverse_step(v, x);
    } while (x != root || v != post);
    return false;
}
```

Лемма 7.3. Если reachable возвращает true, то $v = \text{pre}$ имеет место непосредственно перед возвратом.

Чтобы вычислить вес дерева, подсчитываем количество посещений pre в обходе:

```
template<typename C>
    requires(BidirectionalBifurcateCoordinate(C))
WeightType(C) weight(C c)
{
    // Предусловие: tree(c)
    typedef WeightType(C) N;
    if (empty(c)) return N(0);
    C root = c;
    visit v = pre;
    N n(1); // Инвариант: n — количество выполненных посещений pre
    do {
        traverse_step(v, c);
        if (v == pre) n = successor(n);
    } while (c != root || v != post);
```

```
    return n;
}
```

Упражнение 7.2. Внесите в weight такие изменения, чтобы осуществлялся подсчет посещений in или post вместо pre.

Для вычисления высоты дерева необходимо отслеживать текущие значения высоты и достигнутого максимума:

```
template<typename C>
    requires(BidirectionalBifurcateCoordinate(C))
WeightType(C) height(C c)
{
    // Предусловие: tree(c)
    typedef WeightType(C) N;
    if (empty(c)) return N(0);
    C root = c;
    visit v = pre;
    N n(1); // Инвариант: n — максимальное значение высоты в посещениях
            // pre, достигнутое до настоящего времени
    N m(1); // Инвариант: m — высота текущего посещения pre
    do {
        m = (m - N(1)) + N(traverse_step(v, c) + 1);
        n = max(n, m);
    } while (c != root || v != post);
    return n;
}
```

Дополнительные значения -1 и $+1$ предусмотрены на тот случай, что тип WeightType — без знака. Данный код можно усовершенствовать с применением версии, в которой значение max накапливается.

Мы можем определить итерационную процедуру, соответствующую traverse_nonempty. Включаем проверку на то, что дерево — пустое, поскольку такая проверка не выполняется при каждом рекурсивном вызове:

```
template<typename C, typename Proc>
    requires(BidirectionalBifurcateCoordinate(C) &&
        Procedure(Proc) && Arity(Proc) == 2 &&
        visit == InputType(Proc, 0) &&
        C == InputType(Proc, 1))
Proc traverse(C c, Proc proc)
{
    // Предусловие: tree(c)
    if (empty(c)) return proc;
    C root = c;
    visit v = pre;
    proc(pre, c);
```

```
    do {
        traverse_step(v, c);
        proc(v, c);
    } while (c != root || v != post);
    return proc;
}
```

Упражнение 7.3. Воспользуйтесь traverse_step и процедурами из главы 2 для определения того, образуют ли потомки двунаправленной бифуркатной координаты ориентированный ациклический граф.

Свойство readable_bounded_range для итераторов указывает, что для каждого итератора в интервале source определено. Аналогичным свойством для бифуркатных координат является следующее:

property(C : *Readable*)
  requires(BifurcateCoordinate(C))
readable_tree : C
  $x \mapsto \mathsf{tree}(x) \wedge (\forall y \in C)\, \mathsf{reachable}(x, y) \Rightarrow \mathsf{source}(y)$ определено

Алгоритмы для итераторов, такие как find и count, можно распространить на бифуркатные координаты, для чего применяются два подхода: реализация специализированных версий и реализация вспомогательного типа.

Проект 7.1. Реализуйте версии алгоритмов из главы 6 для двунаправленных бифуркатных координат.

Проект 7.2. Спроектируйте вспомогательный тип, который после получения типа двунаправленной бифуркатной координаты производит тип итератора, осуществляющего доступ к координатам в порядке обхода (pre, in или post), определенного при создании итератора.

7.3. Координатные структуры

До сих пор мы определили отдельные концепции, каждая из которых задает набор процедур и их семантику. Иногда бывает полезно определить *схему концепций*, представляющую собой способ описания некоторых общих свойств семейства концепций. Разумеется, невозможно определять алгоритмы на основе схемы концепций, но возможно описывать структуры связанных алгоритмов на различных концепциях, принадлежащих к одной и той же схеме концепций. Например, мы определили несколько концепций итераторов, описывающих линейные обходы, и концепций бифуркатных координат, описывающих обход бинарных деревьев. Чтобы обеспечить обход в пределах произвольных структур данных, мы вводим схему концепций под названием *координатные*

структуры. Координатная структура может иметь несколько взаимосвязанных типов координат, каждый из которых характеризуется применением различных функций обхода. Координатные структуры абстрагируют навигационные аспекты структур данных, тогда как сложные объекты, которые будут введены в главе 12, абстрагируют управление и владение памятью. Для описания одного и того же множества объектов могут применяться многие разные координатные структуры.

Концепция представляет собой *координатную структуру*, если она состоит из одного или нескольких типов координат, нуля или большего количества типов значений, одной или нескольких функций обхода и нуля или большего количества функций доступа. Каждая функция обхода отображает один или несколько типов координат и типов значений в тип координат, а каждая функция доступа отображает один или несколько типов координат и типов значений в тип значений. Например, читаемый индексированный итератор, будучи рассматриваемым как координатная структура, имеет один тип значений и два типа координат: тип итератора и его тип расстояния. Функциями обхода являются + (складывающая расстояние с итератором) и − (позволяющая получить расстояние между двумя итераторами). Предусмотрена одна функция доступа: source.

7.4. Изоморфизм, эквивалентность и упорядочение

Две коллекции координат из одной и той же концепции координатной структуры являются *изоморфными*, если они имеют одну и ту же форму. Более формально, коллекции являются изоморфными, если имеется взаимнооднозначное соответствие между этими двумя коллекциями, такое, что любое допустимое применение функции обхода к координатам из первой коллекции возвращает координату, соответствующую применению той же функции обхода к соответствующим координатам из второй коллекции.

Изоморфизм не зависит от значений объектов, на которые указывают координаты: в алгоритмах проверки изоморфизма используются только функции обхода. Но изоморфизм требует, чтобы для соответствующих координат были определены или не определены одни и те же функции доступа. Например, два ограниченных или счетных интервала изоморфны, если имеют одинаковый размер. Два нестрогих интервала прямых итераторов изоморфны, если они имеют одну и ту же структуру орбиты, как определено в главе 2. Два дерева изоморфны, если оба они пусты; если же оба дерева непусты, то изоморфизм определяется с помощью следующего кода:

```
template<typename C0, typename C1>
    requires(BifurcateCoordinate(C0) &&
```

```
        BifurcateCoordinate(C1))
bool bifurcate_isomorphic_nonempty(C0 c0, C1 c1)
{
    // Предусловие: tree(c0) ∧ tree(c1) ∧ ¬empty(c0) ∧ ¬empty(c1)
    if (has_left_successor(c0))
        if (has_left_successor(c1)) {
            if (!bifurcate_isomorphic_nonempty(
                    left_successor(c0), left_successor(c1)))
                return false;
        } else return false;
    else if (has_left_successor(c1)) return false;
    if (has_right_successor(c0))
        if (has_right_successor(c1)) {
            if (!bifurcate_isomorphic_nonempty(
                    right_successor(c0), right_successor(c1)))
                return false;
        } else return false;
    else if (has_right_successor(c1)) return false;
    return true;
}
```

Лемма 7.4. Что касается двунаправленных бифуркатных координат, то деревья изоморфны, если одновременные их обходы состоят из одинаковых последовательностей посещений:

```
template<typename C0, typename C1>
    requires(BidirectionalBifurcateCoordinate(C0) &&
        BidirectionalBifurcateCoordinate(C1))
bool bifurcate_isomorphic(C0 c0, C1 c1)
{
    // Предусловие: tree(c0) ∧ tree(c1)
    if (empty(c0)) return empty(c1);
    if (empty(c1)) return false;
    C0 root0 = c0;
    visit v0 = pre;
    visit v1 = pre;
    while (true) {
        traverse_step(v0, c0);
        traverse_step(v1, c1);
        if (v0 != v1) return false;
        if (c0 == root0 && v0 == post) return true;
    }
}
```

В главе 6 содержатся алгоритмы для линейного и дихотомического поиска, зависящие от свойств равенства и полного упорядочения, которые составляют

часть понятия регулярности. Определив операции равенства и упорядочения на коллекциях координат из координатной структуры, мы можем обеспечить поиск коллекции объектов, а не отдельных объектов.

Две коллекции координат из одной и той же читаемой концепции координатной структуры и с одними и теми же типами значений являются *эквивалентными* согласно заданным отношениям эквивалентности (по одному на каждый тип значений), если они изоморфны и если применение одной и той же функции доступа к соответствующим координатам из этих двух коллекций приводит к возврату эквивалентных объектов. Замена отношений эквивалентности равенствами для типов значений естественным образом приводит к определению равенства коллекций координат.

Два читаемых ограниченных интервала эквивалентны, если они имеют один и тот же размер и если соответствующие итераторы имеют эквивалентные значения:

```
template<typename I0, typename I1, typename R>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) &&
        ValueType(I0) == ValueType(I1) &&
        Relation(R) && ValueType(I0) == Domain(R))
bool lexicographical_equivalent(I0 f0, I0 l0, I1 f1, I1 l1, R r)
{
    // Предусловие: readable_bounded_range(f0, l0)
    // Предусловие: readable_bounded_range(f1, l1)
    // Предусловие: equivalence(r)
    pair<I0, I1> p = find_mismatch(f0, l0, f1, l1, r);
    return p.m0 == l0 && p.m1 == l1;
}
```

Несложно реализовать lexicographical_equal, передавая функциональный объект, реализующий равенство на типе значений, функции lexicographical_equivalent:

```
template<typename T>
    requires(Regular(T))
struct equal
{
    bool operator()(const T& x, const T& y)
    {
        return x == y;
    }
};

template<typename I0, typename I1>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) &&
```

```
        ValueType(I0) == ValueType(I1))
bool lexicographical_equal(I0 f0, I0 l0, I1 f1, I1 l1)
{
    return lexicographical_equivalent(f0, l0, f1, l1,
                                      equal<ValueType(I0)>());
}
```

Два читаемых дерева эквивалентны, если они изоморфны и если соответствующие координаты имеют эквивалентные значения:

```
template<typename C0, typename C1, typename R>
    requires(Readable(C0) && BifurcateCoordinate(C0) &&
        Readable(C1) && BifurcateCoordinate(C1) &&
        ValueType(C0) == ValueType(C1) &&
        Relation(R) && ValueType(I0) == Domain(R))
bool bifurcate_equivalent_nonempty(C0 c0, C1 c1, R r)
{
    // Предусловие: readable_tree(c0) ∧ readable_tree(c1)
    // Предусловие: ¬empty(c0) ∧ ¬empty(c1)
    // Предусловие: equivalence(r)
    if (!r(source(c0), source(c1))) return false;
    if (has_left_successor(c0))
        if (has_left_successor(c1)) {
            if (!bifurcate_equivalent_nonempty(
                  left_successor(c0), left_successor(c1), r))
                return false;
        } else return false;
    else if (has_left_successor(c1)) return false;
    if (has_right_successor(c0))
        if (has_right_successor(c1)) {
            if (!bifurcate_equivalent_nonempty(
                  right_successor(c0), right_successor(c1), r))
                return false;
        } else return false;
    else if (has_right_successor(c1)) return false;
    return true;
}
```

Что касается двунаправленных бифуркатных координат, то деревья эквивалентны, если одновременные их обходы приводят к одинаковой последовательности посещений и если соответствующие координаты имеют эквивалентные значения:

```
template<typename C0, typename C1, typename R>
    requires(Readable(C0) &&
        BidirectionalBifurcateCoordinate(C0) &&
        Readable(C1) &&
```

```
        BidirectionalBifurcateCoordinate(C1) &&
        ValueType(C0) == ValueType(C1) &&
        Relation(R) && ValueType(C) == Domain(R))
bool bifurcate_equivalent(C0 c0, C1 c1, R r)
{
    // Предусловие: readable_tree(c0) ∧ readable_tree(c1)
    // Предусловие: equivalence(r)
    if (empty(c0)) return empty(c1);
    if (empty(c1)) return false;
    C0 root0 = c0;
    visit v0 = pre;
    visit v1 = pre;
    while (true) {
        if (v0 == pre && !r(source(c0), source(c1)))
            return false;
        traverse_step(v0, c0);
        traverse_step(v1, c1);
        if (v0 != v1) return false;
        if (c0 == root0 && v0 == post) return true;
    }
}
```

Мы можем распространить нестрогое (полное) упорядочение на читаемые интервалы итераторов, используя лексикографическое упорядочение, при котором игнорируются префиксы эквивалентных (равных) значений, а более короткий интервал рассматривается как предшествующий более длинному:

```
template<typename I0, typename I1, typename R>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) &&
        ValueType(I0) == ValueType(I1) &&
        Relation(R) && ValueType(I0) == Domain(R))
bool lexicographical_compare(I0 f0, I0 l0, I1 f1, I1 l1, R r)
{
    // Предусловие: readable_bounded_range(f0, l0)
    // Предусловие: readable_bounded_range(f1, l1)
    // Предусловие: weak_ordering(r)
    while (true) {
        if (f1 == l1) return false;
        if (f0 == l0) return true;
        if (r(source(f0), source(f1))) return true;
        if (r(source(f1), source(f0))) return false;
        f0 = successor(f0);
        f1 = successor(f1);
    }
}
```

Несложно специализировать этот код для lexicographical_less, передавая как r функциональный объект, воплощающий в себе операцию < на типе значений:

```
template<typename T>
    requires(TotallyOrdered(T))
struct less {
    bool operator()(const T& x, const T& y)
    {
        return x < y;
    }
};

template<typename I0, typename I1>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) &&
        ValueType(I0) == ValueType(I1))
bool lexicographical_less(I0 f0, I0 l0, I1 f1, I1 l1)
{
    return lexicographical_compare(f0, l0, f1, l1,
                                   less<ValueType(I0)>());
}
```

Упражнение 7.4. Объясните, почему в функции lexicographical_compare третью и четвертую инструкции `if` можно переставить местами, а первую и вторую — нет.

Упражнение 7.5. Объясните, почему мы не стали реализовывать функцию lexicographical_compare с помощью find_mismatch.

Мы можем также распространить лексикографическое упорядочение на бифуркатные координаты, игнорируя эквивалентные поддеревья, исходящие из корня, и рассматривая координату без левого последователя как предшествующую координате, имеющей левого последователя. Если текущие значения и левые поддеревья не определяют результат, то рассматривайте координату без правого последователя как предшествующую координате, имеющей правого последователя.

Упражнение 7.6. Реализуйте bifurcate_compare_nonempty для читаемых бифуркатных координат.

Читатели, выполнившие предыдущее упражнение, оценят простоту сравнения деревьев на основе двунаправленных координат и итерационного обхода:

```
template<typename C0, typename C1, typename R>
    requires(Readable(C0) &&
        BidirectionalBifurcateCoordinate(C0) &&
        Readable(C1) &&
```

```
        BidirectionalBifurcateCoordinate(C1) &&
        Relation(R) && ValueType(C) == Domain(R))
bool bifurcate_compare(C0 c0, C1 c1, R r)
{
    // Предусловие: readable_tree(c0) ∧ readable_tree(c1) ∧ weak_ordering(r)
    if (empty(c1)) return false;
    if (empty(c0)) return true;
    C0 root0 = c0;
    visit v0 = pre;
    visit v1 = pre;
    while (true) {
        if (v0 == pre) {
            if (r(source(c0), source(c1))) return true;
            if (r(source(c1), source(c0))) return false;
        }
        traverse_step(v0, c0);
        traverse_step(v1, c1);
        if (v0 != v1) return v0 > v1;
        if (c0 == root0 && v0 == post) return false;
    }
}
```

Мы можем реализовать bifurcate_shape_compare, передавая в bifurcate_compare отношение, которое всегда имеет значение false. Это позволяет отсортировать интервал деревьев, а затем воспользоваться upper_bound для поиска изоморфного дерева за логарифмическое время.

Проект 7.3. Спроектируйте координатную структуру для семейства структур данных и распространите понятия изоморфизма, эквивалентности и упорядочения на эту координатную структуру.

7.5. Резюме

Линейные структуры играют фундаментальную роль в информатике, а итераторы предоставляют естественный интерфейс между такими структурами и алгоритмами для работы с ними. Однако имеются нелинейные структуры данных со своими собственными нелинейными координатными структурами. Двунаправленные бифуркатные координаты предоставляют пример итеративных алгоритмов, весьма отличных от алгоритмов на интервалах итераторов. Мы распространяем понятия изоморфизма, равенства и упорядочения на коллекции координат с различными топологиями.

Глава 8

Координаты с изменяемыми последователями

В настоящей главе вводятся концепции итератора и координатной структуры, которые обеспечивают повторное связывание: изменение successor или других функций обхода применительно к конкретной координате. Повторное связывание позволяет реализовать такие переупорядочения, как сортировки, которые сохраняют значение source в координате. Мы вводим машины повторного связывания, которые сохраняют определенные структурные свойства координат. В заключение рассматривается машина, позволяющая выполнять определенные обходы дерева без использования стека или связей с предшественником с помощью временного повторного связывания координат в течение обхода.

8.1. Связанные итераторы

В главе 6 функция successor для заданного итератора рассматривалась как неизменяемая: применение successor к определенному значению итератора всегда приводит к возврату одного и того же результата. Тип *связанного итератора* — это тип прямого итератора, для которого существует *объект формирователя связей*; применение объекта формирователя связей к итератору позволяет преобразовать функцию successor для этого итератора в изменяемую. Такие итераторы моделируются с помощью связанных списков, в которых связи между узлами могут быть изменены. Мы используем объекты формирователей связей, а не единственную функцию set_successor, перегруженную применительно к типу итератора, чтобы иметь возможность использовать различные связи для одной и той же структуры данных. Например, двухсвязные списки могут быть связаны путем задания связей и с последователем, и с предшественником или с помощью установления связей только с последователем. Это позволяет свести работу многопроходного алгоритма к минимуму путем избавления от необходимости сопровождать связи с пред-

шественником до заключительного прохода. Таким образом, мы определяем концепции для связанных итераторов косвенно, в терминах соответствующих объектов формирователей связей. Но неформально мы продолжаем вести речь о типах связанных итераторов. Чтобы определить требования к объектам формирователей связей, мы определяем следующие взаимозависимые концепции:

$ForwardLinker(\mathrm{S}) \triangleq$
  $\mathsf{IteratorType} : ForwardLinker \to ForwardIterator$
  $\wedge$ Пусть $\mathrm{I} = \mathsf{IteratorType}(\mathrm{S})$ в:
    $(\forall \mathrm{s} \in \mathrm{S})\,(\mathrm{s} : \mathrm{I} \times \mathrm{I} \to \mathsf{void})$
    $\wedge$ $(\forall \mathrm{s} \in \mathrm{S})\,(\forall \mathrm{i}, \mathrm{j} \in \mathrm{I})$ if $\mathsf{successor}(\mathrm{i})$ определено,
      то $\mathrm{s}(\mathrm{i}, \mathrm{j})$ устанавливает $\mathsf{successor}(\mathrm{i}) = \mathrm{j}$

$BackwardLinker(\mathrm{S}) \triangleq$
  $\mathsf{IteratorType} : BackwardLinker \to BidirectionalIterator$
  $\wedge$ Пусть $\mathrm{I} = \mathsf{IteratorType}(\mathrm{S})$ в:
    $(\forall \mathrm{s} \in \mathrm{S})\,(\mathrm{s} : \mathrm{I} \times \mathrm{I} \to \mathsf{void})$
    $\wedge$ $(\forall \mathrm{s} \in \mathrm{S})\,(\forall \mathrm{i}, \mathrm{j} \in \mathrm{I})$ if $\mathsf{predecessor}(\mathrm{j})$ определено,
      то $\mathrm{s}(\mathrm{i}, \mathrm{j})$ устанавливает $\mathrm{i} = \mathsf{predecessor}(\mathrm{j})$

$BidirectionalLinker(\mathrm{S}) \triangleq ForwardLinker(\mathrm{S}) \wedge BackwardLinker(\mathrm{S})$

Два интервала являются *непересекающимися*, если они не включают ни одного общего итератора. Для полуоткрытых ограниченных интервалов это соответствует следующему:

property$(\mathrm{I} : Iterator)$
$\mathsf{disjoint} : \mathrm{I} \times \mathrm{I} \times \mathrm{I} \times \mathrm{I}$
  $(\mathrm{f0}, \mathrm{l0}, \mathrm{f1}, \mathrm{l1}) \mapsto (\forall \mathrm{i} \in \mathrm{I}) \neg(\mathrm{i} \in [\mathrm{f0}, \mathrm{l0}) \wedge \mathrm{i} \in [\mathrm{f1}, \mathrm{l1}))$

то же можно определить для других разновидностей интервалов. Разумеется, связанные итераторы остаются итераторами, поэтому по отношению к ним можно использовать все понятия, которые были определены нами для интервалов, но для связанных итераторов непересекаемость и все другие свойства интервалов со временем могут изменяться. Возможно, чтобы для непересекающихся интервалов прямых итераторов только с формирователем прямых связей (односвязных списков) был предусмотрен один и тот же предел, обычно называемый *nil*.

8.2. Переупорядочение связей

Переупорядочение связей — это алгоритм, принимающий один или несколько связанных интервалов и возвращающий один или несколько связанных интервалов, а также удовлетворяющий следующим свойствам.

- Входные интервалы (будь то счетные или ограниченные) являются попарно непересекающимися.

- Выходные интервалы (будь то счетные или ограниченные) являются попарно непересекающимися.
- Каждый итератор во входном интервале появляется в одном из выходных интервалов.
- Каждый итератор в выходном интервале появляется в одном из входных интервалов.
- Каждый итератор в каждом выходном интервале обозначает тот же объект, что и перед переупорядочением, и этот объект имеет то же значение.

Следует отметить, что отношения successor и predecessor, которые соблюдались во входных интервалах, могут не соблюдаться в выходных интервалах.

Переупорядочение связей является *сохраняющим предшествование*, если всякий раз, когда два итератора $\mathtt{i} \prec \mathtt{j}$ в выходном интервале исходят из одного и того же входного интервала, оказывается, что условие $\mathtt{i} \prec \mathtt{j}$ изначально соблюдалось во входном интервале.

При реализации переупорядочения связей необходимо следить за тем, чтобы удовлетворялись свойства непересекаемости, сохранения и упорядочения. На следующем этапе мы представим три короткие процедуры, или машины, каждая из которых выполняет один шаг обхода или связывания, а затем приступим к составлению на основе этих машин переупорядочений связей для разбиения, объединения и обращения связанных интервалов. Первые две машины устанавливают или поддерживают отношения $\mathtt{f} = \mathsf{successor}(\mathtt{t})$ между двумя объектами итераторов, передаваемыми по ссылке:

```
template<typename I>
    requires(ForwardIterator(I))
void advance_tail(I& t, I& f)
{
    // Предусловие: successor(f) определено
    t = f;
    f = successor(f);
}

template<typename S>
    requires(ForwardLinker(S))
struct linker_to_tail
{
    typedef IteratorType(S) I;
    S set_link;
    linker_to_tail(const S& set_link) : set_link(set_link) { }
    void operator()(I& t, I& f)
    {
        // Предусловие: successor(f) определено
        set_link(t, f);
```

```
        advance_tail(t, f);
    }
};
```

Мы можем использовать advance_tail для поиска последнего итератора в непустом ограниченном интервале[1]:

```
template<typename I>
    requires(ForwardIterator(I))
I find_last(I f, I l) {
    // Предусловие: bounded_range(f, l) ∧ f ≠ l
    I t;
    do
        advance_tail(t, f);
    while (f != l);
    return t;
}
```

Можно также использовать advance_tail и linker_to_tail совместно для разбиения одного интервала на два с учетом значения *псевдопредиката*, применяемого к каждому итератору. *Псевдопредикат* не обязательно является регулярным, причем его результат может зависеть от собственного состояния, а также от его входов. Например, псевдопредикат может игнорировать свои аргументы и возвращать попеременно значения false и true. Рассматриваемый алгоритм принимает ограниченный интервал связанных итераторов, псевдопредикат на типе связанного итератора и объект формирователя связей. Алгоритм возвращает пару интервалов: итераторы, не удовлетворяющие псевдопредикату, и итераторы, удовлетворяющие ему. Полезно представить эти возвращаемые интервалы как закрытые ограниченные интервалы $[h, t]$, где h является первым, или *головным*, итератором, а t — последним, или *хвостовым*, итератором. Возврат хвоста каждого интервала позволяет вызывающему объекту повторно связывать этот итератор, не будучи вынужденным производить обход по направлению к нему (например, с использованием find_last). Но любой из возвращенных интервалов может быть пустым, для представления чего применяется возврат $h = t = l$, где l — предел входного интервала. Связи successor хвостов двух возвращенных интервалов не изменяются этим алгоритмом. Сам алгоритм приведен ниже.

```
template<typename I, typename S, typename Pred>
    requires(ForwardLinker(S) && I == IteratorType(S) &&
        UnaryPseudoPredicate(Pred) && I == Domain(Pred))
pair< pair<I, I>, pair<I, I> >
split_linked(I f, I l, Pred p, S set_link)
```

[1]Следует отметить, что в функции find_adjacent_mismatch_forward из главы 6 неявно использовалась advance_tail.

```
{
    // Предусловие: bounded_range(f, l)
    typedef pair<I, I> P;
    linker_to_tail<S> link_to_tail(set_link);
    I h0 = l; I t0 = l;
    I h1 = l; I t1 = l;
    if (f == l)                                goto s4;
    if (p(f)) { h1 = f; advance_tail(t1, f); goto s1; }
    else      { h0 = f; advance_tail(t0, f); goto s0; }
s0: if (f == l)                                goto s4;
    if (p(f)) { h1 = f; advance_tail(t1, f); goto s3; }
    else      {         advance_tail(t0, f); goto s0; }
s1: if (f == l)                                goto s4;
    if (p(f)) {         advance_tail(t1, f); goto s1; }
    else      { h0 = f; advance_tail(t0, f); goto s2; }
s2: if (f == l)                                goto s4;
    if (p(f)) {         link_to_tail(t1, f); goto s3; }
    else      {         advance_tail(t0, f); goto s2; }
s3: if (f == l)                                goto s4;
    if (p(f)) {         advance_tail(t1, f); goto s3; }
    else      {         link_to_tail(t0, f); goto s2; }
s4: return pair<P, P>(P(h0, t0), P(h1, t1));
}
```

Эта процедура представляет собой конечный автомат. Переменные $\mathrm{t0}$ и $\mathrm{t1}$ указывают на хвосты двух выходных интервалов соответственно. Эти состояния соответствуют следующим условиям:

s0: $\mathsf{successor}(\mathrm{t0}) = \mathrm{f} \wedge \neg \mathrm{p}(\mathrm{t0})$

s1: $\mathsf{successor}(\mathrm{t1}) = \mathrm{f} \wedge \mathrm{p}(\mathrm{t1})$

s2: $\mathsf{successor}(\mathrm{t0}) = \mathrm{f} \wedge \neg \mathrm{p}(\mathrm{t0}) \wedge \mathrm{p}(\mathrm{t1})$

s3: $\mathsf{successor}(\mathrm{t1}) = \mathrm{f} \wedge \neg \mathrm{p}(\mathrm{t0}) \wedge \mathrm{p}(\mathrm{t1})$

Повторное связывание необходимо только при перемещениях между состояниями s2 и s3. Инструкции `goto`, которые ведут от состояния к непосредственно следующему за ним состоянию, включены для симметричности.

Лемма 8.1. Для каждого из интервалов $[\mathrm{h}, \mathrm{t}]$, возвращенных split_linked, имеет место $\mathrm{h} = \mathrm{l} \Leftrightarrow \mathrm{t} = \mathrm{l}$.

Упражнение 8.1. При условии, что один из интервалов (h, t), возвращенных split_linked, не пуст, объясните, на что указывает итератор t и каково значение $\mathsf{successor}(\mathrm{t})$.

Лемма 8.2. split_linked является сохраняющим предшествование переупорядочением связей.

Мы можем также использовать advance_tail и linker_to_tail для реализации алгоритма объединения двух интервалов в один интервал на основе псевдоотношения, применяемого к головам оставшихся частей входных интервалов. *Псевдоотношение* — это бинарный однородный псевдопредикат, который поэтому не обязательно должен быть регулярным. Этот алгоритм принимает два ограниченных интервала связанных итераторов, псевдоотношение на типе связанного итератора и объект формирователя связей. Алгоритм возвращает тройку $(\mathrm{f}, \mathrm{t}, \mathrm{l})$, где $[\mathrm{f}, \mathrm{l})$ — полуоткрытый интервал объединенных итераторов, а $\mathrm{t} \in [\mathrm{f}, \mathrm{l})$ — итератор, посещенный последним. Последующий вызов $\mathsf{find\_last}(\mathrm{t}, \mathrm{l})$ возвращает последний итератор в интервале, что позволяет его связать с другим интервалом. Сам алгоритм приведен ниже.

```
template<typename I, typename S, typename R>
    requires(ForwardLinker(S) && I == IteratorType(S) &&
        PseudoRelation(R) && I == Domain(R))
triple<I, I, I>
combine_linked_nonempty(I f0, I l0, I f1, I l1, R r, S set_link)
{
    // Предусловие: bounded_range(f0, l0) ∧ bounded_range(f1, l1)
    // Предусловие: f0 ≠ l0 ∧ f1 ≠ l1 ∧ disjoint(f0, l0, f1, l1)
    typedef triple<I, I, I> T;
    linker_to_tail<S> link_to_tail(set_link);
    I h; I t;
    if (r(f1, f0)) { h = f1; advance_tail(t, f1); goto s1; }
    else           { h = f0; advance_tail(t, f0); goto s0; }
s0: if (f0 == l0)                                 goto s2;
    if (r(f1, f0)) {         link_to_tail(t, f1); goto s1; }
    else           {         advance_tail(t, f0); goto s0; }
s1: if (f1 == l1)                                 goto s3;
    if (r(f1, f0)) {         advance_tail(t, f1); goto s1; }
    else           {         link_to_tail(t, f0); goto s0; }
s2: set_link(t, f1); return T(h, t, l1);
s3: set_link(t, f0); return T(h, t, l0);
}
```

Упражнение 8.2. Реализуйте combine_linked, допуская возможность пустых входов. Какое значение должно быть возвращено в качестве итератора, посещенного последним?

Эта процедура также представляет собой конечный автомат. Переменная t указывает на хвост выходного интервала. Эти состояния соответствуют следующим условиям:

s0: $\mathsf{successor}(\mathrm{t}) = \mathrm{f0} \wedge \neg \mathrm{r}(\mathrm{f1}, \mathrm{t})$

s1: $\mathsf{successor}(\mathrm{t}) = \mathrm{f1} \wedge \mathrm{r}(\mathrm{t}, \mathrm{f0})$

Переупорядочение связей необходимо только при перемещении между состояниями s0 и s1.

Лемма 8.3. Если вызов combine_linked_nonempty$(\mathrm{f0}, \mathrm{l0}, \mathrm{f1}, \mathrm{l1}, \mathrm{r}, \mathrm{s})$ возвращает $(\mathrm{h}, \mathrm{t}, \mathrm{l})$, то h равно f0 или f1 и независимо от этого l равно l0 или l1.

Лемма 8.4. Если достигнуто состояние s2, то t берется из исходного интервала $[\mathrm{f0}, \mathrm{l0})$, successor$(\mathrm{t}) = \mathrm{l0}$ и $\mathrm{f1} \neq \mathrm{l1}$; если достигнуто состояние s3, то t берется из исходного интервала $[\mathrm{f1}, \mathrm{l1})$, successor$(\mathrm{t}) = \mathrm{l1}$ и $\mathrm{f0} \neq \mathrm{l0}$.

Лемма 8.5. combine_linked_nonempty является сохраняющим предшествование переупорядочением связей.

Третья машина устанавливает связь с головой списка, а не с его хвостом:

```
template<typename I, typename S>
    requires(ForwardLinker(S) && I == IteratorType(S))
struct linker_to_head
{
    S set_link;
    linker_to_head(const S& set_link) : set_link(set_link) { }
    void operator()(I& h, I& f)
    {
        // Предусловие: successor(f) определено
        IteratorType(S) tmp = successor(f);
        set_link(f, h);
        h = f;
        f = tmp;
    }
};
```

С помощью этой машины можно осуществить обращение интервала итераторов:

```
template<typename I, typename S>
    requires(ForwardLinker(S) && I == IteratorType(S))
I reverse_append(I f, I l, I h, S set_link)
{
    // Предусловие: bounded_range(f, l) ∧ h ∉ [f, l)
    linker_to_head<I, S> link_to_head(set_link);
    while (f != l) link_to_head(h, f);
    return h;
}
```

Чтобы можно было избежать совместного использования подлинных хвостов, h должно быть началом непересекающегося связанного списка (для односвязного списка допустимо значение nil), или l. Разумеется, можно использовать l в качестве начального значения для h (что дает нам reverse_linked), но удобнее передавать отдельный накопительный параметр.

8.3. Области применения переупорядочений связей

Если задан предикат на типе значений типа связанных итераторов, можно использовать split_linked для разделения некоторого интервала. Необходима вспомогательная функция для преобразования предиката на значениях в предикат на итераторах:

```
template<typename I, typename P>
    requires(Readable(I) &&
        Predicate(P) && ValueType(I) == Domain(P))
struct predicate_source
{
    P p;
    predicate_source(const P& p) : p(p) { }
    bool operator()(I i)
    {
        return p(source(i));
    }
};
```

Эта вспомогательная функция позволяет разделить интервал на значения, не удовлетворяющие данному предикату, и значения, которые ему удовлетворяют:

```
template<typename I, typename S, typename P>
    requires(ForwardLinker(S) && I == IteratorType(S) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
pair< pair<I, I>, pair<I, I> >
partition_linked(I f, I l, P p, S set_link)
{
    predicate_source<I, P> ps(p);
    return split_linked(f, l, ps, set_link);
}
```

Если задано нестрогое упорядочение на типе значений типа связанных итераторов, мы можем использовать combine_linked_nonempty для объединения увеличивающихся интервалов. И в этом случае требуется вспомогательная функция для преобразования отношения на значениях в отношение на итераторах:

```
template<typename I0, typename I1, typename R>
    requires(Readable(I0) && Readable(I1) &&
        ValueType(I0) == ValueType(I1) &&
        Relation(R) && ValueType(I0) == Domain(R))
struct relation_source
{
    R r;
```

```
    relation_source(const R& r) : r(r) { }
    bool operator()(I0 i0, I1 i1)
    {
        return r(source(i0), source(i1));
    }
};
```

После объединения интервалов с помощью этого отношения остается лишь найти последний итератор объединенного интервала и задать его равным $\mathrm{l1}$:

```
template<typename I, typename S, typename R>
    requires(Readable(I) &&
        ForwardLinker(S) && I == IteratorType(S) &&
        Relation(R) && ValueType(I) == Domain(R))
pair<I, I> merge_linked_nonempty(I f0, I l0, I f1, I l1,
                                 R r, S set_link)
{
    // Предусловие: f0 ≠ l0 ∧ f1 ≠ l1
    // Предусловие: increasing_range(f0, l0, r)
    // Предусловие: increasing_range(f1, l1, r)
    relation_source<I, I, R> rs(r);
    triple<I, I, I> t = combine_linked_nonempty(f0, l0, f1, l1,
                                                rs, set_link);
    set_link(find_last(t.m1, t.m2), l1);
    return pair<I, I>(t.m0, l1);
}
```

Лемма 8.6. Если $[\mathrm{f0}, \mathrm{l0})$ и $[\mathrm{f1}, \mathrm{l1})$ — непустые увеличивающиеся ограниченные интервалы, то результат их объединения с помощью merge_linked_nonempty представляет собой увеличивающийся ограниченный интервал.

Лемма 8.7. Если $\mathrm{i0} \in [\mathrm{f0}, \mathrm{l0})$ и $\mathrm{i1} \in [\mathrm{f1}, \mathrm{l1})$ — итераторы, значения которых эквивалентны согласно отношению r, то при объединении этих интервалов с помощью merge_linked_nonempty имеет место $\mathrm{i0} \prec \mathrm{i1}$.

Если определена функция merge_linked_nonempty, то несложно реализовать сортировку с объединением:

```
template<typename I, typename S, typename R>
    requires(Readable(I) &&
        ForwardLinker(S) && I == IteratorType(S) &&
        Relation(R) && ValueType(I) == Domain(R))
pair<I, I> sort_linked_nonempty_n(I f, DistanceType(I) n,
                                  R r, S set_link)
{
    // Предусловие: counted_range(f, n) ∧ n > 0 ∧ weak_ordering(r)
    typedef DistanceType(I) N;
```

```
    typedef pair<I, I> P;
    if (n == N(1)) return P(f, successor(f));
    N h = half_nonnegative(n);
    P p0 = sort_linked_nonempty_n(f, h, r, set_link);
    P p1 = sort_linked_nonempty_n(p0.m1, n - h, r, set_link);
    return merge_linked_nonempty(p0.m0, p0.m1,
                                 p1.m0, p1.m1, r, set_link);
}
```

Лемма 8.8. sort_linked_nonempty_n представляет собой переупорядочение связей.

Лемма 8.9. Если $[\![\mathrm{f}, \mathrm{n})\!)$ является непустым счетным интервалом, то sort_linked_nonempty_n переупорядочивает его в увеличивающийся ограниченный интервал.

Сортировка на связанном интервале является *стабильной* относительно нестрогого упорядочения r, если всякий раз, когда итераторы $\mathrm{i} \prec \mathrm{j}$ на входе имеют эквивалентные значения относительно r, на выходе соблюдается условие $\mathrm{i} \prec \mathrm{j}$.

Лемма 8.10. sort_linked_nonempty_n является устойчивым относительно предусмотренного нестрогого упорядочения r.

Упражнение 8.3. Определите формулы для количества применений отношения и объекта формирователя связей в sort_linked_nonempty_n в наихудшем случае и в среднем.

Очевидно, что количество операций, выполняемых sort_linked_nonempty_n, близко к оптимальному, но при плохом расположении ссылок полезность этой функции становится ограниченной, если связанная структура не помещается в кэш-памяти. В таких ситуациях, если доступна дополнительная память, необходимо скопировать связанный список в массив и отсортировать массив.

Сортировка связанного интервала не зависит от predecessor. Сопровождение инварианта

$$\mathrm{i} = \mathsf{predecessor}(\mathsf{successor}(\mathrm{i}))$$

требует применения количества операций обратного связывания, пропорционального количеству сравнений. Можно избежать дополнительной работы, нарушая на время этот инвариант. Предположим, что I — тип связанных двунаправленных итераторов, а forward_linker и backward_linker — соответственно объекты формирователей прямых и обратных связей для I. Мы можем предоставить объект forward_linker для процедуры сортировки (рассматривая список как односвязный), а затем исправлять связи predecessor, применяя backward_linker к каждому итератору, начиная с первого:

```
pair<I, I> p = sort_linked_nonempty_n(f, n,
                                      r, forward_linker);
f = p.m0;
while (f != p.m1) {
    backward_linker(f, successor(f));
    f = successor(f);
};
```

Упражнение 8.4. Реализуйте связанное переупорядочение с сохранением предшествования unique, при котором берется связанный интервал и отношение эквивалентности на типе значений итераторов, а затем вырабатываются два интервала путем перемещения всех итераторов, кроме первого, в любую смежную последовательность итераторов с эквивалентными значениями во второй интервал.

8.4. Связанные бифуркатные координаты

Если разрешается внесение изменений в функцию successor, то появляется возможность создания алгоритмов переупорядочения связей, таких как объединение и разбиение. Полезно иметь изменяемые функции обхода для других координатных структур. Проиллюстрируем эту идею с помощью связанных бифуркатных координат.

Применяя связанные итераторы, мы передавали операцию связывания в качестве параметра, поскольку необходимо было использовать разные операции связывания: например, при восстановлении обратных связей после сортировки. Что касается связанных бифуркатных координат, то, по-видимому, нет необходимости применять альтернативные версии операций связывания, поэтому мы определим их в концепции:

$$
\begin{aligned}
&\mathit{LinkedBifurcateCoordinate}(\mathsf{T}) \triangleq \\
&\qquad \mathit{BifurcateCoordinate}(\mathsf{T}) \\
&\quad \wedge\ \mathsf{set\_left\_successor} : \mathsf{T} \times \mathsf{T} \to \mathsf{void} \\
&\qquad\qquad (\mathtt{i}, \mathtt{j}) \mapsto \text{ устанавливает } \mathsf{left\_successor}(\mathtt{i}) = \mathtt{j} \\
&\quad \wedge\ \mathsf{set\_right\_successor} : \mathsf{T} \times \mathsf{T} \to \mathsf{void} \\
&\qquad\qquad (\mathtt{i}, \mathtt{j}) \mapsto \text{ устанавливает } \mathsf{right\_successor}(\mathtt{i}) = \mathtt{j}
\end{aligned}
$$

Область определения для set_left_successor и set_right_successor представляет собой множество непустых координат.

Деревья составляют разнообразный набор возможных структур данных и алгоритмов. В качестве заключения в этой главе приведем небольшой набор алгоритмов для иллюстрации важного метода программирования. Этот метод, называемый *обращением связей*, предусматривает изменение связей во время обхода дерева и восстановление исходного состояния после полного обхода,

для чего требуется лишь постоянное дополнительное пространство. Для обращения связей требуются дополнительные аксиомы, которые позволяют иметь дело с *пустыми* координатами: теми, на которых функции обхода не определены:

$$
\begin{aligned}
&\mathit{EmptyLinkedBifurcateCoordinate}(\mathsf{T}) \triangleq \\
&\quad \mathit{LinkedBifurcateCoordinate}(\mathsf{T}) \\
&\quad \wedge\ \mathsf{empty}(\mathsf{T}())^2 \\
&\quad \wedge\ \neg\mathsf{empty}(\mathtt{i}) \Rightarrow \\
&\qquad \mathsf{left\_successor}(\mathtt{i}) \text{ и } \mathsf{right\_successor}(\mathtt{i}) \text{ определены} \\
&\quad \wedge\ \neg\mathsf{empty}(\mathtt{i}) \Rightarrow \\
&\qquad (\neg\mathsf{has\_left\_successor}(\mathtt{i}) \Leftrightarrow \mathsf{empty}(\mathsf{left\_successor}(\mathtt{i}))) \\
&\quad \wedge\ \neg\mathsf{empty}(\mathtt{i}) \Rightarrow \\
&\qquad (\neg\mathsf{has\_right\_successor}(\mathtt{i}) \Leftrightarrow \mathsf{empty}(\mathsf{right\_successor}(\mathtt{i})))
\end{aligned}
$$

traverse_step из главы 7 является эффективным способом осуществления обхода с помощью двунаправленных бифуркатных координат, но требует применения функции predecessor. Если функция predecessor недоступна и рекурсивный обход (на основе стека) неприемлем из-за того, что деревья несбалансированы, может использоваться обращение связей для сохранения на время связей с предшественником в связи, обычно содержащей последователя, гарантируя тем самым, что есть путь назад к корню[3].

Если мы будем рассматривать левого и правого последователей узла дерева вместе с координатой предыдущего узла дерева как составляющие тройку, то сможем выполнять обращение этих трех элементов тройки с помощью данной машины:

```
template<typename C>
     requires(EmptyLinkedBifurcateCoordinate(C))
void tree_rotate(C& curr, C& prev)
{
    // Предусловие: ¬empty(curr)
    C tmp = left_successor(curr);
    set_left_successor(curr, right_successor(curr));
    set_right_successor(curr, prev);
    if (empty(tmp)) { prev = tmp; return; }
    prev = curr;
    curr = tmp;
}
```

[2]Иными словами, функция empty равна true на стандартных сконструированных значениях, а также, возможно, на других значениях.

[3]Обращение связей было представлено в [Schorr and Waite 1967] и независимо открыто Л. П. Дойчем (L. P. Deutsch). Версия без маркирующих битов была опубликована в [Robson 1973] и в [Morris 1979]. Мы показываем метод вращения связей, который представлен в [Lindstrom 1973] и независимо от этого в [Dwyer 1974].

Повторные применения tree_rotate позволяют осуществить обход всего дерева:

```
template<typename C, typename Proc>
    requires(EmptyLinkedBifurcateCoordinate(C) &&
        Procedure(Proc) && Arity(Proc) == 1 &&
        C == InputType(Proc, 0))
Proc traverse_rotating(C c, Proc proc)
{
    // Предусловие: tree(c)
    if (empty(c)) return proc;
    C curr = c;
    C prev;
    do {
        proc(curr);
        tree_rotate(curr, prev);
    } while (curr != c);
    do {
        proc(curr);
        tree_rotate(curr, prev);
    } while (curr != c);
    proc(curr);
    tree_rotate(curr, prev);
    return proc;
}
```

Теорема 8.1. Рассмотрим вызов traverse_rotating$(c, proc)$ и любого непустого потомка i для c, где i имеет начальных левого и правого последователей, l и r, и предшественника p. В таком случае:

1. Левые и правые последователи i проходят три преобразования:

$$(l, r) \overset{pre}{\to} (r, p) \overset{in}{\to} (p, l) \overset{post}{\to} (l, r)$$

2. Если n_l и n_r — веса l и r, то преобразования $(r, p) \overset{in}{\to} (p, l)$ и $(p, l) \overset{post}{\to} (l, r)$ принимают вызовы $3n_l + 1$ и $3n_r + 1$ функции tree_rotate соответственно.
3. Если k — число вызовов tree_rotate, то значение $k \bmod 3$ является различным для каждого из трех преобразований последователей i.
4. Во время вызова traverse_rotating$(c, proc)$ общее количество вызовов tree_rotate равно $3n$, где n — вес c.

Доказательство. По индукции по n, весу c. □

Упражнение 8.5. Нарисуйте диаграммы каждого состояния обхода с помощью traverse_rotating полного бинарного дерева с семью узлами.

traverse_rotating выполняет ту же последовательность посещений preorder, inorder и postorder, как и функция traverse_nonempty из главы 7. К сожалению, нам неизвестно, как определить, является ли конкретное посещение координаты посещением pre, in или post. Тем не менее с помощью traverse_rotating мы все еще имеем возможность вычислить полезные данные, такие как вес дерева:

```
template<typename T, typename N>
    requires(Integer(N))
struct counter {
    N n;
    counter() : n(0) { }
    counter(N n) : n(n) { }
    void operator()(const T&) { n = successor(n); }
};

template<typename C>
    requires(EmptyLinkedBifurcateCoordinate(C))
WeightType(C) weight_rotating(C c)
{
    // Предусловие: tree(c)
    typedef WeightType(C) N;
    return traverse_rotating(c, counter<C, N>()).n / N(3);
}
```

Можно посетить каждую координату только один раз, подсчитывая посещения по модулю 3:

```
template<typename N, typename Proc>
    requires(Integer(N) &&
        Procedure(Proc) && Arity(Proc) == 1)
struct phased_applicator
{
    N period;
    N phase;
    N n;
    // Инвариант: n, phase ∈ [0, period)
    Proc proc;
    phased_applicator(N period, N phase, N n, Proc proc) :
        period(period), phase(phase), n(n), proc(proc) { }
    void operator()(InputType(Proc, 0) x)
    {
        if (n == phase) proc(x);
        n = successor(n);
        if (n == period) n = 0;
    }
};
```

```
template<typename C, typename Proc>
    requires(EmptyLinkedBifurcateCoordinate(C) &&
        Procedure(Proc) && Arity(Proc) == 1 &&
        C == InputType(Proc, 0))
Proc traverse_phased_rotating(C c, int phase, Proc proc)
{
    // Предусловие: tree(c) ∧ 0 ⩽ phase < 3
    phased_applicator<int, Proc> applicator(3, phase, 0, proc);
    return traverse_rotating(c, applicator).proc;
}
```

Проект 8.1. Используйте tree_rotate для реализации свойств изоморфизма, эквивалентности и упорядочения на бинарных деревьях.

8.5. Резюме

Связанные координатные структуры с изменяемыми функциями обхода позволяют создавать полезные алгоритмы переупорядочения, такие как сортировка связанных интервалов. Подобные алгоритмы можно составлять систематически из простых машин, что приводит к получению эффективного кода с точно определенными математическими свойствами. Продуманное использование goto является допустимым способом реализации конечных автоматов. Инварианты, распространяющиеся больше чем на один объект, могут на время нарушаться при обновлении одного из объектов. Алгоритм определяет область, в которой допускается нарушение инвариантов, если гарантируется, что они будут восстановлены перед выходом из области.

Глава 9
Копирование

В настоящей главе приведено вводное описание записываемых итераторов, функции доступа которых позволяют изменять значения итераторов. Для иллюстрации использования записываемых итераторов будет применяться семейство алгоритмов копирования, созданных на основе простых машин, которые копируют один объект и обновляют входные и выходные итераторы. Тщательно продуманная спецификация предусловий допускает, чтобы входные и выходные интервалы перекрывались во время копирования. Если два неперекрывающихся интервала одного и того же размера являются изменяемыми, то для обмена их содержимого может использоваться семейство алгоритмов взаимной перестановки.

9.1. Записываемость

В настоящей главе обсуждается вторая разновидность доступа к итераторам и другим координатным структурам — записываемость. Тип является *записываемым*, если на нем определена унарная процедура sink; sink может использоваться только на левой стороне оператора присваивания, правая сторона которого принимает значение объекта ValueType(T):

$Writable(\mathsf{T}) \triangleq$

ValueType : $Writable \to Regular$

$\wedge\ (\forall \mathrm{x} \in \mathsf{T})\,(\forall \mathrm{v} \in \mathsf{ValueType}(\mathsf{T}))\,\mathsf{sink}(\mathrm{x}) \leftarrow \mathrm{v}$ — формально правильная инструкция

Единственный способ использования sink(x), обоснованный концепцией *Writable*, состоит в ее размещении в левой стороне оператора присваивания. Разумеется, могут быть предусмотрены и другие способы использования этой процедуры, поддерживаемые конкретным типом, моделирующим *Writable*.

sink не должна быть всеобъемлющей; не исключено наличие объектов записываемого типа, на которых sink не определена. Как и в отношении чи-

таемости, в этой концепции не предусмотрен предикат области определения, который позволил бы выяснить, определена ли процедура sink для конкретного объекта. Допустимость ее использования в алгоритме должна быть выводимой из предусловий.

Для определенного состояния объекта x должно быть обосновано только однократное присваивание значения процедуре $\mathsf{sink}(x)$ в соответствии с концепцией *Writable*; для того или иного конкретного типа может быть предусмотрен некоторый протокол, допускающий последующие присваивания значений процедуре $\mathsf{sink}(x)$[1].

Записываемый объект x и читаемый объект y являются *обозначенными псевдонимом*, если определены и $\mathsf{sink}(x)$, и $\mathsf{source}(y)$, а присваивание любого значения v процедуре $\mathsf{sink}(x)$ вынуждает ее обнаруживаться в качестве значения $\mathsf{source}(y)$:

property(T : *Writable*, U : *Readable*)
  requires($\mathsf{ValueType}(T) = \mathsf{ValueType}(U)$)
$\mathsf{aliased} : T \times U$
  $(x, y) \mapsto \mathsf{sink}(x)$ определено $\wedge$
    $\mathsf{source}(y)$ определено $\wedge$
    $(\forall v \in \mathsf{ValueType}(T))\ \mathsf{sink}(x) \leftarrow v$ устанавливает $\mathsf{source}(y) = v$

Последней разновидностью доступа является *изменяемость*, при которой читаемость и записываемость сочетаются непротиворечивым способом:

$\mathit{Mutable}(T) \triangleq$
  $\mathit{Readable}(T) \wedge \mathit{Writable}(T)$
  $\wedge\ (\forall x \in T)\ \mathsf{sink}(x)$ определено $\Leftrightarrow$ $\mathsf{source}(x)$ определено
  $\wedge\ (\forall x \in T)\ \mathsf{sink}(x)$ определено $\Rightarrow$ $\mathsf{aliased}(x, x)$
  $\wedge\ \mathsf{deref} : T \to \mathsf{ValueType}(T)\&$
  $\wedge\ (\forall x \in T)\ \mathsf{sink}(x)$ определено $\Leftrightarrow$ $\mathsf{deref}(x)$ определено

Что касается изменяемого итератора, то замена $\mathsf{source}(x)$ или $\mathsf{sink}(x)$ на $\mathsf{deref}(x)$ не затрагивает смысл или производительность программы.

Интервал итераторов, которые относятся к типу, моделирующему *Writable* и *Iterator*, является *записываемым*, если процедура sink определена на всех итераторах в интервале:

property(I : *Writable*)
  requires(*Iterator*(I))
writable_bounded_range : I × I
  $(f, l) \mapsto \mathsf{bounded\_range}(f, l) \wedge (\forall i \in [f, l))\ \mathsf{sink}(i)$ определено

writable_weak_range и writable_counted_range определены аналогичным образом.

[1] Джерри Шварц (Jerry Schwarz) предлагает потенциально более изящный интерфейс: замену sink процедурой store, такой, что вызов $\mathsf{store}(v, x)$ эквивалентен $\mathsf{sink}(x) \leftarrow v$.

При наличии читаемого итератора i процедура source(i) может вызываться неоднократно, при этом всегда возвращая одинаковое значение, поскольку она является регулярной. Это позволяет писать простые, полезные алгоритмы, такие как find_if. Однако, что касается записываемого итератора j, то присваивание значения процедуре sink(j) не является повторимым: в вызове successor должны быть разделены два присваивания через итератор. Асимметрия между читаемым и записываемым итераторами введена намеренно: складывается впечатление, что она не устраняет полезные алгоритмы и допускает применение моделей, не являющихся буферизированными, таких как выходные потоки. Благодаря наличию нерегулярной процедуры successor в концепции *Iterator* и нерегулярной процедуры sink появляется возможность использовать алгоритмы по отношению к входным и выходным потокам, а не только к структурам данных в оперативной памяти.

Интервал итераторов, которые относятся к типу, моделирующему *Mutable* и *ForwardIterator*, является *изменяемым*, если процедуры sink, а следовательно source и deref, определены на всех итераторах в интервале. Чтение и запись в одном и том же интервале могут осуществлять только многопроходные алгоритмы. Это означает, что для изменяемых интервалов требуются по меньшей мере прямые итераторы, и мы опускаем требование, согласно которому два присваивания итератору должны быть разделены вызовом процедуры successor:

property(I : *Mutable*)
  requires(*ForwardIterator*(I))
mutable_bounded_range : I × I
  (f, l) ↦ bounded_range(f, l) ∧ (∀i ∈ [f, l)) sink(i) определено

mutable_weak_range и mutable_counted_range определены аналогичным образом.

9.2. Копирование с учетом позиции

Представим семейство алгоритмов копирования объектов из одного или нескольких входных интервалов в один или несколько выходных интервалов. Вообще говоря, постусловия этих алгоритмов определяют равенство между объектами в выходных интервалах и исходными значениями объектов во входных интервалах. Если входные и выходные интервалы не *перекрываются*, то несложно установить желаемое постусловие. Но часто возникает необходимость копировать объекты между перекрывающимися интервалами, поэтому предусловие для каждого алгоритма указывает, какое перекрытие является допустимым.

Основное правило для перекрытия состоит в том, что если итератор во входном интервале связан псевдонимом с итератором в выходном интервале, то алгоритм может не применять процедуру source к входному итератору

после применения процедуры sink к выходному итератору. Мы разработаем точные условия и общие свойства для их выражения по ходу представления алгоритмов.

Все машины, из которых мы составляем алгоритмы копирования, принимают два итератора по ссылке и отвечают за копирование и обновление итераторов. Наиболее часто используемая машина копирует один объект, а затем наращивает оба итератора:

```
template<typename I, typename O>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O))
void copy_step(I& f_i, O& f_o)
{
    // Предусловие: source(f_i) и sink(f_o) определены
    sink(f_o) = source(f_i);
    f_i = successor(f_i);
    f_o = successor(f_o);
}
```

Общая форма алгоритмов копирования предусматривает выполнение шага копирования до тех пор, пока не удовлетворено условие завершения. Например, copy копирует полуоткрытый ограниченный интервал в выходной интервал, определенный его первым итератором:

```
template<typename I, typename O>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O))
O copy(I f_i, I l_i, O f_o)
{
    // Предусловие: not_overlapped_forward(f_i, l_i, f_o, f_o + (l_i - f_i))
    while (f_i != l_i) copy_step(f_i, f_o);
    return f_o;
}
```

copy возвращает предел выходного интервала, поскольку он может оказаться неизвестным для вызывающего объекта. Тип выходного итератора может не допускать многочисленные обходы, и в этом случае, если не был бы возвращен предел, то отсутствовала бы возможность восстановления.

Постусловие для copy состоит в том, что последовательность значений в выходном интервале равна исходной последовательности значений во входном интервале. Чтобы удовлетворялось это постусловие, предусловие должно гарантировать соответственно читаемость и записываемость входного и выходного интервалов; достаточный размер выходного интервала; и если входной

и выходной интервалы перекрываются, то отсутствие чтения входного итератора после записи связанного с помощью псевдонима выходного итератора. Эти условия формализованы с помощью свойства not_overlapped_forward. Читаемый и записываемый интервалы не являются *перекрывающимися вперед*, если в индексе во входном интервале обнаруживаются какие-либо связанные псевдонимом итераторы, которые не превышают индекс в выходном интервале:

property$(\mathrm{I} : \mathit{Readable}, \mathrm{O} : \mathit{Writable})$
  requires$(\mathit{Iterator}(\mathrm{I}) \wedge \mathit{Iterator}(\mathrm{O}))$
$\mathsf{not\_overlapped\_forward} : \mathrm{I} \times \mathrm{I} \times \mathrm{O} \times \mathrm{O}$
  $(\mathrm{f_i}, \mathrm{l_i}, \mathrm{f_o}, \mathrm{l_o}) \mapsto$
    $\mathsf{readable\_bounded\_range}(\mathrm{f_i}, \mathrm{l_i}) \wedge$
    $\mathsf{writable\_bounded\_range}(\mathrm{f_o}, \mathrm{l_o}) \wedge$
    $(\forall \mathrm{k_i} \in [\mathrm{f_i}, \mathrm{l_i}))(\forall \mathrm{k_o} \in [\mathrm{f_o}, \mathrm{l_o}))$
      $\mathsf{aliased}(\mathrm{k_o}, \mathrm{k_i}) \Rightarrow \mathrm{k_i} - \mathrm{f_i} \leqslant \mathrm{k_o} - \mathrm{f_o}$

Иногда размеры входного и выходного интервалов могут быть разными:

```
template<typename I, typename O>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O))
pair<I, O> copy_bounded(I f_i, I l_i, O f_o, O l_o)
{
    // Предусловие: not_overlapped_forward(f_i, l_i, f_o, l_o)
    while (f_i != l_i && f_o != l_o) copy_step(f_i, f_o);
    return pair<I, O>(f_i, f_o);
}
```

Поскольку концы обоих интервалов известны вызывающему объекту, возврат этой пары позволяет вызывающему объекту определить, какой интервал меньше и где в большем интервале остановлено копирование. По сравнению с copy ослаблено выходное предусловие: выходной интервал может быть короче входного интервала. Можно даже утверждать, что самым нестрогим предусловием должно быть следующее:

$$\mathsf{not\_overlapped\_forward}(\mathrm{f_i}, \mathrm{f_i} + \mathrm{n}, \mathrm{f_o}, \mathrm{f_o} + \mathrm{n})$$

где $\mathrm{n} = \min(\mathrm{l_i} - \mathrm{f_i}, \mathrm{l_o} - \mathrm{f_o})$.

Эти вспомогательные машины обрабатывают условие завершения для счетных интервалов:

```
template<typename N>
    requires(Integer(N))
bool count_down(N& n)
{
    // Предусловие: n ⩾ 0
```

```
    if (zero(n)) return false;
    n = predecessor(n);
    return true;
}
```

copy_n копирует полуоткрытый счетный интервал в выходной интервал, определенный его первым итератором:

```
template<typename I, typename O, typename N>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O) &&
        Integer(N))
pair<I, O> copy_n(I f_i, N n, O f_o)
{
    // Предусловие: not_overlapped_forward(f_i, f_{i+n}, f_o, f_{o+n})
    while (count_down(n)) copy_step(f_i, f_o);
    return pair<I, O>(f_i, f_o);
}
```

Результат применения copy_bounded к двум счетным интервалам получен путем вызова copy_n, самое меньшее, с двумя размерами.

Если интервалы перекрываются в прямом направлении, все еще остается возможность копирования, при условии, что типы итераторов моделируют *BidirectionalIterator* и поэтому обеспечивают движение в обратном направлении. Это ведет к созданию следующей машины:

```
template<typename I, typename O>
    requires(Readable(I) && BidirectionalIterator(I) &&
        Writable(O) && BidirectionalIterator(O) &&
        ValueType(I) == ValueType(O))
void copy_backward_step(I& l_i, O& l_o)
{
    // Предусловие: source(predecessor(l_i)) и sink(predecessor(l_o))
    //              определены
    l_i = predecessor(l_i);
    l_o = predecessor(l_o);
    sink(l_o) = source(l_i);
}
```

Поскольку мы имеем дело с полуоткрытыми интервалами и начинаем с предела, следует уменьшить значение итератора перед копированием, что ведет к созданию copy_backward:

```
template<typename I, typename O>
    requires(Readable(I) && BidirectionalIterator(I) &&
        Writable(O) && BidirectionalIterator(O) &&
        ValueType(I) == ValueType(O))
```

```
O copy_backward(I f_i, I l_i, O l_o)
{
    // Предусловие: not_overlapped_backward(f_i, l_i, l_o − (l_i − f_i), l_o)
    while (f_i != l_i) copy_backward_step(l_i, l_o);
    return l_o;
}
```

copy_backward_n является аналогичным.

Постусловие для copy_backward аналогично таковому для copy и формализовано с помощью свойства not_overlapped_backward. Читаемый и записываемый интервалы не являются *перекрывающимися назад*, если в индексе из предела входного интервала обнаруживаются какие-либо связанные псевдонимом итераторы, которые не превышают индекса из предела выходного интервала:

property(I : *Readable*, O : *Writable*)
  requires(*Iterator*(I) ∧ *Iterator*(O))
not_overlapped_backward : I × I × O × O
  $(f_i, l_i, f_o, l_o) \mapsto$
    readable_bounded_range(f_i, l_i) ∧
    writable_bounded_range(f_o, l_o) ∧
    $(\forall k_i \in [f_i, l_i))(\forall k_o \in [f_o, l_o))$
      aliased$(k_o, k_i) \Rightarrow l_i - k_i \leqslant l_o - k_o$

Если один из интервалов имеет тип итератора, моделирующего *BidirectionalIterator*, то можно изменить на обратное направление выходного интервала по отношению к входному интервалу с использованием машины, которая движется в обратном направлении в выходном интервале, или машины, которая движется в обратном направлении во входном интервале:

```
template<typename I, typename O>
    requires(Readable(I) && BidirectionalIterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O))
void reverse_copy_step(I& l_i, O& f_o)
{
    // Предусловие: source(predecessor(l_i)) и sink(f_o) определены
    l_i = predecessor(l_i);
    sink(f_o) = source(l_i);
    f_o = successor(f_o);
}

template<typename I, typename O>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && BidirectionalIterator(O) &&
        ValueType(I) == ValueType(O))
```

```
void reverse_copy_backward_step(I& f_i, O& l_o)
{
    // Предусловие: source(f_i) и sink(predecessor(l_o)) определены
    l_o = predecessor(l_o);
    sink(l_o) = source(f_i);
    f_i = successor(f_i);
}
```

что ведет к следующим алгоритмам:

```
template<typename I, typename O>
    requires(Readable(I) && BidirectionalIterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O))
O reverse_copy(I f_i, I l_i, O f_o)
{
    // Предусловие: not_overlapped(f_i, l_i, f_o, f_o + (l_i - f_i))
    while (f_i != l_i) reverse_copy_step(l_i, f_o);
    return f_o;
}

template<typename I, typename O>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && BidirectionalIterator(O) &&
        ValueType(I) == ValueType(O))
O reverse_copy_backward(I f_i, I l_i, O l_o)
{
    // Предусловие: not_overlapped(f_i, l_i, l_o - (l_i - f_i), l_o)
    while (f_i != l_i) reverse_copy_backward_step(f_i, l_o);
    return l_o;
}
```

reverse_copy_n и reverse_copy_backward_n аналогичны.

Постусловие и для reverse_copy, и для reverse_copy_backward состоит в том, что выходной интервал является обращенной копией исходной последовательности значений входного интервала. Практически применимое, но не самое нестрогое предусловие состоит в том, что входной и выходной интервалы не перекрываются, что можно формально представить с помощью свойства not_overlapped. Читаемый и записываемый интервалы не *перекрываются*, если они не имеют общих связанных псевдонимами итераторов:

property$(\mathrm{I} : \mathit{Readable}, \mathrm{O} : \mathit{Writable})$
  requires$(\mathsf{Iterator}(\mathrm{I}) \wedge \mathsf{Iterator}(\mathrm{O}))$
$\mathsf{not\_overlapped} : \mathrm{I} \times \mathrm{I} \times \mathrm{O} \times \mathrm{O}$
  $(\mathrm{f_i}, \mathrm{l_i}, \mathrm{f_o}, \mathrm{l_o}) \mapsto$
    $\mathsf{readable\_bounded\_range}(\mathrm{f_i}, \mathrm{l_i}) \wedge$

$$\mathsf{writable\_bounded\_range}(f_o, l_o) \wedge$$
$$(\forall k_i \in [f_i, l_i))\,(\forall k_o \in [f_o, l_o))\,\neg\mathsf{aliased}(k_o, k_i)$$

Упражнение 9.1. Найдите самые нестрогие предусловия для процедуры reverse_copy и сопутствующей ей процедуры reverse_copy_backward.

Разумеется, главной причиной для введения copy_backward, а также copy является обработка интервалов, перекрывающихся в любом направлении, но причина для введения reverse_copy_backward, а также reverse_copy состоит в том, чтобы обеспечить бóльшую гибкость с точки зрения требований к итераторам.

9.3. Копирование на основе предиката

В алгоритмах, представленных до сих пор, каждый объект во входном интервале копируется в выходной интервал, а их постусловия не зависят от значения какого-либо итератора. Алгоритмы в этом разделе принимают в качестве аргумента предикат и используют его для управления каждым шагом копирования.

Например, выполнение шага копирования по условию, определяемому унарным предикатом, ведет к получению процедуры copy_select:

```
template<typename I, typename O, typename P>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
        ValueType(I) == ValueType(O) &&
        UnaryPredicate(P) && I == Domain(P))
O copy_select(I f_i, I l_i, O f_t, P p)
{
    // Предусловие: not_overlapped_forward(f_i, l_i, f_t, f_t + n_t)
    // где n_t — верхняя граница для числа итераторов, удовлетворяющих p
    while (f_i != l_i)
        if (p(f_i)) copy_step(f_i, f_t);
        else f_i = successor(f_i);
    return f_t;
}
```

Наихудшим случаем для n_t является $l_i - f_i$; контекст мог бы гарантировать меньшее значение.

В наиболее распространенном случае предикат применяется не к итератору, а к его значению:

```
template<typename I, typename O, typename P>
    requires(Readable(I) && Iterator(I) &&
        Writable(O) && Iterator(O) &&
```

```
        ValueType(I) == ValueType(O) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
O copy_if(I f_i, I l_i, O f_t, P p)
{
    // Предусловие: то же, что и для copy_select
    predicate_source<I, P> ps(p);
    return copy_select(f_i, l_i, f_t, ps);
}
```

В главе 8 представлены процедуры split_linked и combine_linked_nonempty, оперирующие на связанных интервалах итераторов. Есть и аналогичные алгоритмы копирования:

```
template<typename I, typename O_f, typename O_t, typename P>
    requires(Readable(I) && Iterator(I) &&
        Writable(O_f) && Iterator(O_f) &&
        Writable(O_t) && Iterator(O_t) &&
        ValueType(I) == ValueType(O_f) &&
        ValueType(I) == ValueType(O_t) &&
        UnaryPredicate(P) && I == Domain(P))
pair<O_f, O_t> split_copy(I f_i, I l_i, O_f f_f, O_t f_t,
                          P p)
{
    // Предусловие: см. ниже
    while (f_i != l_i)
        if (p(f_i)) copy_step(f_i, f_t);
        else        copy_step(f_i, f_f);
    return pair<O_f, O_t>(f_f, f_t);
}
```

Упражнение 9.2. Напишите постусловие для split_copy.

Чтобы удовлетворялось его постусловие, вызов split_copy должен гарантировать, что эти два выходных интервала вообще не перекрываются. Допустимо, чтобы любой из выходных интервалов перекрывал входной интервал, при условии, что интервалы не перекрываются в прямом направлении. Это приводит к получению следующего предусловия:

$$\begin{aligned}&\mathsf{not\_write\_overlapped}(f_f, n_f, f_t, n_t) \wedge \\ &\quad((\mathsf{not\_overlapped\_forward}(f_i, l_i, f_f, f_f + n_f) \wedge \mathsf{not\_overlapped}(f_i, l_i, f_t, l_t)) \vee \\ &\qquad(\mathsf{not\_overlapped\_forward}(f_i, l_i, f_t, f_t + n_t) \wedge \mathsf{not\_overlapped}(f_i, l_i, f_f, l_f)))\end{aligned}$$

где n_f и n_t — верхние границы для количества итераторов, не удовлетворяющих и удовлетворяющих p соответственно.

Определение свойства not_write_overlapped зависит от понятия *применения псевдонимов при записи*: два записываемых объекта, x и y, таких, что

определены и $\mathsf{sink}(x)$, и $\mathsf{sink}(y)$, и любой наблюдатель результатов записи в x наблюдает также результаты записи в y:

property$(T : \mathit{Writable}, U : \mathit{Writable})$
  requires$(\mathsf{ValueType}(T) = \mathsf{ValueType}(U))$
$\mathsf{write\_aliased} : T \times U$
  $(x, y) \mapsto \mathsf{sink}(x)$ определено $\wedge$ $\mathsf{sink}(y)$ определено $\wedge$
    $(\forall V \in \mathit{Readable})\,(\forall v \in V)\,\mathsf{aliased}(x, v) \Leftrightarrow \mathsf{aliased}(y, v)$

Это приводит к определению *не перекрывающихся при записи*, или записываемых интервалов, которые не имеют общих стоков, связанных псевдонимами:

property$(O_0 : \mathit{Writable}, O_1 : \mathit{Writable})$
  requires$(\mathsf{Iterator}(O_0) \wedge \mathsf{Iterator}(O_1))$
$\mathsf{not\_write\_overlapped} : O_0 \times O_0 \times O_1 \times O_1$
  $(f_0, l_0, f_1, l_1) \mapsto$
    $\mathsf{writable\_bounded\_range}(f_0, l_0) \wedge$
    $\mathsf{writable\_bounded\_range}(f_1, l_1) \wedge$
    $(\forall k_0 \in [f_0, l_0))(\forall k_1 \in [f_1, l_1)) \neg \mathsf{write\_aliased}(k_0, k_1)$

Как и в отношении select_copy, предикат в наиболее распространенном случае применения split_copy применяется не к итератору, а к его значению[2]:

```
template<typename I, typename O_f, typename O_t, typename P>
    requires(Readable(I) && Iterator(I) &&
        Writable(O_f) && Iterator(O_f) &&
        Writable(O_t) && Iterator(O_t) &&
        ValueType(I) == ValueType(O_f) &&
        ValueType(I) == ValueType(O_t) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
pair<O_f, O_t> partition_copy(I f_i, I l_i, O_f f_f, O_t f_t,
                              P p)
{
    // Предусловие: то же, что и для split_copy
    predicate_source<I, P> ps(p);
    return split_copy(f_i, l_i, f_f, f_t, ps);
}
```

Значения в каждом из этих двух выходных интервалов находятся в одном и том же относительном порядке, как и во входном интервале; то же касается и partition_copy_n.

Код для combine_copy столь же прост:

```
template<typename I0, typename I1, typename O, typename R>
    requires(Readable(I0) && Iterator(I0) &&
```

[2]Этот интерфейс предложен Т. К. Лакшманом (T. K. Lakshman).

```
        Readable(I1) && Iterator(I1) &&
        Writable(O) && Iterator(O) &&
        BinaryPredicate(R) &&
        ValueType(I0) == ValueType(O) &&
        ValueType(I1) == ValueType(O) &&
        I0 == InputType(R, 1) && I1 == InputType(R, 0))
O combine_copy(I0 f_i0, I0 l_i0, I1 f_i1, I1 l_i1, O f_o, R r)
{
    // Предусловие: см. ниже
    while (f_i0 != l_i0 && f_i1 != l_i1)
        if (r(f_i1, f_i0))  copy_step(f_i1, f_o);
        else                copy_step(f_i0, f_o);
    return copy(f_i1, l_i1, copy(f_i0, l_i0, f_o));
}
```

Что касается combine_copy, то чтение с перекрытием между входными интервалами является приемлемым. Кроме того, допустимо, чтобы один из входных интервалов перекрывался с выходным интервалом, но такое перекрытие не может осуществляться в прямом направлении и должно быть возмещено в обратном направлении, по меньшей мере, на размер другого входного интервала, как описано в свойстве backward_offset, используемом в предусловии combine_copy:

$$(\mathsf{backward\_offset}(\mathrm{f_{i_0}}, \mathrm{l_{i_0}}, \mathrm{f_o}, \mathrm{l_o}, \mathrm{l_{i_1}} - \mathrm{f_{i_1}}) \wedge \mathsf{not\_overlapped}(\mathrm{f_{i_1}}, \mathrm{l_{i_1}}, \mathrm{f_o}, \mathrm{l_o})) \vee$$
$$(\mathsf{backward\_offset}(\mathrm{f_{i_1}}, \mathrm{l_{i_1}}, \mathrm{f_o}, \mathrm{l_o}, \mathrm{l_{i_0}} - \mathrm{f_{i_0}}) \wedge \mathsf{not\_overlapped}(\mathrm{f_{i_0}}, \mathrm{l_{i_0}}, \mathrm{f_o}, \mathrm{l_o}))$$

где $\mathrm{l_o} = \mathrm{f_o} + (\mathrm{l_{i_0}} - \mathrm{f_{i_0}}) + (\mathrm{l_{i_1}} - \mathrm{f_{i_1}})$ — предел выходного интервала.

Свойство backward_offset удовлетворяется читаемым интервалом, записываемым интервалом и смещением $\mathrm{n} \geqslant 0$, если какие-либо связанные псевдонимом итераторы обнаруживаются в индексе во входном интервале, который после увеличения n не превышает индекса в выходном интервале:

property$(\mathrm{I} : \mathit{Readable}, \mathrm{O} : \mathit{Writable}, \mathrm{N} : \mathit{Integer})$
  requires$(\mathit{Iterator}(\mathrm{I}) \wedge \mathit{Iterator}(\mathrm{O}))$
backward_offset : $\mathrm{I} \times \mathrm{I} \times \mathrm{O} \times \mathrm{O} \times \mathrm{N}$
  $(\mathrm{f_i}, \mathrm{l_i}, \mathrm{f_o}, \mathrm{l_o}, \mathrm{n}) \mapsto$
    $\mathsf{readable\_bounded\_range}(\mathrm{f_i}, \mathrm{l_i}) \wedge$
    $\mathrm{n} \geqslant 0 \wedge$
    $\mathsf{writable\_bounded\_range}(\mathrm{f_o}, \mathrm{l_o}) \wedge$
    $(\forall \mathrm{k_i} \in [\mathrm{f_i}, \mathrm{l_i}))(\forall \mathrm{k_o} \in [\mathrm{f_o}, \mathrm{l_o}))$
      $\mathsf{aliased}(\mathrm{k_o}, \mathrm{k_i}) \Rightarrow \mathrm{k_i} - \mathrm{f_i} + \mathrm{n} \leqslant \mathrm{k_o} - \mathrm{f_o}$

Следует отметить, что имеет место $\mathsf{not\_overlapped\_forward}(\mathrm{f_i}, \mathrm{l_i}, \mathrm{f_o}, \mathrm{l_o}) = \mathsf{backward\_offset}(\mathrm{f_i}, \mathrm{l_i}, \mathrm{f_o}, \mathrm{l_o}, 0)$.

Упражнение 9.3. Запишите постусловие для combine_copy и докажите, что оно удовлетворяется каждый раз, когда соблюдается предусловие.

combine_copy_backward является аналогичным. Чтобы можно было гарантировать соблюдение того же постусловия, порядок предложений `if` должен быть обратным по отношению к порядку в combine_copy:

```
template<typename I0, typename I1, typename O, typename R>
    requires(Readable(I0) && BidirectionalIterator(I0) &&
        Readable(I1) && BidirectionalIterator(I1) &&
        Writable(O) && BidirectionalIterator(O) &&
        BinaryPredicate(R) &&
        ValueType(I0) == ValueType(O) &&
        ValueType(I1) == ValueType(O) &&
        I0 == InputType(R, 1) && I1 == InputType(R, 0))
O combine_copy_backward(I0 f_i0, I0 l_i0, I1 f_i1, I1 l_i1,
                        O l_o, R r)
{
    // Предусловие: см. ниже
    while (f_i0 != l_i0 && f_i1 != l_i1) {
        if (r(predecessor(l_i1), predecessor(l_i0)))
            copy_backward_step(l_i0, l_o);
        else
            copy_backward_step(l_i1, l_o);
    }
    return copy_backward(f_i0, l_i0,
                         copy_backward(f_i1, l_i1, l_o));
}
```

Предусловием для combine_copy_backward является

$$(\mathsf{forward\_offset}(\mathrm{f_{i_0}}, \mathrm{l_{i_0}}, \mathrm{f_o}, \mathrm{l_o}, \mathrm{l_{i_1}} - \mathrm{f_{i_1}}) \wedge \mathsf{not\_overlapped}(\mathrm{f_{i_1}}, \mathrm{l_{i_1}}, \mathrm{f_o}, \mathrm{l_o})) \vee$$
$$(\mathsf{forward\_offset}(\mathrm{f_{i_1}}, \mathrm{l_{i_1}}, \mathrm{f_o}, \mathrm{l_o}, \mathrm{l_{i_0}} - \mathrm{f_{i_0}}) \wedge \mathsf{not\_overlapped}(\mathrm{f_{i_0}}, \mathrm{l_{i_0}}, \mathrm{f_o}, \mathrm{l_o}))$$

где $\mathrm{f_o} = \mathrm{l_o} - (\mathrm{l_{i_0}} - \mathrm{f_{i_0}}) + (\mathrm{l_{i_1}} - \mathrm{f_{i_1}})$ — первый итератор выходного интервала.

Свойство forward_offset удовлетворяется читаемым интервалом, записываемым интервалом и смещением $\mathrm{n} \geqslant 0$, если какие-либо связанные псевдонимом итераторы обнаруживаются в индексе из предела входного интервала, который после увеличения n не превышает индекса из предела выходного интервала:

property$(\mathrm{I} : \mathit{Readable}, \mathrm{O} : \mathit{Writable}, \mathrm{N} : \mathit{Integer})$
  requires$(\mathit{Iterator}(\mathrm{I}) \wedge \mathit{Iterator}(\mathrm{O}))$
forward_offset : $\mathrm{I} \times \mathrm{I} \times \mathrm{O} \times \mathrm{O} \times \mathrm{N}$
  $(\mathrm{f_i}, \mathrm{l_i}, \mathrm{f_o}, \mathrm{l_o}, \mathrm{n}) \mapsto$
    $\mathsf{readable\_bounded\_range}(\mathrm{f_i}, \mathrm{l_i}) \wedge$
    $\mathrm{n} \geqslant 0 \wedge$

$$\text{writable\_bounded\_range}(f_o, l_o) \wedge$$
$$(\forall k_i \in [f_i, l_i))(\forall k_o \in [f_o, l_o))$$
$$\text{aliased}(k_o, k_i) \Rightarrow l_i - k_i + n \leqslant l_o - k_o$$

Следует отметить, что имеет место $\text{not\_overlapped\_backward}(f_i, l_i, f_o, l_o) = \text{forward\_offset}(f_i, l_i, f_o, l_o, 0)$.

Упражнение 9.4. Запишите постусловие для combine_copy_backward и докажите, что оно удовлетворяется каждый раз, когда соблюдается предусловие.

Если алгоритмам копирования, в которых сочетается переход в прямом и обратном направлениях, передается нестрогое упорядочение на типе значений, они производят слияние увеличивающихся интервалов:

```
template<typename I0, typename I1, typename O, typename R>
    requires(Readable(I0) && Iterator(I0) &&
        Readable(I1) && Iterator(I1) &&
        Writable(O) && Iterator(O) &&
        Relation(R) &&
        ValueType(I0) == ValueType(O) &&
        ValueType(I1) == ValueType(O) &&
        ValueType(I0) == Domain(R))
O merge_copy(I0 f_i0, I0 l_i0, I1 f_i1, I1 l_i1, O f_o, R r)
{
    // Предусловие: дополнительно к таковому для combine_copy
    //    weak_ordering(r) ∧
    //    increasing_range(f_i0, l_i0, r) ∧ increasing_range(f_i1, l_i1, r)
    relation_source<I1, I0, R> rs(r);
    return combine_copy(f_i0, l_i0, f_i1, l_i1, f_o, rs);
}

template<typename I0, typename I1, typename O, typename R>
    requires(Readable(I0) && BidirectionalIterator(I0) &&
        Readable(I1) && BidirectionalIterator(I1) &&
        Writable(O) && BidirectionalIterator(O) &&
        Relation(R) &&
        ValueType(I0) == ValueType(O) &&
        ValueType(I1) == ValueType(O) &&
        ValueType(I0) == Domain(R))
O merge_copy_backward(I0 f_i0, I0 l_i0, I1 f_i1, I1 l_i1, O l_o,
                      R r)
{
    // Предусловие: дополнительно к таковому для combine_copy_backward
    //    weak_ordering(r) ∧
    //    increasing_range(f_i0, l_i0, r) ∧ increasing_range(f_i1, l_i1, r)
    relation_source<I1, I0, R> rs(r);
    return combine_copy_backward(f_i0, l_i0, f_i1, l_i1, l_o,
                                 rs);
}
```

Упражнение 9.5. Реализуйте combine_copy_n и combine_copy_backward_n с соответствующими возвращаемыми значениями.

Лемма 9.1. Если размеры входных интервалов равны n_0 и n_1, то merge_copy и merge_copy_backward выполняют $n_0 + n_1$ присваиваний и, в наихудшем случае, $n_0 + n_1 - 1$ сравнений.

Упражнение 9.6. Определите количество сравнений в наилучшем случае и в среднем.

Проект 9.1. Современные вычислительные системы включают оптимизированные библиотечные процедуры для выполнения операций копирования в памяти; в качестве примера можно привести `memmove` и `memcpy`, в которых используются методы оптимизации, не рассматриваемые в настоящей книге. Изучите процедуры, предусмотренные в вашей базовой системе, определите методы, которые в них используются (такие как развертывание цикла и программная конвейерная обработка), и спроектируйте абстрактные процедуры, выражающие настолько многие из этих методов, насколько это возможно. Какие требования к типу и предусловия являются необходимыми для каждого метода? Какие языковые расширения позволили бы компилятору достичь полной гибкости при выполнении этой оптимизации?

9.4. Взаимная перестановка интервалов

Вместо копирования одного интервала в другой иногда бывает полезно *менять местами* два интервала одного и того же размера: взаимно обменивать значения объектов в соответствующих позициях. Алгоритмы взаимной перестановки весьма напоминают алгоритмы копирования, за исключением того, что присваивание заменяется процедурой, взаимно обменивающей значения объектов, на которые указывают два изменяемых итератора:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && Mutable(I1) &&
        ValueType(I0) == ValueType(I1))
void exchange_values(I0 x, I1 y)
{
    // Предусловие: deref(x) и deref(y) определены
    ValueType(I0) t = source(x);
            sink(x) = source(y);
            sink(y) = t;
}
```

Упражнение 9.7. Каковым является постусловие exchange_values?

Лемма 9.2. Результаты применения exchange_values(i, j) и exchange_values(j, i) эквивалентны.

Мы хотели бы избежать в реализации exchange_values фактического создания или уничтожения любых объектов и заниматься просто взаимным обменом значениями двух объектов, чтобы стоимость применения этой процедуры не возрастала с увеличением количества ресурсов, принадлежащих объектам. Мы сможем достичь этой цели в главе 12, применяя понятие *основополагающего типа*.

Как и по отношению к копированию, мы создаем алгоритмы взаимной перестановки из машин, которые принимают два итератора по ссылке и отвечают не только за обмен, но и за обновление итераторов. Одна машина взаимно обменивает два объекта, а затем увеличивает оба итератора:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && ForwardIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
void swap_step(I0& f0, I1& f1)
{
    // Предусловие: deref(f0) и deref(f1) определены
    exchange_values(f0, f1);
    f0 = successor(f0);
    f1 = successor(f1);
}
```

Это ведет к созданию первого алгоритма, аналогичного copy:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && ForwardIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
I1 swap_ranges(I0 f0, I0 l0, I1 f1)
{
    // Предусловие: mutable_bounded_range(f0, l0)
    // Предусловие: mutable_counted_range(f1, l0 - f0)
    while (f0 != l0) swap_step(f0, f1);
    return f1;
}
```

Второй алгоритм аналогичен copy_bounded:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && ForwardIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
pair<I0, I1> swap_ranges_bounded(I0 f0, I0 l0, I1 f1, I1 l1)
```

```
{
    // Предусловие: mutable_bounded_range(f0, l0)
    // Предусловие: mutable_bounded_range(f1, l1)
    while (f0 != l0 && f1 != l1) swap_step(f0, f1);
    return pair<I0, I1>(f0, f1);
}
```

Третий алгоритм аналогичен copy_n:

```
template<typename I0, typename I1, typename N>
    requires(Mutable(I0) && ForwardIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1) &&
        Integer(N))
pair<I0, I1> swap_ranges_n(I0 f0, I1 f1, N n)
{
    // Предусловие: mutable_counted_range(f0, n)
    // Предусловие: mutable_counted_range(f1, n)
    while (count_down(n)) swap_step(f0, f1);
    return pair<I0, I1>(f0, f1);
}
```

Если интервалы, передаваемые в алгоритмы взаимной перестановки интервалов, не перекрываются, то очевидно, что результатом их применения становится взаимный обмен значений объектов в соответствующих позициях. В следующей главе мы выведем постусловие для случая перекрытия.

Обратное копирование приводит к получению копии, в которой позиции являются обратными по отношению к исходной последовательности; аналогична и обратная взаимная перестановка. Для этого требуется вторая машина, которая движется в обратном направлении в первом интервале и в прямом направлении во втором интервале:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && BidirectionalIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
void reverse_swap_step(I0& l0, I1& f1)
{
    // Предусловие: deref(predecessor(l0)) и deref(f1) определены
    l0 = predecessor(l0);
    exchange_values(l0, f1);
    f1 = successor(f1);
}
```

Процедура exchange_values действует симметрично, поэтому reverse_swap_ranges может использоваться в любом случае, если по крайней мере один тип

итератора является двунаправленным; какие-либо обратные версии не требуются:

```
template<typename I0, typename I1>
    requires(Mutable(I0) && BidirectionalIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
I1 reverse_swap_ranges(I0 f0, I0 l0, I1 f1)
{
    // Предусловие: mutable_bounded_range(f0, l0)
    // Предусловие: mutable_counted_range(f1, l0 - f0)
    while (f0 != l0) reverse_swap_step(l0, f1);
    return f1;
}

template<typename I0, typename I1>
    requires(Mutable(I0) && BidirectionalIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1))
pair<I0, I1>reverse_swap_ranges_bounded(I0 f0, I0 l0,
                                        I1 f1, I1 l1)
{
    // Предусловие: mutable_bounded_range(f0, l0)
    // Предусловие: mutable_bounded_range(f1, l1)
    while (f0 != l0 && f1 != l1)
        reverse_swap_step(l0, f1);
    return pair<I0, I1>(l0, f1);
}

template<typename I0, typename I1, typename N>
    requires(Mutable(I0) && BidirectionalIterator(I0) &&
        Mutable(I1) && ForwardIterator(I1) &&
        ValueType(I0) == ValueType(I1) &&
        Integer(N))
pair<I0, I1> reverse_swap_ranges_n(I0 l0, I1 f1, N n)
{
    // Предусловие: mutable_counted_range(l0 - n, n)
    // Предусловие: mutable_counted_range(f1, n)
    while (count_down(n)) reverse_swap_step(l0, f1);
    return pair<I0, I1>(l0, f1);
}
```

9.5. Резюме

Расширение типа итератора с помощью процедуры sink приводит к получению таких методов доступа, как записываемость и изменяемость. Аксиома

для sink проста, но из-за проблем, обусловленных связыванием псевдонимами и параллельными обновлениями, которые не рассматриваются в настоящей книге, императивное программирование усложняется. В частности, требует значительной осторожности определение предусловий, которые касаются связывания псевдонимами различных типов итераторов. Алгоритмы копирования являются простыми, мощными и широко используются. Составление этих алгоритмов из простых машин помогает организовать их в семейство путем выявления общих свойств и предоставления дополнительных разновидностей. Использование взаимного обмена значениями вместо присваивания значений приводит к созданию аналогичного, но несколько меньшего семейства полезных алгоритмов взаимной перестановки интервалов.

Глава 10
Переупорядочения

В настоящей главе вводятся концепция перестановки и таксономия для класса алгоритмов, называемых переупорядочениями, которые переставляют элементы интервала в целях удовлетворения заданного постусловия. Мы предоставляем итеративные алгоритмы обращения для двунаправленных итераторов и итераторов с произвольным доступом, а также алгоритм декомпозиции для обращения на прямых итераторах. Мы покажем, как преобразовать алгоритмы декомпозиции, чтобы обеспечить их более быстрое выполнение, если доступна дополнительная память. Описаны три алгоритма вращения, соответствующие различным концепциям итератора, где под вращением подразумевается взаимный обмен двух смежных интервалов не обязательно равного размера. В заключение обсуждается, как оформить алгоритмы для предоставления возможности выбора среди них во время компиляции на основе требований к алгоритмам.

10.1. Перестановки

Преобразование f является *инъективным* преобразованием, если для всех x в его области определения существует y в его области определения, такой, что $y = f(x)$. Преобразование f является *суръективным* преобразованием, если для всех y в его области определения существует x в его области определения, такой, что $y = f(x)$. Преобразование f является *биективным* преобразованием, если для всех x, x' в его области определения имеет место $f(x) = f(x') \Rightarrow x = x'$.

Лемма 10.1. Преобразование в конечной области определения является суръективным преобразованием, если и только если оно является и инъективным, и биективным преобразованием.

Упражнение 10.1. Найдите преобразование натуральных чисел, которое является инъективным и суръективным, но не биективным, а также такое преобразование, которое является инъективным и биективным, но не суръективным.

Неподвижная точка преобразования — это элемент x, такой что $f(x) = x$. *Тождественное преобразование* — это такое преобразование, которое имеет каждый элемент своей области определения как неподвижную точку. Мы обозначаем тождественное преобразование на множестве S как $\mathsf{identity}_S$.

Перестановка — это суръективное преобразование в конечной области определения. Ниже приведен пример перестановки на $[0,6)$.

$$\begin{aligned} p(0) &= 5 \\ p(1) &= 2 \\ p(2) &= 4 \\ p(3) &= 3 \\ p(4) &= 1 \\ p(5) &= 0 \end{aligned}$$

Если p и q — две перестановки на множестве S, то *композиция* $q \circ p$ преобразует $x \in S$ в $q(p(x))$.

Лемма 10.2. Композиция перестановок является перестановкой.

Лемма 10.3. Композиция перестановок ассоциативна.

Лемма 10.4. Для каждой перестановки p на множестве S имеется *обратная перестановка* p^{-1}, такая, что $p^{-1} \circ p = p \circ p^{-1} = \mathsf{identity}_S$.

Перестановки на множестве образуют группу согласно композиции.

Лемма 10.5. Каждая конечная группа представляет собой подгруппу группы перестановок ее элементов, где каждая перестановка в подгруппе формируется путем умножения всех элементов на отдельный элемент.

Например, группа умножения по модулю 5 имеет следующую таблицу умножения:

$\times$	**1**	**2**	**3**	**4**
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

Каждая строка и каждый столбец таблицы умножения — это перестановка. В таблице умножения присутствует не каждая из $4! = 24$ перестановок четырех элементов, поэтому группа умножения по модулю 5 является собственной подгруппой группы перестановок четырех элементов.

Цикл — это циклическая орбита в пределах перестановки. *Тривиальный цикл* — это цикл с размером 1; элементом в тривиальном цикле является неподвижная точка. Перестановка, содержащая единственный нетривиальный цикл, называется *циклической перестановкой*. *Транспозиция* — это циклическая перестановка с размером цикла 2.

Лемма 10.6. Каждый элемент в перестановке принадлежит к уникальному циклу.

Лемма 10.7. Любая перестановка множества с n элементами содержит $k \leqslant n$ циклов.

Лемма 10.8. Непересекающиеся циклические перестановки коммутативны.

Упражнение 10.2. Приведите пример двух ненепересекающихся циклических перестановок, которые не являются коммутативными.

Лемма 10.9. Каждая перестановка может быть представлена как произведение циклических перестановок, соответствующих ее циклам.

Лемма 10.10. Инверсия перестановки является произведением инверсий ее циклов.

Лемма 10.11. Каждая циклическая перестановка — это произведение транспозиций.

Лемма 10.12. Каждая перестановка — произведение транспозиций.

Конечное множество S с размером n — это множество, для которого может быть определена пара функций

$$\mathrm{choose}_S : [0, n) \to S$$
$$\mathrm{index}_S : S \to [0, n)$$

удовлетворяющих условию

$$\mathrm{choose}_S(\mathrm{index}_S(x)) = x$$
$$\mathrm{index}_S(\mathrm{choose}_S(i)) = i$$

Иными словами, S может быть приведено во взаимно-однозначное соответствие с интервалом натуральных чисел.

Если p — перестановка на конечном множестве S с размером n, то имеется соответствующая *индексная перестановка* p' на $[0, n)$, определенная как

$$p'(i) = \mathrm{index}_S(p(\mathrm{choose}_S(i)))$$

Лемма 10.13. $p(x) = \mathrm{choose}_S(p'(\mathrm{index}_S(x)))$

Мы будем часто определять перестановки с помощью соответствующих индексных перестановок.

10.2. Переупорядочения

Переупорядочение — это алгоритм, который копирует объекты из входного интервала в выходной интервал, таким образом, что отображение между индексами входного и выходного интервалов представляет собой перестановку. В настоящей главе рассматриваются переупорядочения *с учетом позиции*, в которых выбор будущей позиции значения зависит только от его исходной позиции, а не от самого значения. В следующей главе рассматриваются переупорядочения *с учетом предиката*, в которых выбор будущей позиции значения зависит только от результата применения предиката к значению, а также переупорядочения *с учетом упорядочения*, в которых выбор будущей позиции зависит только от упорядочения значений.

В главе 8 рассматривались переупорядочения связей, такие как reverse_linked, где для осуществления переупорядочения изменения вносятся в связи. В главе 9 было приведено описание переупорядочений путем копирования, таких как copy и reverse_copy. В настоящей и следующей главах речь идет об *изменяемых* переупорядочениях, в которых входные и выходные интервалы идентичны.

Каждое изменяемое переупорядочение соответствует двум перестановкам: *входящей перестановке*, отображающей итератор `i` в итератор, который указывает на будущую позицию элемента в `i`, и *исходящей перестановке*, отображающей итератор `i` в итератор, который указывает на исходную позицию элемента, перемещенного в `i`.

Лемма 10.14. Входящая и исходящая перестановки для переупорядочения являются инверсиями друг друга.

Если известна входящая перестановка, мы можем перестроить цикл с помощью следующего алгоритма:

```
template<typename I, typename F>
    requires(Mutable(I) && Transformation(F) && I == Domain(F))
void cycle_to(I i, F f)
{
    // Предусловие: орбита i при f является циклической
    // Предусловие: (∀n ∈ ℕ) deref(f^n(i)) определено
    I k = f(i);
    while (k != i) {
        exchange_values(i, k);
        k = f(k);
    }
}
```

После применения cycle_to(i, f) значение source(f(j)) и исходное значение source(j) становятся равными для всех j в орбите i согласно f. В этом вызове выполняются $3(n-1)$ присваиваний для цикла с размером n.

Упражнение 10.3. Реализуйте версию cycle_to, которая выполняет $2n-1$ присваиваний.

Если известна исходящая перестановка, мы можем перестроить цикл с помощью следующего алгоритма:

```
template<typename I, typename F>
    requires(Mutable(I) && Transformation(F) && I == Domain(F))
void cycle_from(I i, F f)
{
    // Предусловие: орбита i при f является циклической
    // Предусловие: (∀n ∈ ℕ) deref(f^n(i)) определено
    ValueType(I) tmp = source(i);
    I j = i;
    I k = f(i);
    while (k != i) {
        sink(j) = source(k);
        j = k;
        k = f(k);
    }
    sink(j) = tmp;
}
```

После применения cycle_from(i, f) значение source(j) и исходное значение source(f(j)) становятся равными для всех j в орбите i согласно f. В этом вызове выполняется $n+1$ присваиваний, тогда как после реализации его с помощью exchange_values выполнялось бы $3(n-1)$ присваиваний. Следует отметить, что мы требуем изменяемости только по отношению к типу I; какие-либо функции обхода не требуются, поскольку обход выполняет преобразование f. В дополнение к исходящей перестановке для реализации изменяемого переупорядочения с использованием cycle_from требуется способ получения представительного элемента из каждого цикла. В некоторых случаях структура цикла и представительные элементы циклов известны.

Упражнение 10.4. Реализуйте алгоритм, который выполняет произвольное переупорядочение интервала индексированных итераторов. Используйте массив из n булевых значений, чтобы отмечать элементы после их размещения, и проверяйте этот массив на наличие неотмеченных значений в целях определения представительного элемента следующего цикла.

Упражнение 10.5. Руководствуясь предположением, что итераторы полностью упорядочены, спроектируйте алгоритм, который использует постоян-

ный объем памяти, для определения того, является ли итератор представительным элементом для цикла; используйте этот алгоритм, чтобы реализовать произвольное переупорядочение.

Лемма 10.15. Если дана исходящая перестановка, появляется возможность выполнить изменяемое переупорядочение с использованием $n + c_N - c_T$ присваиваний, где n — количество элементов; c_N — количество нетривиальных циклов; c_T — количество тривиальных циклов.

10.3. Алгоритмы обращения

Простое, но полезное изменяемое переупорядочение с учетом позиции позволяет полностью найти обращение интервала. Это переупорядочение можно вывести по индукции с помощью *обратной перестановки* на конечном множестве с n элементами, которое определено индексной перестановкой

$$p(i) = (n - 1) - i$$

Лемма 10.16. Количество нетривиальных циклов в обратной перестановке равно $\lfloor n/2 \rfloor$; количество тривиальных циклов составляет $n \bmod 2$.

Лемма 10.17. $\lfloor n/2 \rfloor$ является наибольшим количеством нетривиальных циклов в перестановке.

Это определение обращения непосредственно приводит к следующему алгоритму для индексированных итераторов[1]:

```
template<typename I>
    requires(Mutable(I) && IndexedIterator(I))
void reverse_n_indexed(I f, DistanceType(I) n)
{
    // Предусловие: mutable_counted_range(f, n)
    DistanceType(I) i(0);
    n = predecessor(n);
    while (i < n) {
        // n = (n_original - 1) - i
        exchange_values(f + i, f + n);
        i = successor(i);
        n = predecessor(n);
    }
}
```

[1]Этот алгоритм мог бы возвращать интервал элементов, которые не были перемещены: средний элемент при нечетном размере интервала или пустой интервал между двумя средними элементами, если размер интервала — четный. Нам неизвестен пример, в котором это возвращаемое значение было бы полезным, поэтому возвращаем `void`. Разумеется, в версиях, принимающих счетный интервал прямых итераторов, полезно обеспечить возврат значения предела.

Если этот алгоритм используется с прямыми или двунаправленными итераторами, то выполняет квадратичное количество приращений итератора. Что касается двунаправленных итераторов, то требуются по две проверки на каждую итерацию:

```
template<typename I>
    requires(Mutable(I) && BidirectionalIterator(I))
void reverse_bidirectional(I f, I l)
{
    // Предусловие: mutable_bounded_range(f,l)
    while (true) {
        if (f == l) return;
        l = predecessor(l);
        if (f == l) return;
        exchange_values(f, l);
        f = successor(f);
    }
}
```

Если размер интервала известен, то может использоваться reverse_swap_ranges_n:

```
template<typename I>
    requires(Mutable(I) && BidirectionalIterator(I))
void reverse_n_bidirectional(I f, I l, DistanceType(I) n)
{
    // Предусловие: mutable_bounded_range(f,l) ∧ 0 ⩽ n ⩽ l − f
    reverse_swap_ranges_n(l, f, half_nonnegative(n));
}
```

Порядок первых двух аргументов функции reverse_swap_ranges_n определен в соответствии с тем фактом, что эта функция осуществляет продвижение назад в первом интервале. Передача $n < l - f$ в функцию reverse_n_bidirectional приводит к тому, что значения в середине остаются в своих исходных позициях.

Если в структуре данных предусмотрены прямые итераторы, то иногда они являются связанными итераторами и в этом случае может использоваться reverse_linked. В других случаях может быть доступна дополнительная буферная память, что позволяет применять следующий алгоритм:

```
template<typename I, typename B>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && BidirectionalIterator(B) &&
        ValueType(I) == ValueType(B))
I reverse_n_with_buffer(I f_i, DistanceType(I) n, B f_b)
{
    // Предусловие: mutable_counted_range(f_i,n)
    // Предусловие: mutable_counted_range(f_b,n)
```

```
    return reverse_copy(f_b, copy_n(f_i, n, f_b).m1, f_i);
}
```

reverse_n_with_buffer выполняет $2n$ присваиваний.

Мы будем использовать этот подход с копированием в буфер и обратно для других переупорядочений.

Если буферная память недоступна, но в качестве пространства стека может использоваться память с объемом, определяемым по логарифмическому закону, возможно применение алгоритма декомпозиции: разбейте интервал на две части, выполните обращение каждой части и, наконец, поменяйте местами эти части с помощью swap_ranges_n.

Лемма 10.18. Разбиение, настолько равное, насколько это возможно, позволяет свести работу к минимуму.

Возврат значения предела позволяет оптимизировать обход в сторону средней точки с использованием метода, названного нами *вспомогательным вычислением в ходе рекурсии*:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
I reverse_n_forward(I f, DistanceType(I) n)
{
    // Предусловие: mutable_counted_range(f, n)
    typedef DistanceType(I) N;
    if (n < N(2)) return f + n;
    N h = half_nonnegative(n);
    N n_mod_2 = n - twice(h);
    I m = reverse_n_forward(f, h) + n_mod_2;
    I l = reverse_n_forward(m, h);
    swap_ranges_n(f, m, h);
    return l;
}
```

Правильность reverse_n_forward зависит от следующего.

Лемма 10.19. Обратная перестановка на $[0, n)$ является единственной перестановкой, удовлетворяющей $i < j \Rightarrow p(j) < p(i)$.

Очевидно, что это условие соблюдается для интервалов с размером 1. Для рекурсивных вызовов по индукции можно установить, что условие соблюдается в пределах каждой половины. Условие вновь устанавливается с помощью swap_ranges_n между этими половинами и пропущенным средним элементом, если таковой имеется.

Лемма 10.20. Для интервала с длиной $n = \sum_{i=0}^{\lfloor \log n \rfloor} a_i 2^i$, где a_i — ith цифра в двоичном представлении n, количество присваиваний равно $\frac{3}{2} \sum_{i=0}^{\lfloor \log n \rfloor} a_i i 2^i$.

reverse_n_forward требует логарифмического объема пространства для стека вызовов. В *адаптивном к памяти* алгоритме для максимального повышения производительности используется весь объем дополнительного пространства, какой только он может получить. Даже несколько процентов дополнительного пространства дают значительное повышение производительности. Это приводит к следующему алгоритму, в котором используется разделение и происходит переключение на функцию reverse_n_with_buffer с линейными затратами времени каждый раз, когда данные для подзадачи вписываются в буфер:

```
template<typename I, typename B>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && BidirectionalIterator(B) &&
        ValueType(I) == ValueType(B))
I reverse_n_adaptive(I f_i, DistanceType(I) n_i,
                     B f_b, DistanceType(I) n_b)
{
    // Предусловие: mutable_counted_range(f_i, n_i)
    // Предусловие: mutable_counted_range(f_b, n_b)
    typedef DistanceType(I) N;
    if (n_i < N(2))
        return f_i + n_i;
    if (n_i <= n_b)
        return reverse_n_with_buffer(f_i, n_i, f_b);
    N h_i = half_nonnegative(n_i);
    N n_mod_2 = n_i - twice(h_i);
    I m_i = reverse_n_adaptive(f_i, h_i, f_b, n_b) + n_mod_2;
    I l_i = reverse_n_adaptive(m_i, h_i, f_b, n_b);
    swap_ranges_n(f_i, m_i, h_i);
    return l_i;
}
```

Упражнение 10.6. Выведите формулу для количества присваиваний, выполняемых в reverse_n_adaptive, для заданного интервала и размеров буферов.

10.4. Алгоритмы вращения

Перестановка p элементов в количестве n, определенная с помощью индексной перестановки $p(i) = (i + k) \bmod n$, называется k-*вращением*.

Лемма 10.21. Инверсия k-вращения n элементов представляет собой $(n - k)$-вращение.

Элемент с индексом i находится в цикле

$$\{i, (i + k) \bmod n, (i + 2k) \bmod n, \ldots\} = \{(i + uk) \bmod n\}$$

Длина цикла — наименьшее положительное целое число m, такое, что

$$i = (i + mk) \bmod n$$

Это равенство эквивалентно равенству $mk \bmod n = 0$, которое показывает длину цикла, обеспечивающую независимость от i. Здесь m — наименьшее положительное число, такое, что $mk \bmod n = 0$, $\mathrm{lcm}(k, n) = mk$, где $\mathrm{lcm}(a, b)$ — *наименьшее общее кратное* a и b. Используя стандартное равенство

$$\mathrm{lcm}(a, b)\,\gcd(a, b) = ab$$

получим, что размер цикла равен

$$m = \frac{\mathrm{lcm}(k, n)}{k} = \frac{kn}{\gcd(k, n)k} = \frac{n}{\gcd(k, n)}$$

Количество циклов, таким образом, равно $\gcd(k, n)$.

Рассмотрим два элемента в цикле: $(i + uk) \bmod n$ и $(i + vk) \bmod n$. Расстояние между ними составляет

$$\begin{aligned} |(i + uk) \bmod n - (i + vk) \bmod n| &= (u - v)k \bmod n \\ &= (u - v)k - pn \end{aligned}$$

где $p = \mathsf{quotient}((u - v)k, n)$. И k, и n являются делимыми на $d = \gcd(k, n)$, поэтому делимым является и это расстояние. Это означает, что расстояние между различными элементами в одном и том же цикле равно по меньшей мере d и элементы с индексами в $[0, d)$ принадлежат к непересекающимся циклам.

Переупорядочение с k-вращением интервала $[f, l)$ эквивалентно взаимному обмену относительными позициями значений в подынтервалах $[f, m)$ и $[m, l)$, где $m = f + ((l - f) - k) = l - k$. Таким образом, m представляет собой более полезное входное значение, чем k. Если задействованы прямые или двунаправленные итераторы, то появляется возможность обойтись без выполнения операций с линейными затратами времени для вычисления m из k. Возврат итератора $m' = f + k$, указывающего на новую позицию элемента в f, является полезным для многих других алгоритмов[2].

Лемма 10.22. Вращение интервала $[f, l)$ вокруг итератора m, а затем вращение его вокруг возвращенного значения m' приводит к возврату m и восстановлению интервала в его исходное состояние.

[2] Джозеф Тай (Joseph Tighe) предлагает возвращать пару, m и m', в таком порядке, чтобы эти элементы составляли допустимый интервал; безусловно, это — интересное предложение, позволяющее сохранить всю информацию, но нам еще не известен достаточно привлекательный способ использования подобного интерфейса.

Мы можем использовать cycle_from для реализации переупорядочения с k-вращением интервала индексированных итераторов или итераторов с произвольным доступом. Входящей перестановкой является $p(i) = (i + k) \bmod n$, а исходящей перестановкой — ее инверсия: $p^{-1}(i) = (i + (n - k)) \bmod n$, где $n - k = m - f$. Мы хотим избежать вычисления mod и отмечаем, что

$$p^{-1}(i) = \begin{cases} i + (n - k) & \text{if } i < k \\ i - k & \text{if } i \geqslant k \end{cases}$$

Это приводит к получению следующего функционального объекта для итераторов с произвольным доступом:

```
template<typename I>
    requires(RandomAccessIterator(I))
struct k_rotate_from_permutation_random_access
{
    DistanceType(I) k;
    DistanceType(I) n_minus_k;
    I m_prime;
    k_rotate_from_permutation_random_access(I f, I m, I l) :
        k(l - m), n_minus_k(m - f), m_prime(f + (l - m))
    {
        // Предусловие: bounded_range(f, l) ∧ m ∈ [f, l)
    }
    I operator()(I x)
    {
        // Предусловие: x ∈ [f, l)
        if (x < m_prime) return x + n_minus_k;
        else             return x - k;
    }
};
```

Что касается индексированных итераторов, то отсутствие естественного упорядочения и необходимость вычитания расстояния из итератора приводят к тому, что приходится выполнять одну-две дополнительные операции сложения:

```
template<typename I>
    requires(IndexedIterator(I))
struct k_rotate_from_permutation_indexed
{
    DistanceType(I) k;
    DistanceType(I) n_minus_k;
    I f;
    k_rotate_from_permutation_indexed(I f, I m, I l) :
        k(l - m), n_minus_k(m - f), f(f)
```

```
    {
        // Предусловие: bounded_range(f, l) ∧ m ∈ [f, l)
    }
    I operator()(I x)
    {
        // Предусловие: x ∈ [f, l)
        DistanceType(I) i = x - f;
        if (i < k) return x + n_minus_k;
        else       return f + (i - k);
    }
};
```

Следующая процедура выполняет вращение каждого цикла:

```
template<typename I, typename F>
    requires(Mutable(I) && IndexedIterator(I) &&
        Transformation(F) && I == Domain(F))
I rotate_cycles(I f, I m, I l, F from)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ m ∈ [f, l]
    // Предусловие: from является исходящей перестановкой на [f, l)
    typedef DistanceType(I) N;
    N d = gcd<N, N>(m - f, l - m);
    while (count_down(d)) cycle_from(f + d, from);
    return f + (l - m);
}
```

Этот алгоритм соответствует алгоритму, впервые опубликованному в [Fletcher and Silver 1966], если не считать того, что его авторы использовали cycle_to, а мы используем cycle_from. Эти процедуры выбирают подходящий функциональный объект:

```
template<typename I>
    requires(Mutable(I) && IndexedIterator(I))
I rotate_indexed_nontrivial(I f, I m, I l)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    k_rotate_from_permutation_indexed<I> p(f, m, l);
    return rotate_cycles(f, m, l, p);
}

template<typename I>
    requires(Mutable(I) && RandomAccessIterator(I))
I rotate_random_access_nontrivial(I f, I m, I l)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    k_rotate_from_permutation_random_access<I> p(f, m, l);
    return rotate_cycles(f, m, l, p);
}
```

Количество присваиваний составляет $n + c_N - c_T = n + \gcd(n, k)$. Напомним, что n — количество элементов, c_N — количество нетривиальных циклов, c_T — количество тривиальных циклов. Ожидаемым значением $\gcd(n, k)$ для $1 \leqslant n, k \leqslant m$ является $\frac{6}{\pi^2} \ln m + C + O(\frac{\ln m}{\sqrt{m}})$ (см. [Diaconis and Erdös 2004]).

Следующее свойство позволяет получить алгоритм вращения для двунаправленных итераторов.

Лемма 10.23. k-вращение на $[0, n)$ является единственной перестановкой p, которая инвертирует относительное упорядочение между подынтервалами $[0, n-k)$ и $[n-k, n)$, но сохраняет относительное упорядочение в каждом подынтервале:

1. $i < n-k \wedge n-k \leqslant j < n \Rightarrow p(j) < p(i)$
2. $i < j < n-k \vee n-k \leqslant i < j \Rightarrow p(i) < p(j)$

Обратное переупорядочение удовлетворяет условию 1, но не 2. Применение обращения к подынтервалам $[0, n-k)$ и $[n-k, n)$ с последующим применением обращения ко всему интервалу удовлетворяет обоим условиям:

```
reverse_bidirectional(f, m);
reverse_bidirectional(m, l);
reverse_bidirectional(f, l);
```

Поиск возвращаемого значения m' осуществляется с использованием reverse_swap_ranges_bounded[3]:

```
template<typename I>
    requires(Mutable(I) && BidirectionalIterator(I))
I rotate_bidirectional_nontrivial(I f, I m, I l)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    reverse_bidirectional(f, m);
    reverse_bidirectional(m, l);
    pair<I, I> p = reverse_swap_ranges_bounded(m, l, f, m);
    reverse_bidirectional(p.m1, p.m0);
    if (m == p.m0) return p.m1;
    else           return p.m0;
}
```

Лемма 10.24. Количество присваиваний равно $3(\lfloor n/2 \rfloor + \lfloor k/2 \rfloor + \lfloor (n-k)/2 \rfloor)$, что составляет $3n$, если и n, и k являются четными, а $3(n-2)$ — нечетным.

Если задан интервал $[f, l)$ и итератор m в этом интервале, то вызов

$$p \leftarrow \text{swap\_ranges\_bounded}(f, m, m, l)$$

[3]Использование reverse_swap_ranges_bounded для определения m' было предложено нам Уилсоном Хо (Wilson Ho) и Реймондом Ло (Raymond Lo).

задает p равным паре итераторов, таких, что

$$\mathrm{p.m0} = \mathrm{m} \vee \mathrm{p.m1} = \mathrm{l}$$

Если $\mathrm{p.m0} = \mathrm{m} \wedge \mathrm{p.m1} = \mathrm{l}$, то наша цель достигнута. В противном случае $[\mathrm{f}, \mathrm{p.m0})$ находится в окончательной позиции и, в зависимости от того, имеет ли место $\mathrm{p.m0} = \mathrm{m}$ или $\mathrm{p.m1} = \mathrm{l}$, необходимо выполнить вращение $[\mathrm{p.m0}, \mathrm{l})$ вокруг $\mathrm{p.m1}$ или m соответственно. Это непосредственно приводит к следующему алгоритму, впервые опубликованному в [Gries and Mills 1981]:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
void rotate_forward_annotated(I f, I m, I l)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
                                        DistanceType(I) a = m - f;
                                        DistanceType(I) b = l - m;
    while (true) {
        pair<I, I> p = swap_ranges_bounded(f, m, m, l);
        if (p.m0 == m && p.m1 == l) { assert(a == b);
            return;
        }
        f = p.m0;
        if (f == m) {                   assert(b > a);
            m = p.m1;                   b = b - a;
        } else {                        assert(a > b);
                                        a = a - b;
        }
    }
}
```

Лемма 10.25. После первого применения предложения `else` имеет место $\mathrm{f} = \mathrm{m}'$, стандартное возвращаемое значение для вращения.

Дополнительные переменные a и b остаются равными размерам двух подынтервалов, с которыми должен быть произведен взаимный обмен. Вместе с тем они находят путем вычитания $\gcd$ по отношению к начальным размерам. Каждый вызов exchange_values, выполненный в swap_ranges_bounded, приводит к размещению одного значения в его окончательной позиции, кроме последнего вызова swap_ranges_bounded, при котором каждый вызов exchange_values размещает два значения в их окончательных позициях. В последнем вызове swap_ranges_bounded выполняются $\gcd(\mathrm{n}, \mathrm{k})$ вызовов exchange_values, поэтому общее количество вызовов в exchange_values равно $\mathrm{n} - \gcd(\mathrm{n}, \mathrm{k})$.

Предыдущая лемма предоставляет еще одну возможность реализации полной процедуры rotate_forward: создание второй копии кода, которая сохраняет копию функции f в предложении `else`, а затем вызывает rotate_forward_

annotated для завершения вращения. Это может быть преобразовано в следующие две процедуры:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
void rotate_forward_step(I& f, I& m, I l)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    I c = m;
    do {
        swap_step(f, c);
        if (f == m) m = c;
    } while (c != l);
}

template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
I rotate_forward_nontrivial(I f, I m, I l)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    rotate_forward_step(f, m, l);
    I m_prime = f;
    while (m != l) rotate_forward_step(f, m, l);
    return m_prime;
}
```

Упражнение 10.7. Проверьте, действительно ли rotate_forward_nontrivial вращает $[\mathrm{f}, \mathrm{l})$ вокруг m и возвращает m'.

Иногда бывает полезно выполнить *частичное вращение* интервала, перемещая правильно выбранные объекты в $[\mathrm{f}, \mathrm{m}')$, но оставляя объекты в $[\mathrm{m}', \mathrm{l})$ в некотором переупорядочении объектов, первоначально находившихся в $[\mathrm{f}, \mathrm{m})$. Например, это может использоваться для перемещения нежелательных объектов в конец последовательности при подготовке к их стиранию. Это можно осуществить с помощью следующего алгоритма:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
I rotate_partial_nontrivial(I f, I m, I l)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    return swap_ranges(m, l, f);
}
```

Лемма 10.26. Постусловием для rotate_partial_nontrivial является то, что эта функция выполняет частичное вращение, такое, что объекты в позициях $[\mathrm{m}', \mathrm{l})$ подвергнуты k-вращению, где $\mathrm{k} = -(\mathrm{l} - \mathrm{f}) \bmod (\mathrm{m} - \mathrm{f})$.

Иногда может быть полезна обратная версия rotate_partial_nontrivial, в которой используется обратная версия swap_ranges.

Если доступна дополнительная буферная память, может использоваться следующий алгоритм:

```
template<typename I, typename B>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B))
I rotate_with_buffer_nontrivial(I f, I m, I l, B f_b)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    // Предусловие: mutable_counted_range(f_b, l − f)
    B l_b = copy(f, m, f_b);
    I m_prime = copy(m, l, f);
    copy(f_b, l_b, m_prime);
    return m_prime;
}
```

rotate_with_buffer_nontrivial выполняет $(l-f)+(m-f)$ присваиваний, тогда как следующий алгоритм выполняет $(l-f)+(l-m)$ присваиваний. Если производится вращение интервала двунаправленных итераторов, то может быть выбран алгоритм, который сводит к минимуму количество присваиваний, хотя для вычисления разностей во время выполнения требуется линейное количество операций successor:

```
template<typename I, typename B>
    requires(Mutable(I) && BidirectionalIterator(I) &&
        Mutable(B) && ForwardIterator(B))
I rotate_with_buffer_backward_nontrivial(I f, I m, I l, B f_b) {
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    // Предусловие: mutable_counted_range(f_b, l − f)
    B l_b = copy(m, l, f_b);
    copy_backward(f, m, l);
    return copy(f_b, l_b, f);
}
```

10.5. Выбор алгоритма

В разделе 10 мы представили обратные алгоритмы с различными требованиями к итераторам и сигнатурами процедур, включая версии, принимающие счетный и ограниченный интервалы. Имеет смысл определить такие разновидности алгоритмов, которые предоставляют доступ к наиболее удобным сигнатурам, доступным для дополнительных типов итераторов. Например, применение дополнительной операции вычисления разности итераторов с постоянными

затратами времени приводит к созданию алгоритма для обращения ограниченного интервала индексированных итераторов:

```
template<typename I>
    requires(Mutable(I) && IndexedIterator(I))
void reverse_indexed(I f, I l)
{
    // Предусловие: mutable_bounded_range(f, l)
    reverse_n_indexed(f, l - f);
}
```

Если должно быть выполнено обращение интервала прямых итераторов, обычно имеется достаточный объем дополнительной памяти для обеспечения эффективного выполнения reverse_n_adaptive. Если размер интервала, подлежащего обращению, остается не слишком большим, то дополнительная память может быть получена обычным образом (например, с помощью функции malloc). Но если этот размер очень велик, то может не оказаться достаточного объема доступной физической памяти для распределения буфера такого размера. Такие алгоритмы, как reverse_n_adaptive, работают эффективно, даже если размер буфера невелик по сравнению с интервалом, подлежащим изменению, поэтому имеет смысл предусмотреть в системе способ распределения *временного буфера*. При этом распределении может быть зарезервирован меньший объем памяти по сравнению с запрашиваемым; в системе с виртуальной памятью такая распределенная память имеет назначенную ей физическую память. Временный буфер предназначен для краткосрочного использования, а его возврат после завершения работы алгоритма гарантирован.

Например, в следующем алгоритме используется тип temporary_buffer:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
void reverse_n_with_temporary_buffer(I f, DistanceType(I) n)
{
    // Предусловие: mutable_counted_range(f, n)
    temporary_buffer<ValueType(I)> b(n);
    reverse_n_adaptive(f, n, begin(b), size(b));
}
```

Конструктор $b(n)$ распределяет память для хранения некоторого количества $m \leqslant n$ смежно расположенных объектов типа ValueType(I); size(b) возвращает количество m, а begin(b) возвращает итератор, указывающий на начало этого интервала. Деструктор для b освобождает память.

Для одной и той же задачи часто предусмотрены разные алгоритмы для различных требований к типам. Например, что касается вращения, то имеются три полезных алгоритма для индексированных (и с произвольным доступом), двунаправленных и прямых итераторов. Есть возможность автоматически вы-

бирать алгоритмы из семейства алгоритмов на основе требований, которым удовлетворяют типы. Мы достигаем этой цели с использованием механизма, известного как *планирование концепции*. Для этого прежде всего определяется процедура планирования верхнего уровня, которая в данном случае осуществляет также тривиальные вращения:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
I rotate(I f, I m, I l)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ m ∈ [f, l]
    if (m == f) return l;
    if (m == l) return f;
    return rotate_nontrivial(f, m, l, IteratorConcept(I)());
}
```

Функция типа IteratorConcept возвращает *тип дескриптора концепции*, который представляет в коде самую сильную концепцию, смоделированную ее аргументом. Затем мы реализуем процедуру для каждого типа дескриптора концепции:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
I rotate_nontrivial(I f, I m, I l, forward_iterator_tag) {
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    return rotate_forward_nontrivial(f, m, l);
}

template<typename I>
    requires(Mutable(I) && BidirectionalIterator(I))
I rotate_nontrivial(I f, I m, I l, bidirectional_iterator_tag)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    return rotate_bidirectional_nontrivial(f, m, l);
}

template<typename I>
    requires(Mutable(I) && IndexedIterator(I))
I rotate_nontrivial(I f, I m, I l, indexed_iterator_tag)
{
    // Предусловие: mutable_bounded_range(f, l) ∧ f ≺ m ≺ l
    return rotate_indexed_nontrivial(f, m, l);
}

template<typename I>
    requires(Mutable(I) && RandomAccessIterator(I))
I rotate_nontrivial(I f, I m, I l, random_access_iterator_tag)
```

```
{
    // Предусловие: mutable_bounded_range(f,l) ∧ f ≺ m ≺ l
    return rotate_random_access_nontrivial(f, m, l);
}
```

При планировании концепции не учитываются иные факторы, кроме требований к типам. Например, как представлено в итоговом виде в табл. 10.1, мы можем вращать интервал итераторов с произвольным доступом при использовании трех алгоритмов, в каждом из которых выполняется различное количество присваиваний. Если интервал вписывается в кэш-память, то $n+\gcd(n,k)$ присваиваний, выполняемых алгоритмом с произвольным доступом, обеспечивают наилучшую производительность. Но если интервал не вписывается в кэш, то быстрее становится двунаправленный алгоритм с $3n$ присваиваниями или прямой алгоритм с $3(n-\gcd(n,k))$ присваиваниями. В этом случае на то, будет ли работать быстрее двунаправленный или прямой алгоритмы, влияют дополнительные факторы, включая более регулярную структуру цикла двунаправленного алгоритма, из-за чего могут потребоваться дополнительные выполняемые в нем присваивания, и нюансы архитектуры процессора, такие как конфигурация его кэша и логика предварительной выборки. Следует также отметить, что эти алгоритмы выполняют операции с итераторами в дополнение к присваиваниям типа значений; по мере того, как размер типа значений становится меньше, относительная стоимость этих прочих операций возрастает.

Таблица 10.1. Количество присваиваний, выполняемых алгоритмами вращения

Алгоритм	**Присваивания**
indexed, random_access	$n+\gcd((n,k)$
bidirectional	$3n$ or $3(n-2)$
forward	$3(n-\gcd(n,k))$
with_buffer	$n+(n\text{-}k)$
with_buffer_backward	$n+k$
partial	$3k$

Примечание. Здесь $n=l-f$ и $k=l-m$

Проект 10.1. Спроектируйте тест для сравнения производительности всех алгоритмов при различных размерах массивов, размерах элементов и количествах вращений. На основе результатов этого теста спроектируйте составной алгоритм, который соответствующим образом использует один из алгоритмов вращения в зависимости от концепции итератора, размера интервала, количества вращений, размера элемента, размера кэша, доступности временного буфера и других относящихся к делу соображений.

Проект 10.2. Мы представили две разновидности алгоритмов переупорядочения с учетом позиции: обращение и вращение. Однако в литературе есть и другие примеры таких алгоритмов. Разработайте таксономию переупорядочений с учетом позиции, сведите в каталог существующие алгоритмы, определите недостающие алгоритмы и создайте библиотеку.

10.6. Резюме

Структура перестановок позволяет проектировать и анализировать алгоритмы переупорядочения. Даже простые задачи, такие как обращения и вращения, ведут к созданию широкого набора полезных алгоритмов. Выбор наиболее подходящего из них зависит от требований к итераторам и особенностей вычислительной системы. Алгоритмы, адаптивные к памяти, предоставляют практическую альтернативу теоретическому понятию замещающих алгоритмов.

Глава 11

Разбиение и слияние

В настоящей главе формируются переупорядочения на основе предикатов и упорядочений из компонентов, описанных в предыдущих главах. После представления алгоритмов разбиения для прямых и двунаправленных итераторов мы реализуем алгоритм стабильного разбиения. Затем введем механизм двоичного счетчика для преобразования восходящих алгоритмов декомпозиции, таких как алгоритм стабильного разбиения, в итерационную форму. Мы введем алгоритм стабильного слияния, адаптивный к памяти, и воспользуемся им для создания эффективной стабильной сортировки, адаптивной к памяти, применимой и для прямых итераторов: наименее строгая концепция, которая допускает переупорядочения.

11.1. Разбиение

В главе 6 было введено понятие интервала, разделенного по предикату, наряду с фундаментальным алгоритмом partition_point на таком интервале. Теперь мы рассмотрим алгоритмы для преобразования произвольного интервала в интервал, подвергнутый разбиению.

Упражнение 11.1. Реализуйте алгоритм partitioned_at_point, который проверяет, разделен ли данный ограниченный интервал по указанному итератору.

Упражнение 11.2. Реализуйте алгоритм potential_partition_point, возвращающий итератор, в котором появилась бы точка разбиения после проведения разбиения.

Лемма 11.1. Если $\mathrm{m} = \mathsf{potential\_partition\_point}(\mathrm{f}, \mathrm{l}, \mathrm{p})$, то

$$\mathsf{count\_if}(\mathrm{f}, \mathrm{m}, \mathrm{p}) = \mathsf{count\_if\_not}(\mathrm{m}, \mathrm{l}, \mathrm{p})$$

Иными словами, количество стоящих не на своих местах элементов с обеих сторон от m является одинаковым.

Эта лемма позволяет определить минимальное количество присваиваний для разбиения интервала, $2n + 1$, где n — количество стоящих не на своих местах элементов с обеих сторон от m: $2n$ присваиваний для элементов, стоящих не на своих местах, и одно присваивание временной переменной.

Лемма 11.2. Существует $u!v!$ перестановок, которые подвергают разбиению интервал с u значениями false и v значениями true.

Разбиение с переупорядочением является *стабильным*, если сохраняется относительный порядок элементов, не удовлетворяющих предикату, как и относительный порядок элементов, удовлетворяющих предикату.

Лемма 11.3. Результат стабильного разбиения является уникальным.

Разбиение с переупорядочением является *полустабильным*, если сохраняется относительный порядок элементов, не удовлетворяющих предикату. Следующий алгоритм выполняет полустабильное разбиение[1]:

```
template<typename I, typename P>
    requires(Mutable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_semistable(I f, I l, P p)
{
    // Предусловие: mutable_bounded_range(f, l)
    I i = find_if(f, l, p);
    if (i == l) return i;
    I j = successor(i);
    while (true) {
        j = find_if_not(j, l, p);
        if (j == l) return i;
        swap_step(i, j);
    }
}
```

Правильность partition_semistable зависит от следующих трех лемм.

Лемма 11.4. Перед проверкой условия выхода имеет место $\mathsf{none}(f, i, p) \wedge \mathsf{all}(i, j, p)$.

Лемма 11.5. После проверки условия выхода имеет место $p(\mathsf{source}(i)) \wedge \neg p(\mathsf{source}(j))$.

Лемма 11.6. После вызова swap_step имеет место $\mathsf{none}(f, i, p) \wedge \mathsf{all}(i, j, p)$.

Полустабильность следует из того факта, что вызов swap_step перемещает элемент, не удовлетворяющий предикату, в позицию перед интервалом элементов, удовлетворяющих предикату, поэтому порядок элементов, не удовлетворяющих предикату, не изменяется.

[1] В [Bentley 1984, стр. 287–291] этот алгоритм приписывается Нико Ломуто (Nico Lomuto).

partition_semistable использует только один временный объект, в swap_step.

Допустим, что $n = l - f$ — количество элементов в интервале, а w — количество элементов, не удовлетворяющих предикату, которые следуют за первым элементом, удовлетворяющим предикату. Тогда предикат применяется n раз, exchange_values выполняется w раз, а количество приращений итератора равняется $n + w$.

Упражнение 11.3. Перепишите partition_semistable, подставив код вместо вызова find_if_not и устранив лишнюю проверку значения l.

Упражнение 11.4. Приведите постусловие алгоритма, которое следует из замены вызова swap_step(i, j) вызовом copy_step(j, i) в функции partition_semistable, предложите подходящее имя и сравните назначение полученной процедуры с назначением процедуры partition_semistable.

Допустим, что n — количество элементов в интервале, подлежащем разбиению.

Лемма 11.7. Разбиение с переупорядочением, которое возвращает точку разбиения, требует n применений предиката.

Лемма 11.8. Разбиение с переупорядочением непустого интервала, которое не возвращает точку разбиения, требует $n - 1$ применений предиката[2].

Упражнение 11.5. Реализуйте разбиение с переупорядочением для непустых интервалов, в котором осуществляется $n - 1$ применений предиката.

Рассмотрим интервал с одним элементом, удовлетворяющим предикату, за которым следуют n элементов, не удовлетворяющих предикату. В partition_semistable выполняется n вызовов exchange_values, тогда как достаточно одного. Если мы объединим прямой поиск элемента, удовлетворяющего предикату, с обратным поиском элемента, не удовлетворяющего предикату, то избежим ненужных обменов местами. Для этого алгоритма требуются двунаправленные итераторы:

```
template<typename I, typename P>
    requires(Mutable(I) && BidirectionalIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_bidirectional(I f, I l, P p)
{
    // Предусловие: mutable_bounded_range(f, l)
    while (true) {
        f = find_if(f, l, p);
        l = find_backward_if_not(f, l, p);
        if (f == l) return f;
```

[2]Эта лемма и следующее упражнение были предложены нам Джоном Брандтом (Jon Brandt).

```
            reverse_swap_step(l, f);
        }
    }
```

Как и в случае partition_semistable, в partition_bidirectional используется только один временный объект.

Лемма 11.9. Количество раз выполнения exchange_values, ν равняется количеству стоящих не на своих местах элементов, не удовлетворяющих предикату. Поэтому общее количество присваиваний равно 3ν.

Упражнение 11.6. Реализуйте разбиение с переупорядочением для прямых итераторов, в котором exchange_values вызывается такое же количество раз, как и partition_bidirectional, благодаря тому, что вначале вычисляется потенциальная точка разбиения.

Возможно осуществить разбиение с другим переупорядочением, которое имеет лишь единственный цикл, что приводит к $2\nu + 1$ присваиваниям. Идея состоит в том, что первый стоящий не на своем месте элемент сохраняется и тем самым создается "дырка", а затем повторно осуществляется поиск стоящего не на своем месте элемента с противоположной стороны от потенциальной точки разбиения и его перемещение в дырку, создание новой дырки и, наконец, перемещение сохраненного элемента в последнюю дырку.

Упражнение 11.7. Используя этот метод, реализуйте partition_single_cycle.

Упражнение 11.8. Реализуйте разбиение с переупорядочением для двунаправленных итераторов, в котором осуществляется поиск подходящих защитных элементов, а затем используются find_if_unguarded и незащищенная версия find_backward_if_not.

Упражнение 11.9. Повторите предыдущее упражнение, применив метод с единственным циклом.

Идеи двунаправленного алгоритма разбиения, а также вариантов с единственным циклом и защищенным элементом принадлежат Ч. Э. Р. Хоару[3].

Если стабильность необходима для обеих сторон разбиения, а объем доступной памяти для буфера с таким же размером, как и интервал, является достаточным, может использоваться следующий алгоритм:

[3]См. в [Hoare 1962] описание алгоритма быстрой сортировки. В соответствии с требованиями быстрой сортировки при разбиении по Хоару происходит обмен местами элементов, которые больше или равны выбранному элементу, с элементами, меньшими или равными выбранному элементу. Интервал равных элементов разделяется посередине. Следует отметить, что эти два отношения, $\leqslant$ и $\geqslant$, не являются дополнениями друг друга.

```
template<typename I, typename B, typename P>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B) &&
        ValueType(I) == ValueType(B) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_stable_with_buffer(I f, I l, B f_b, P p)
{
    // Предусловие: mutable_bounded_range(f, l)
    // Предусловие: mutable_counted_range(f_b, l - f)
    pair<I, B> x = partition_copy(f, l, f, f_b, p);
    copy(f_b, x.m1, x.m0);
    return x.m0;
}
```

Если же памяти для полноразмерного буфера недостаточно, то возможно реализовать стабильное разбиение с использованием алгоритма декомпозиции. Если интервал является одноэлементным, то он уже разделен и его точку разбиения можно определить с помощью одного применения предиката:

```
template<typename I, typename P>
    requires(Mutable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
pair<I, I> partition_stable_singleton(I f, P p)
{
    // Предусловие: readable_bounded_range(f, successor(f))
    I l = successor(f);
    if (!p(source(f))) f = l;
    return pair<I, I>(f, l);
}
```

Возвращенным значением является точка разбиения и предел интервала: иными словами, интервал значений, удовлетворяющих предикату.

Два смежных интервала, подвергнутых разбиению, можно объединить в единственный интервал, подвергнутый разбиению, вращая интервал, ограниченный первой и второй точками разбиения, вокруг середины:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
pair<I, I> combine_ranges(const pair<I, I>& x,
                          const pair<I, I>& y)
{
    // Предусловие: mutable_bounded_range(x.m0, y.m0)
    // Предусловие: x.m1 ∈ [x.m0, y.m0]
    return pair<I, I>(rotate(x.m0, x.m1, y.m0), y.m1);
}
```

Лемма 11.10. Функция combine_ranges является ассоциативной, будучи применяемой к трем неперекрывающимся интервалам.

Лемма 11.11. Если для некоторого предиката p имеет место

$$(\forall \mathrm{i} \in [\mathrm{x}.\mathrm{m}0, \mathrm{x}.\mathrm{m}1))\, \mathrm{p}(\mathrm{i}) \wedge$$
$$(\forall \mathrm{i} \in [\mathrm{x}.\mathrm{m}1, \mathrm{y}.\mathrm{m}0))\, \neg\mathrm{p}(\mathrm{i}) \wedge$$
$$(\forall \mathrm{i} \in [\mathrm{y}.\mathrm{m}0, \mathrm{y}.\mathrm{m}1))\, \mathrm{p}(\mathrm{i})$$

то после применения

$$\mathrm{z} \leftarrow \mathsf{combine\_ranges}(\mathrm{x}, \mathrm{y})$$

соблюдается следующее условие:

$$(\forall \mathrm{i} \in [\mathrm{x}.\mathrm{m}0, \mathrm{z}.\mathrm{m}0))\, \neg\mathrm{p}(\mathrm{i})$$
$$(\forall \mathrm{i} \in [\mathrm{z}.\mathrm{m}0, \mathrm{z}.\mathrm{m}1))\, \mathrm{p}(\mathrm{i})$$

Входы относятся к интервалам значений, удовлетворяющих предикату, и то же касается выходов; поэтому неодноэлементный интервал подвергается стабильному разбиению путем разделения его посередине, рекурсивного разбиения обеих половин, а затем объединения частей, подвергнутых разбиению:

```
template<typename I, typename P>
    requires(Mutable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
pair<I, I> partition_stable_n_nonempty(I f, DistanceType(I) n,
                                                          P p)
{
    // Предусловие: mutable_counted_range(f, n) ∧ n > 0
    if (one(n)) return partition_stable_singleton(f, p);
    DistanceType(I) h = half_nonnegative(n);
    pair<I, I> x = partition_stable_n_nonempty(f, h, p);
    pair<I, I> y = partition_stable_n_nonempty(x.m1, n - h, p);
    return combine_ranges(x, y);
}
```

Получение пустых интервалов никогда не становится результатом дополнительного разбиения интервала с размером больше 1, поэтому мы рассматриваем этот случай только на верхнем уровне:

```
template<typename I, typename P>
    requires(Mutable(I) && ForwardIterator(I) &&
        UnaryPredicate(P) && ValueType(I) == Domain(P))
pair<I, I> partition_stable_n(I f, DistanceType(I) n, P p)
{
    // Предусловие: mutable_counted_range(f, n)
    if (zero(n)) return pair<I, I>(f, f);
    return partition_stable_n_nonempty(f, n, p);
}
```

На нижнем уровне рекурсии осуществляется ровно n применений предиката. Глубина рекурсии для partition_stable_n_nonempty составляет $\lceil \log_2 n \rceil$. На каждом уровне рекурсии мы вращаем в среднем $n/2$ элементов, что требует от $n/2$ до $3n/2$ присваиваний, в зависимости от категории итератора. Общее количество присваиваний равно $n \log_2 n/2$ для итераторов с произвольным доступом и $3n \log_2 n/2$ для прямых и двунаправленных итераторов.

Упражнение 11.10. Воспользуйтесь методами из предыдущей главы для получения версии partition_stable_n, адаптивной к памяти.

11.2. Сбалансированное приведение

Разумеется, производительность partition_stable_n зависит от дополнительного разбиения интервала в середине, но ее правильность — нет. Операция combine_ranges является частично ассоциативной, поэтому дополнительное разбиение может быть выполнено в любой точке. Мы можем воспользоваться этим фактом для создания итеративного алгоритма с аналогичной производительностью; такой алгоритм полезен, например, когда размер интервала неизвестен заранее или должны быть устранены издержки вызова процедуры. Основная идея состоит в использовании приведения, применении partition_stable_singleton к каждому единичному интервалу и объединении результатов с помощью combine_ranges:

```
reduce_nonempty(
    f, l,
    combine_ranges<I>,
    partition_trivial<I, P>(p));
```

где partition_trivial — функциональный объект, который связывает параметр предиката с partition_stable_singleton:

```
template<typename I, typename P>
    requires(ForwardIterator(I) &&
    UnaryPredicate(P) && ValueType(I) == Domain(P))
struct partition_trivial
{
    P p;
    partition_trivial(const P & p) : p(p) { }
    pair<I, I> operator()(I i)
    {
        return partition_stable_singleton<I, P>(i, p);
    }
};
```

В связи с тем, что используется reduce_nonempty, сложность становится квадратичной. Мы должны воспользоваться частичной ассоциативностью, чтобы создать сбалансированное дерево приведения. Мы используем метод двоичного счетчика для создания дерева приведения снизу вверх[4]. Аппаратный двоичный счетчик увеличивает n-битовое двоичное целое число на 1. Значение 1 в позиции i имеет *вес*, равный 2^i; перенос из этой позиции имеет вес 2^{i+1} и распространяется на следующую по высоте позицию. В нашем счетчике используется бит в позиции i для представления либо пустого интервала, либо результата сокращения 2^i элементов из исходного интервала. Если перенос распространяется на следующую по высоте позицию, то либо сохраняется, либо объединяется с другим значением, имеющим тот же вес. Возврат переноса из самой высокой позиции осуществляется в следующей процедуре, которая принимает нейтральный элемент в качестве явно заданного параметра, как и reduce_nonzeroes:

```
template<typename I, typename Op>
    requires(Mutable(I) && ForwardIterator(I) &&
        BinaryOperation(Op) && ValueType(I) == Domain(Op))
Domain(Op) add_to_counter(I f, I l, Op op, Domain(Op) x,
                          const Domain(Op)& z)
{
    if (x == z) return z;
    while (f != l) {
        if (source(f) == z) {
            sink(f) = x;
            return z;
        }
        x = op(source(f), x);
        sink(f) = z;
        f = successor(f);
    }
    return x;
}
```

Память для счетчика предоставляется с применением следующего типа, который обрабатывает переполнения, поступающие из add_to_counter, с помощью следующего расширенного варианта счетчика:

```
template<typename Op>
    requires(BinaryOperation(Op))
struct counter_machine
{
    typedef Domain(Op) T;
```

[4]Этот метод приписывается Джону Маккарти (John McCarthy) в [Knuth 1998, раздел 5.2.4 (Сортировка слиянием), упр. 17, стр. 167].

```
    Op op;
    T z;
    T f[64];
    pointer(T) l;
    counter_machine(Op op, const Domain(Op)& z) :
        op(op), z(z), l(f) { }
    void operator()(const T& x)
    {
        // Предусловие: вызов не должен осуществляться больше 2^64 - 1 раз
         T tmp = add_to_counter(f, l, op, x, z);
         if (tmp != z) {
             sink(l) = tmp;
             l = successor(l);
         }
    }
};
```

При этом используется встроенный массив C++; возможны и альтернативные реализации[5].

После вызова add_to_counter для каждого элемента интервала непустые позиции в счетчике объединяются с крайним левым приведением для получения окончательного результата:

```
template<typename I, typename Op, typename F>
    requires(Iterator(I) && BinaryOperation(Op) &&
        UnaryFunction(F) && I == Domain(F) &&
        Codomain(F) == Domain(Op))
Domain(Op) reduce_balanced(I f, I l, Op op, F fun,
                           const Domain(Op)& z)
{
    // Предусловие: bounded_range(f, l) ∧ l - f < 2^64
    // Предусловие: partially_associative(op)
    // Предусловие: (∀x ∈ [f, l)) fun(x) определено
    counter_machine<Op> c(op, z);
    while (f != l) {
        c(fun(f));
        f = successor(f);
    }
    transpose_operation<Op> t_op(op);
    return reduce_nonzeroes(c.f, c.l, t_op, z);
}
```

Значения в более высоких позициях счетчика соответствуют более ранним элементам исходного интервала, и операция не обязательно является комму-

[5]Выбор 64 элементов для массива подходит для любого приложения в 64-битовых архитектурах.

тативной. Поэтому мы должны использовать транспонированную версию этой операции, которую получаем с использованием следующего функционального объекта:

```
template<typename Op>
    requires(BinaryOperation(Op))
struct transpose_operation
{
    Op op;
    transpose_operation(Op op) : op(op) { }
    typedef Domain(Op) T;
    T operator()(const T& x, const T& y)
    {
        return op(y, x);
    }
};
```

Теперь мы можем реализовать итерационную версию стабильного разбиения с помощью следующей процедуры:

```
template<typename I, typename P>
    requires(ForwardIterator(I) && UnaryPredicate(P) &&
        ValueType(I) == Domain(P))
I partition_stable_iterative(I f, I l, P p)
{
    // Предусловие: bounded_range(f, l) ∧ l − f < 2^64
    return reduce_balanced(
        f, l,
        combine_ranges<I>,
        partition_trivial<I, P>(p),
        pair<I, I>(f, f)
      ).m0;
}
```

$\mathsf{pair}_{\mathrm{I,I}}(\mathrm{f}, \mathrm{f})$ является хорошим способом представления нейтрального элемента, поскольку его возврат никогда не происходит с применением partition_trivial или операции объединения.

Этот итеративный алгоритм создает иное дерево приведения, чем рекурсивный алгоритм. Если размер задачи равен 2^{k}, то рекурсивные и итерационные версии выполняют одну и ту же последовательность операций объединения; в ином случае итерационная версия может потребовать вплоть до линейного объема дополнительной работы. Например, в некоторых алгоритмах сложность составляет от $\mathrm{n}\log_2 \mathrm{n}$ до $\mathrm{n}\log_2 \mathrm{n} + \frac{\mathrm{n}}{2}$.

Упражнение 11.11. Реализуйте итерационную версию функции sort_linked_nonempty_n из главы 8, используя reduce_balanced.

Упражнение 11.12. Реализуйте итерационную версию функции reverse_n_adaptive из главы 10, используя reduce_balanced.

Упражнение 11.13. Воспользуйтесь reduce_balanced, чтобы реализовать итерационную и адаптивную к памяти версию partition_stable_n.

11.3. Слияние

В главе 9 были представлены алгоритмы копирования путем слияния, в которых два увеличивающихся интервала объединяются в третий увеличивающийся интервал. Что касается сортировки, то полезно иметь переупорядочение, которое производит слияние двух смежных увеличивающихся интервалов в единственный увеличивающийся интервал. При наличии буфера, размер которого равен размеру первого интервала, можно воспользоваться следующей процедурой[6]:

```
template<typename I, typename B, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B) &&
        ValueType(I) == ValueType(B) &&
        Relation(R) && ValueType(I) == Domain(R))
I merge_n_with_buffer(I f0, DistanceType(I) n0,
                      I f1, DistanceType(I) n1, B f_b, R r)
{
    // Предусловие: mergeable(f0, n0, f1, n1, r)
    // Предусловие: mutable_counted_range(f_b, n0)
    copy_n(f0, n0, f_b);
    return merge_copy_n(f_b, n0, f1, n1, f0, r).m2;
}
```

где свойство mergeable определено следующим образом:

property(I : *ForwardIterator*, N : *Integer*, R : *Relation*)
requires(*Mutable*(I) ∧ ValueType(I) = Domain(R))
mergeable : I × N × I × N × R
$(f_0, n_0, f_1, n_1, r) \mapsto f_0 + n_0 = f_1 \wedge$
mutable_counted_range$(f_0, n_0 + n_1) \wedge$
weak_ordering$(r) \wedge$
increasing_counted_range$(f_0, n_0, r) \wedge$
increasing_counted_range(f_1, n_1, r)

Лемма 11.12. Постусловием для merge_n_with_buffer является

$$\text{increasing\_counted\_range}(f_0, n_0 + n_1, r)$$

[6]Решение упр. 9.5 служит объяснением необходимости извлечения элемента m2.

Слияние является *стабильным*, если в выходном интервале сохраняется относительный порядок эквивалентных элементов и в каждом входном интервале, и между первым и вторым входными интервалами.

Лемма 11.13. merge_n_with_buffer является стабильной.

Следует учитывать, что merge_linked_nonempty, merge_copy и merge_copy_backward также стабильны.

Мы можем отсортировать интервал с помощью буфера, размер которого равен половине размера интервала[7]:

```
template<typename I, typename B, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B) &&
        ValueType(I) == ValueType(B) &&
        Relation(R) && ValueType(I) == Domain(R))
I sort_n_with_buffer(I f, DistanceType(I) n, B f_b, R r)
{
    // Предусловие: mutable_counted_range(f, n) ∧ weak_ordering(r)
    // Предусловие: mutable_counted_range(f_b, ⌈n/2⌉)
    DistanceType(I) h = half_nonnegative(n);
    if (zero(h)) return f + n;
    I m = sort_n_with_buffer(f, h,     f_b, r);
          sort_n_with_buffer(m, n - h, f_b, r);
    return merge_n_with_buffer(f, h, m, n - h, f_b, r);
}
```

Лемма 11.14. Постусловием для sort_n_with_buffer является

$$\mathrm{increasing\_counted\_range}(f, n, r)$$

Алгоритм сортировки является *стабильным*, если он сохраняет относительный порядок элементов с эквивалентными значениями.

Лемма 11.15. sort_n_with_buffer является стабильной.

Алгоритм имеет $\lceil \log_2 n \rceil$ уровней рекурсии. На каждом уровне выполняется самое большее $3n/2$ присваиваний с общим количеством, ограниченным значением $\frac{3}{2} n \lceil \log_2 n \rceil$. На i-м уровне снизу количество сравнений в наихудшем случае равно $n - \frac{n}{2^i}$, что позволяет получить следующее граничное значение количества сравнений:

$$n\lceil \log_2 n \rceil - \sum_{i=1}^{\lceil \log_2 n \rceil} \frac{n}{2^i} \approx n\lceil \log_2 n \rceil - n$$

[7] Аналогичный алгоритм был впервые описан в лекции Джона В. Мочли (John W. Mauchly) “Сортировка и упорядочение” [Mauchly 1946].

Если доступен буфер достаточного размера, алгоритм sort_n_with_buffer является эффективным. Если доступно меньше памяти, может использоваться алгоритм слияния, адаптивный к памяти. Дополнительное разбиение первого подынтервала в середине и использование среднего элемента для дополнительного разбиения второго подынтервала в его нижней граничной точки приводит к созданию четырех подынтервалов, r_0, r_1, r_2 и r_3, таких, что значения в r_2 строго меньше, чем значения в r_1. Вращение интервалов r_1 и r_2 приводит к созданию двух новых подзадач слияния (r_0 с r_2 и r_1 с r3):

```
template<typename I, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
void merge_n_step_0(I f0, DistanceType(I) n0,
                    I f1, DistanceType(I) n1, R r,
                    I& f0_0, DistanceType(I)& n0_0,
                    I& f0_1, DistanceType(I)& n0_1,
                    I& f1_0, DistanceType(I)& n1_0,
                    I& f1_1, DistanceType(I)& n1_1)
{
    // Предусловие: mergeable(f0, n0, f1, n1, r)
    f0_0 = f0;
    n0_0 = half_nonnegative(n0);
    f0_1 = f0_0 + n0_0;
    f1_1 = lower_bound_n(f1, n1, source(f0_1), r);
    f1_0 = rotate(f0_1, f1, f1_1);
    n0_1 = f1_0 - f0_1;
    f1_0 = successor(f1_0);
    n1_0 = predecessor(n0 - n0_0);
    n1_1 = n1 - n0_1;
}
```

Лемма 11.16. Вращение не изменяет относительных позиций элементов с эквивалентными значениями.

Итератор i в интервале является *центром вращения*, если его значение не меньше любого предшествующего ему значения и не больше любого следующего за ним значения.

Лемма 11.17. После применения merge_n_step_0 элемент f1_0 становится центром вращения.

Мы можем выполнить аналогичное дополнительное разбиение справа с использованием upper_bound:

```
template<typename I, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
```

```
void merge_n_step_1(I f0, DistanceType(I) n0,
                    I f1, DistanceType(I) n1, R r,
                    I& f0_0, DistanceType(I)& n0_0,
                    I& f0_1, DistanceType(I)& n0_1,
                    I& f1_0, DistanceType(I)& n1_0,
                    I& f1_1, DistanceType(I)& n1_1)
{
    // Предусловие: mergeable(f0, n0, f1, n1, r)
    f0_0 = f0;
    n0_1 = half_nonnegative(n1);
    f1_1 = f1 + n0_1;
    f0_1 = upper_bound_n(f0, n0, source(f1_1), r);
    f1_1 = successor(f1_1);
    f1_0 = rotate(f0_1, f1, f1_1);
    n0_0 = f0_1 - f0_0;
    n1_0 = n0 - n0_0;
    n1_1 = predecessor(n1 - n0_1);
}
```

Это приводит к получению следующего алгоритма, принадлежащего [Dudziński and Dydek 1981]:

```
template<typename I, typename B, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B) &&
        ValueType(I) == ValueType(B) &&
        Relation(R) && ValueType(I) == Domain(R))
I merge_n_adaptive(I f0, DistanceType(I) n0,
                   I f1, DistanceType(I) n1,
                   B f_b, DistanceType(B) n_b, R r)
{
    // Предусловие: mergeable(f0, n0, f1, n1, r)
    // Предусловие: mutable_counted_range(f_b, n_b)
    typedef DistanceType(I) N;
    if (zero(n0) || zero(n1)) return f0 + n0 + n1;
    if (n0 <= N(n_b))
        return merge_n_with_buffer(f0, n0, f1, n1, f_b, r);
    I f0_0; I f0_1; I f1_0; I f1_1;
    N n0_0; N n0_1; N n1_0; N n1_1;
    if (n0 < n1) merge_n_step_0(
                            f0, n0, f1, n1, r,
                            f0_0, n0_0, f0_1, n0_1,
                            f1_0, n1_0, f1_1, n1_1);
    else         merge_n_step_1(
                            f0, n0, f1, n1, r,
                            f0_0, n0_0, f0_1, n0_1,
                            f1_0, n1_0, f1_1, n1_1);
```

```
            merge_n_adaptive(f0_0, n0_0, f0_1, n0_1,
                             f_b, n_b, r);
    return merge_n_adaptive(f1_0, n1_0, f1_1, n1_1,
                             f_b, n_b, r);
}
```

Лемма 11.18. merge_n_adaptive завершает свою работу с получением увеличивающегося интервала.

Лемма 11.19. merge_n_adaptive является стабильной.

Лемма 11.20. Количество уровней рекурсии составляет самое большее $\lfloor \log_2(\mathsf{min}(\mathfrak{n}0, \mathfrak{n}1)) \rfloor + 1$.

Используя merge_n_adaptive, можем реализовать следующую процедуру сортировки:

```
template<typename I, typename B, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Mutable(B) && ForwardIterator(B) &&
        ValueType(I) == ValueType(B) &&
        Relation(R) && ValueType(I) == Domain(R))
I sort_n_adaptive(I f, DistanceType(I) n,
                  B f_b, DistanceType(B) n_b, R r)
{
    // Предусловие: mutable_counted_range(f, n) ∧ weak_ordering(r)
    // Предусловие: mutable_counted_range(f_b, n_b)
    DistanceType(I) h = half_nonnegative(n);
    if (zero(h)) return f + n;
    I m = sort_n_adaptive(f, h,     f_b, n_b, r);
          sort_n_adaptive(m, n - h, f_b, n_b, r);
    return merge_n_adaptive(f, h, m, n - h, f_b, n_b, r);
}
```

Упражнение 11.14. Определите формулы для количества присваиваний и количества сравнений как функций от размера входного и буферного интервалов. В [Dudziński and Dydek 1981] содержится тщательный анализ сложности случая, в котором буфер не применяется.

В завершение представим следующий алгоритм:

```
template<typename I, typename R>
    requires(Mutable(I) && ForwardIterator(I) &&
        Relation(R) && ValueType(I) == Domain(R))
I sort_n(I f, DistanceType(I) n, R r)
{
    // Предусловие: mutable_counted_range(f, n) ∧ weak_ordering(r)
    temporary_buffer<ValueType(I)> b(half_nonnegative(n));
```

```
    return sort_n_adaptive(f, n, begin(b), size(b), r);
}
```

Этот алгоритм работает на интервалах с минимальными требованиями к итераторам, является стабильным и эффективным, даже если функция temporary_buffer способна распределить лишь несколько процентов от запрашиваемой памяти.

Проект 11.1. Разработайте библиотеку алгоритмов сортировки, созданных из абстрактных компонентов. Спроектируйте эталонный тест для анализа их производительности при различных размерах массивов, элементов и буферов. Снабдите библиотеку документацией с рекомендациями, касающимися обстоятельств, при которых каждый алгоритм является приемлемым.

11.4. Резюме

Сложные алгоритмы могут быть разложены на более простые абстрактные компоненты с тщательно определенными интерфейсами. Обнаруженные таким образом компоненты могут использоваться для реализации других алгоритмов. Итеративный процесс, переходящий от простого к сложному и обратно, является основным способом составления систематического каталога эффективных компонентов.

Глава 12
Составные объекты

В главах 6–11 представлены алгоритмы, работающие с коллекциями объектов (структурами данных) с применением итераторов или координатных структур; при этом не рассматривались вопросы построения, уничтожения и структурного изменения таких коллекций: сами коллекции не рассматривались как объекты. В настоящей главе приведены примеры составных объектов, начиная с пар и массивов постоянного размера и заканчивая таксономией реализаций динамических последовательностей. Будет описана общая схема составного объекта, содержащего другие объекты в качестве своих частей. В заключение будет показан механизм, обеспечивающий эффективное поведение алгоритмов переупорядочения на вложенных составных объектах.

12.1. Простые составные объекты

Чтобы проще было понять, как распространить понятие регулярности на составные объекты, начнем с некоторых простых случаев. В главе 1 было представлено понятие конструктора типа pair, который после получения двух типов, T_0 и T_1, возвращает тип структуры pair_{T_0,T_1}. Мы реализовываем pair с помощью шаблона структуры вместе с некоторыми глобальными процедурами:

```
template<typename T0, typename T1>
    requires(Regular(T0) && Regular(T1))
struct pair
{
    T0 m0;
    T1 m1;
    pair() { } // стандартный конструктор
    pair(const T0& m0, const T1& m1) : m0(m0), m1(m1) { }
};
```

Язык C++ обеспечивает выполнение стандартным конструктором стандартного формирования обоих элементов, гарантируя то, что они находятся в ча-

стично сформированных состояниях, поэтому есть возможность присваивать им значения или уничтожать их. Язык C++ автоматически генерирует конструктор копии и присваивание, которые копируют или присваивают значение каждому элементу, а также автоматически генерирует деструктор, который вызывает деструктор для каждого элемента. Мы должны обеспечить вручную равенство и упорядочение:

```
template<typename T0, typename T1>
    requires(Regular(T0) && Regular(T1))
bool operator==(const pair<T0, T1>& x, const pair<T0, T1>& y)
{
    return x.m0 == y.m0 && x.m1 == y.m1;
}

template<typename T0, typename T1>
    requires(TotallyOrdered(T0) && TotallyOrdered(T1))
bool operator<(const pair<T0, T1>& x, const pair<T0, T1>& y)
{
    return x.m0 < y.m0 || (!(y.m0 < x.m0) && x.m1 < y.m1);
}
```

Упражнение 12.1. Реализуйте стандартное упорядочение less для $\mathsf{pair}_{T0,T1}$, используя стандартные упорядочения для T0 и T1 применительно к ситуациям, в которых оба типа элемента не полностью упорядочены.

Упражнение 12.2. Реализуйте $\mathsf{triple}_{T_0,T_1,T_2}$.

Разумеется, pair — неоднородный конструктор типа, а array_k — однородный конструктор типа, который после получения целого числа k и типа T возвращает тип последовательности постоянного размера $\mathsf{array\_k}_{k,T}$:

```
template<int k, typename T>
    requires(0 < k && k <= MaximumValue(int) / sizeof(T) &&
        Regular(T))
struct array_k
{
    T a[k];
    T& operator[](int i)
    {
        // Предусловие: 0 ⩽ i < k
        return a[i];
    }
};
```

Требование к k определено в терминах атрибутов типов. Функция MaximumValue(N) возвращает максимальное значение, представимое целочисленным типом N, а sizeof — встроенный атрибут типа, который возвращает

размер типа. Язык C++ генерирует стандартный конструктор, конструктор копии, присваивание и деструктор для array_k с правильной семантикой. Мы реализуем функцию-член, которая обеспечивает чтение или запись x[i][1].

IteratorType(array_$k_{k,T}$) определена как указатель на T. Мы предоставляем процедуры для возврата первого элемента и предела элементов массива[2]:

```
template<int k, typename T>
    requires(Regular(T))
pointer(T) begin(array_k<k, T>& x)
{
    return addressof(x.a[0]);
}

template<int k, typename T>
    requires(Regular(T))
pointer(T) end(array_k<k, T>& x)
{
    return addressof(x.a[k]);
}
```

Объект x типа array_$k_{k,T}$ может быть инициализирован как копия счетного интервала $[\![f, k)\!)$ с применением примерно такого кода:

```
copy_n(f, k, begin(x));
```

Мы не знаем, как реализовать подходящий конструктор инициализации, который позволил бы избежать необходимости осуществляемого автоматически стандартного формирования каждого элемента массива. Кроме того, copy_n принимает итератор любой категории и возвращает предельный итератор, но отсутствует способ возврата предельного итератора из конструктора копии.

Равенство и упорядочение для массивов реализуются с использованием лексикографических расширений, введенных в главе 7:

```
template<int k, typename T>
    requires(Regular(T))
bool operator==(const array_k<k, T>& x, const array_k<k, T>& y)
{
    return lexicographical_equal(begin(x), end(x),
                                 begin(y), end(y));
}
```

[1]Как и при использовании begin и end, для полной реализации необходима перегрузка по отношению к постоянному значению.

[2]Полная реализация должна включать постоянный тип итератора и постоянный указатель на T, наряду с версиями begin и end с перегрузкой по отношению к постоянному значению array_k, которая возвращает постоянный тип итератора.

```
template<int k, typename T>
    requires(Regular(T))
bool operator<(const array_k<k, T>& x, const array_k<k, T>& y)
{
    return lexicographical_less(begin(x), end(x),
                                begin(y), end(y));
}
```

Упражнение 12.3. Реализуйте версии $=$ и $<$ для $\mathsf{array\_k}_{\mathsf{k,T}}$, которые формируют встроенный развернутый код для небольших k.

Упражнение 12.4. Реализуйте стандартное упорядочение less для $\mathsf{array\_k}_{\mathsf{k,T}}$.

Мы предоставляем процедуру для возврата количества элементов в массиве:

```
template<int k, typename T>
    requires(Regular(T))
int size(const array_k<k, T>& x)
{
    return k;
}
```

и еще одну процедуру для определения того, равен ли размер 0:

```
template<int k, typename T>
    requires(Regular(T))
bool empty(const array_k<k, T>& x)
{
    return false;
}
```

Мы постарались определить size и empty таким образом, чтобы с помощью array_k можно было моделировать концепцию *Sequence*, которая будет определена позже.

Упражнение 12.5. Дополните array_k, чтобы можно было принимать значения $\mathsf{k} = 0$.

array_k моделирует концепцию *Linearizable*:

$$
\begin{aligned}
&\mathit{Linearizable}(\mathsf{W}) \triangleq \\
&\quad \mathit{Regular}(\mathsf{W}) \\
&\quad \wedge\ \mathsf{IteratorType} : \mathit{Linearizable} \to \mathit{Iterator} \\
&\quad \wedge\ \mathsf{ValueType} : \mathit{Linearizable} \to \mathit{Regular} \\
&\qquad\qquad \mathsf{W} \mapsto \mathsf{ValueType}(\mathsf{IteratorType}(\mathsf{W})) \\
&\quad \wedge\ \mathsf{SizeType} : \mathit{Linearizable} \to \mathit{Integer} \\
&\qquad\qquad \mathsf{W} \mapsto \mathsf{DistanceType}(\mathsf{IteratorType}(\mathsf{W})) \\
&\quad \wedge\ \mathsf{begin} : \mathsf{W} \to \mathsf{IteratorType}(\mathsf{W})
\end{aligned}
$$

$\wedge$ end : $W \to$ IteratorType(W)
$\wedge$ size : $W \to$ SizeType(W)
$\quad x \mapsto \mathsf{end}(x) - \mathsf{begin}(x)$
$\wedge$ empty : $W \to$ bool
$\quad x \mapsto \mathsf{begin}(x) = \mathsf{end}(x)$
$\wedge$ [ ] : $W \times$ SizeType(W) $\to$ ValueType(W)&
$\quad (w, i) \mapsto \mathsf{deref}(\mathsf{begin}(w) + i)$

empty всегда требует постоянных затрат времени, даже если size требует линейных затрат времени. Предусловием для $w[i]$ является $0 \leqslant i \leqslant \mathsf{size}(w)$; при этом сложность определяется спецификацией типа итератора для концепций, уточняющих *Linearizable*: линейная для прямых и двунаправленных итераторов и постоянная для индексированных итераторов и итераторов с произвольным доступом.

Линеаризуемый тип описывает интервал итераторов с помощью стандартных функций begin и end, но в отличие от array_k копирование линеаризуемого типа не должно требовать копирования основополагающих объектов; как будет показано ниже, это не *контейнер*, — последовательность, которая владеет своими элементами. Например, следующий тип моделирует концепцию *Linearizable* и не является контейнером; он определяет ограниченный интервал итераторов, находящийся в некоторой структуре данных:

```
template<typename I>
    requires(Readable(I) && Iterator(I))
struct bounded_range {
    I f;
    I l;
    bounded_range() { }
    bounded_range(const I& f, const I& l) : f(f), l(l) { }
    const ValueType(I)& operator[](int i)
    {
        // Предусловие: 0 ⩽ i < l − f
        return source(f + i);
    }
};
```

Язык C++ автоматически генерирует конструктор копии, присваивание и деструктор с той же семантикой, что и для $\mathsf{pair}_{I,I}$. Если T представляет собой $\mathsf{bounded\_range}_I$, то IteratorType(T) определяется как I, а SizeType(T) определяется как DistanceType(I).

Несложно определить процедуры, связанные с итератором:

```
template<typename I>
    requires(Readable(I) && Iterator(I))
I begin(const bounded_range<I>& x) { return x.f; }
```

```
template<typename I>
    requires(Readable(I) && Iterator(I))
I end(const bounded_range<I>& x) { return x.l; }

template<typename I>
    requires(Readable(I) && Iterator(I))
DistanceType(I) size(const bounded_range<I>& x)
{
    return end(x) - begin(x);
}

template<typename I>
    requires(Readable(I) && Iterator(I))
bool empty(const bounded_range<I>& x)
{
    return begin(x) == end(x);
}
```

В отличие от array_k, равенство для bounded_range не реализовано с использованием лексикографического равенства, но вместо этого, по существу, объект рассматривается как пара итераторов и проводится сравнение соответствующих значений:

```
template<typename I>
    requires(Readable(I) && Iterator(I))
bool operator==(const bounded_range<I>& x,
                const bounded_range<I>& y)
{
    return begin(x) == begin(y) && end(x) == end(y);
}
```

Равенство, определенное таким образом, соответствует конструктору копии, сгенерированному языком C++, в котором оно трактуется так же, как и для пары итераторов. Рассмотрим тип W, который моделирует *Linearizable*. Если W — контейнер с линейной структурой координат, то lexicographical_equal служит для него правильно заданным равенством, как было определено для array_k. Если W — однородный контейнер, структура координат которого нелинейна (например, дерево или матрица), то ни lexicographical_equal, ни *равенство интервалов* (как было определено для bounded_range) не являются правильно заданным равенством, хотя алгоритм lexicographical_equal все еще может оказаться полезным. Если W — не контейнер, а лишь описание интервала, принадлежащего другой структуре данных, то равенство интервалов является для него правильно заданным равенством.

Стандартное полное упорядочение для $\text{bounded\_range}_{\text{I}}$ определено лексикографически на паре итераторов с использованием стандартного полного упорядочения для I:

```
template<typename I>
    requires(Readable(I) && Iterator(I))
struct less< bounded_range<I> >
{
    bool operator()(const bounded_range<I>& x,
                    const bounded_range<I>& y)
    {
        less<I> less_I;
        return less_I(begin(x), begin(y)) ||
               (!less_I(begin(y), begin(x)) &&
                 less_I(end(x), end(y)));
    }
};
```

Даже если тип итератора не имеет естественного полного упорядочения, он должен предусматривать стандартное полное упорядочение: например, путем трактовки битового шаблона как целого числа без знака. pair и array_k — примеры очень широкого класса *составных объектов*. Объект является *составным объектом*, если он состоит из других объектов, называемых его *частями*. Связь "целое–часть" удовлетворяет четырем свойствам, *связность*, *нецикличность*, *непересекаемость* и *принадлежность*. *Связность* означает, что объект имеет присоединенную структуру координат, которая позволяет достигать каждой части объекта из начального адреса объекта. *Нецикличность* означает, что объект не является подчастью самого себя, притом что *подчасти* объекта являются его частями и подчастями его частей. (Из нецикличности следует, что ни один объект не является частью самого себя.) *Непересекаемость* означает, что если два объекта имеют общую подчасть, то один из этих двух объектов является подчастью другого. *Принадлежность* означает, что копирование объекта приводит к копированию его частей, а уничтожение объекта вызывает уничтожение его частей. Составной объект является *динамическим*, если множество его частей может изменяться на протяжении его существования.

Мы именуем тип, к которому относится составной объект, типом составного объекта, а концепцию, моделируемую типом составного объекта, — концепцией составного объекта. На составных объектах как таковых не могут быть определены какие-либо алгоритмы, поскольку составной объект — это схема концепций, а не концепция.

array_k является моделью концепции *Sequence*: концепции составного объекта, который уточняет *Linearizable* и интервал элементов которого определяет его части:

$$\begin{aligned} &\mathit{Sequence}(\mathrm{S}) \triangleq \\ &\quad \mathit{Linearizable}(\mathrm{S}) \\ &\quad \wedge\ (\forall \mathrm{s} \in \mathrm{S})\,(\forall \mathrm{i} \in [\mathsf{begin}(\mathrm{s}), \mathsf{end}(\mathrm{s}))\,\mathsf{deref}(\mathrm{i}) \text{ является частью } \mathrm{s} \end{aligned}$$

$$\begin{aligned}
&\wedge\ = : \mathsf{S} \times \mathsf{S} \to \mathsf{bool} \\
&\qquad (\mathsf{s}, \mathsf{s}') \mapsto \mathsf{lexicographical_equal}(\\
&\qquad\qquad \mathsf{begin}(\mathsf{s}), \mathsf{end}(\mathsf{s}), \mathsf{begin}(\mathsf{s}'), \mathsf{end}(\mathsf{s}')) \\
&\wedge\ < : \mathsf{S} \times \mathsf{S} \to \mathsf{bool} \\
&\qquad (\mathsf{s}, \mathsf{s}') \mapsto \mathsf{lexicographical_less}(\\
&\qquad\qquad \mathsf{begin}(\mathsf{s}), \mathsf{end}(\mathsf{s}), \mathsf{begin}(\mathsf{s}'), \mathsf{end}(\mathsf{s}'))
\end{aligned}$$

Если s и s' — равные, но не идентичные последовательности, то $\mathsf{begin}(\mathsf{s}) \neq \mathsf{begin}(\mathsf{s}')$, но $\mathsf{source}(\mathsf{begin}(\mathsf{s})) = \mathsf{source}(\mathsf{begin}(\mathsf{s}'))$. В этом состоит пример *регулярности проекции*. Следует отметить, что begin и end могут быть регулярными для концепции *Linearizable*, которая не является *Sequence*; например, они регулярны для bounded_range.

Упражнение 12.6. Определите свойство projection_regular_function.

12.2. Динамические последовательности

$\mathsf{array\_k}_{\mathsf{k},\mathsf{T}}$ является *последовательностью постоянного размера*: параметр k определяется во время компиляции и применяется ко всем объектам этого типа. Мы не определяем соответствующую концепцию для последовательностей постоянного размера, поскольку не знаем, имеются ли другие полезные модели. Аналогичным образом, мы не определяем концепцию для *последовательности постоянного размера*, размер которой определяется во время формирования. Все известные нам структуры данных, которые моделируют последовательность постоянного размера, моделируют также *последовательность динамического размера*, размер которой изменяется по мере вставки или стирания элементов. (Однако есть и составные объекты постоянного размера; примером могут служить квадратные матрицы $\mathsf{n} \times \mathsf{n}$.)

Вне зависимости от конкретной структуры данных, требования к регулярным типам диктуют стандартное поведение для динамической последовательности. При уничтожении структуры данных все ее элементы уничтожаются, а занимаемые ими ресурсы освобождаются. Равенство и полное упорядочение на динамических последовательностях определяются лексикографически, так же, как и для array_k. При присваивании значений динамической последовательности она становится равной, но непересекающейся с правой стороной; аналогичным образом конструктор копии создает равную, но непересекающуюся последовательность.

Если s — имеющая динамический размер, или просто *динамическая* последовательность с размером $\mathsf{n} \geqslant 0$, то *вставка* интервала r с размером k в *индекс вставки* i приводит к увеличению размера до $\mathsf{n} + \mathsf{k}$. Индекс вставки i может представлять собой любое из значений $\mathsf{n} + 1$ в замкнутом интервале

$[0, n]$. Если s' — значение последовательности после вставки, то

$$s'[j] = \begin{cases} s[j] & \text{if } 0 \leqslant j < i \\ r[j-i] & \text{if } i \leqslant j < i+k \\ s[j-k] & \text{if } i+k \leqslant j < n+k \end{cases}$$

Аналогичным образом, если s — последовательность с размером $n \geqslant k$, то *стирание* k элементов в *индексе стирания* i приводит к уменьшению размера до $n-k$. Индекс стирания i может представлять собой любое из значений $n-k$ в открытом интервале $[0, n-k)$. Если s' — значение последовательности после стирания, то

$$s'[j] = \begin{cases} s[j] & \text{if } 0 \leqslant j < i \\ s[j+k] & \text{if } i \leqslant j < n-k \end{cases}$$

В связи с необходимостью обеспечивать вставку и стирание элементов приходится создавать многочисленные разновидности последовательных структур данных на основе различных компромиссных решений, касающихся сложности применяемых процедур вставки и стирания. Все эти категории зависят от присутствия *отдельно расположенных* частей. Часть является расположенной отдельно, если она не расположена с постоянным смещением от адреса определенного объекта, но должна быть достигнута посредством обхода в координатной структуре объекта, начиная с его *заголовка*. Заголовок составного объекта — это коллекция его *локальных* частей, иными словами, частей, расположенных с постоянными смещениями от начального адреса объекта. Количество локальных частей в объекте — это постоянная, определяемая его типом.

В этом разделе мы суммируем свойства последовательных структур данных, относящихся к фундаментальным категориям: *связанные* и *основанные на экстенте*.

Связанные структуры данных соединяют данные с помощью указателей, которые служат в качестве связей. Каждый элемент находится в одной из отдельных частей *с постоянным размещением*: на протяжении срока существования элемента его адрес никогда не изменяется. По аналогии с элементом, часть содержит связи, соединяющие ее со смежными частями. Итераторы относятся к категории связанных итераторов; индексированные итераторы не поддерживаются. Возможно применение операций вставки и стирания, характеризующихся постоянными затратами времени, поскольку они реализуются на основе операций повторного связывания, поэтому их применение не приводит к тому, что итераторы становятся недействительными. Имеются две основные разновидности связных списков: односвязный и двухсвязный.

Односвязный список имеет связанный *ForwardIterator*. Стоимость вставки и стирания элемента после указанного итератора является постоянной, тогда как стоимость вставки и стирания перед позицией произвольного итератора линейно зависит от расстояния до начала списка. Поэтому стоимость вставки и стирания в начале списка постоянна. Имеется несколько разновидностей односвязных списков, отличающихся по структуре заголовка и типу связи с последним элементом. Заголовок *базового* списка состоит из связи с первым элементом или специального значения *null*, указывающего, что список пуст; связь с последним элементом имеет значение null. Заголовок *циклического* списка состоит из связи с последним элементом или имеет значение null, указывающее на пустой список; связь с последним элементом указывает на первый элемент. Заголовок списка *"первый–последний"* состоит из двух частей: заголовок базового списка, оканчивающегося значением null, и связь с последним элементом списка или null, если список пуст.

На выбор реализации односвязного списка влияют несколько факторов. Меньший заголовок хорошо подходит для приложения с большим количеством списков, многие из которых пусты. Размер итератора для циклического списка больше, а операция successor с ним выполняется медленнее, поскольку необходимо проводить различие между указателем на первый элемент и указателем на предельное значение. Структура данных, поддерживающая вставку с постоянными затратами времени в начальной части списка, может использоваться в качестве очереди или двусторонней очереди с ограничением по выходу. В следующей таблице приведены сводные данные о том, как расцениваются эти факторы с точки зрения реализации.

Разновидность	Однословный заголовок	Простой итератор	Обратная вставка
Простой	да	да	нет
Циклический	да	нет	да
"Первый–последний"	нет	да	да

Двухсвязный список имеет связанный *BidirectionalIterator*. Стоимость вставки до или после стирания в позиции произвольного итератора постоянна. Как и в случае односвязных списков, имеется несколько разновидностей двухсвязных списков. Заголовок *циклического* списка состоит из указателя на первый элемент или значения null, указывающего на пустой список; обратная связь первого элемента указывает на последний элемент, а прямая связь последнего элемента указывает на первый элемент. Список *с фиктивным узлом* подобен циклическому списку, но имеет дополнительный фиктивный узел между последним и первым элементами; заголовок состоит из связи к фиктивному узлу, который позволяет опустить фактический объект данных. *Заголовок с двумя указателями* аналогичен списку с фиктивным узлом, но его

заголовок состоит из двух указателей, соответствующих связям фиктивного узла.

Два фактора, от которых зависит выбор реализации односвязного списка, являются значимыми и для реализаций двухсвязных списков; речь идет о размере заголовка и сложности итератора. Для двухсвязных списков характерны также дополнительные проблемы. Некоторые алгоритмы могут быть упрощены, если список имеет итератор с постоянным пределом, поскольку этот предел может использоваться в качестве значения, отличного от любого действительного итератора, на протяжении всего срока существования списка. Как будет показано ниже в этой главе, из-за присутствия связей от отдельно расположенных частей до локальных частей выполнение переупорядочения становится более дорогостоящим для элементов, которые относятся к типу списка. В следующей таблице приведены сводные данные о том, как расцениваются эти факторы с точки зрения реализации:

Разновидность	**Однословный заголовок**	**Простой итератор**	**Без связей от отдельных к локальным**	**Постоянный предел**
Циклический	да	нет	да	нет
С фиктивным узлом	да	да	да	нет[3]
Заголовок с двумя указателями	нет	да	нет	да

В главе 8 было представлено понятие переупорядочений связей; под этим подразумевается переопределение взаимосвязей связанных итераторов в одном или нескольких связанных интервалах без создания или уничтожения итераторов либо изменения отношений между итераторами и объектами, которые они определяют. Переупорядочения связей могут ограничиваться одним списком или охватывать несколько списков; в этом случае принадлежность элементов изменяется. Например, split_linked может использоваться для перемещения элементов, удовлетворяющих предикату, из одного списка в другой, а combine_linked_nonempty может использоваться для перемещения элементов из одного списка в полученные путем слияния позиции в другом списке. *Сращивание* — это переупорядочение связей, в котором стирается некоторый интервал в одном списке и снова вставляется в другой список.

Обратные связи в связанной структуре не используются в алгоритмах, подобных сортировке. Однако такие связи обеспечивают стирание и вставку элементов в произвольной позиции с постоянными затратами времени, что обходится более дорого в односвязной структуре. Наиболее значимой причиной

[3]Если фиктивный узел распределен, даже если список пуст, то список имеет постоянный предел; к сожалению, при этом нарушается желаемое свойство пустых структур данных — не иметь отдельных частей, — позволяющее конструировать их без дополнительных ресурсов.

для выбора связной структуры часто бывает эффективность вставки и удаления, поэтому следует наряду с этим рассмотреть возможность применения двунаправленных связей.

Структуры данных *на основе экстентов* группируют элементы в одном или нескольких *экстентах*, или отдельно расположенных блоках частей данных, а также обеспечивают произвольный доступ к ним. Вставка и стирание в произвольной позиции требуют затрат времени, пропорциональных размеру последовательности, тогда как вставка и стирание в конце, а также, возможно, в начале занимают амортизированное постоянное время[4]. В зависимости от того, какие конкретные правила применяются для той или иной реализации, после вставки и стирания некоторые итераторы становятся недействительными; иными словами, ни один элемент не имеет постоянного места. В некоторых структурах данных на основе экстентов используется *один экстент*, тогда как другие являются *сегментированными*, т. е. в них используются многочисленные экстенты, а также дополнительные индексные структуры.

В массиве с единственным экстентом этот экстент должен присутствовать, только если массив имеет размер, отличный от нуля. Для предотвращения перераспределения при каждой вставке экстент содержит резервную область; после того как резервная область исчерпывается, экстент перераспределяется. Заголовок содержит указатель на экстент; дополнительные указатели отслеживают данные, а резервные области обычно находятся в префиксе экстента. Благодаря размещению дополнительных указателей в префиксе, а не в заголовке улучшаются характеристики и пространственной, и временной сложности, когда массивы являются вложенными.

Имеется несколько разновидностей массивов с единственным экстентом. В *одностороннем* массиве данные начинаются с фиксированного смещения в экстенте, и за ними следует резервная область[5]. В *двустороннем* массиве данные находятся в середине экстента, а резервные области окружают их с обеих сторон; если в результате роста с любой из сторон соответствующая резервная область исчерпывается, экстент перераспределяется. В *циклическом* массиве экстент рассматривается так, как если бы последователь для его самого высокого адреса имел самый низкий его адрес; поэтому единственная резервная область всегда логически предшествует и следует за данными, которые могут расти в обоих направлениях.

На выбор реализации массива с единственным экстентом влияет несколько факторов. Для одно- и двусторонних массивов наиболее эффективной реализа-

[4] *Амортизированная* сложность операции представляет собой сложность, усредненную по последовательности операций в наихудшем случае. Понятие амортизированной сложности было введено в [Tarjan 1985].

[5] Безусловно, возможно обеспечить наращивание данных с конца в направлении вниз, но, по-видимому, на практике это не принесет особой пользы.

цией итераторов являются машинные адреса; итератор для кольцевого массива больше, а его функция обхода действует медленнее в связи с необходимостью следить за тем, не произошел ли перенос используемой области в начало экстента. Структура данных, поддерживающая вставку/стирание в начале с постоянными затратами времени, может применяться в качестве очереди или двусторонней очереди с ограничением по выходу. Двусторонний массив может требовать перераспределения, даже если в одной из двух его резервных областей имеется доступное пространство; односторонний или кольцевой массив требует перераспределения только после исчерпания резерва.

Разновидность	**Простой итератор**	**Со вставкой/стиранием в начале**	**С эффективным перераспределением**
Односторонний	Да	Нет	Да
Двусторонний	Да	Да	Нет
Циклический	Нет	Да	Да

Если происходит вставка, притом что экстент одностороннего или кольцевого массива заполнен, выполняется *перераспределение*: распределяется больший экстент и существующие элементы перемещаются в новый экстент. В случае двустороннего массива при вставке, исчерпывающей резерв на одном конце массива, требуется либо перераспределение, либо перемещение элементов на другой конец для переброски оставшегося резерва. Перераспределение (и перемещение элементов в двустороннем массиве) приводит к тому, что все итераторы, указывающие на массив, становятся недействительными.

Если происходит перераспределение, то увеличение размера экстента на мультипликативный коэффициент приводит к тому, что над каждым элементом выполняются операции формирования, количество которых в среднем измеряется амортизированной постоянной величиной. Проведенные нами эксперименты показывают, что коэффициент, равный 2, позволяет успешно достичь компромисса между уменьшением амортизированного количества операций формирования в расчете на каждый элемент и использованием памяти.

Упражнение 12.7. Выведите выражения для использования памяти и количества операций формирования в расчете на каждый элемент для различных мультипликативных коэффициентов.

Проект 12.1. Объедините теоретический анализ с экспериментированием, чтобы определить оптимальные стратегии перераспределения для массивов с единственным экстентом при различных реалистичных рабочих нагрузках.

Для одностороннего или циклического массива с одним экстентом $\mathfrak{a}$ предусмотрена функция $\mathsf{capacity}$, такая, что $\mathsf{size}(\mathfrak{a}) \leqslant \mathsf{capacity}(\mathfrak{a})$, а вставка в $\mathfrak{a}$ приводит к перераспределению, только если размер после вставки больше емкости перед вставкой. Имеется также процедура $\mathsf{reserve}$, которая позволяет увеличить емкость массива до указанной величины.

Упражнение 12.8. Разработайте интерфейс, позволяющий управлять емкостью и резервом, для двусторонних массивов.

Сегментированный массив имеет одни или несколько экстентов, в которых хранятся элементы, и *индексную* структуру данных, управляющую указателями на экстенты. В связи с необходимостью проверять, не достигнут ли конец экстента, функция обхода с итератором действует медленнее, чем для массива с одним экстентом. Индекс должен поддерживать такое же поведение, как и в сегментированном массиве: он должен обеспечивать произвольный доступ, а также вставку и стирание в конце, а в случае необходимости в начале. Потребность в полном перераспределении никогда не возникает, поскольку после заполнения существующего экстента добавляется еще один экстент. Резервное пространство требуется только в экстентах на одном или обоих концах.

Основным источником разнообразия сегментированных массивов является структура индекса. Индекс *с одним экстентом* — это массив указателей на экстенты данных с одним экстентом; такой индекс поддерживает рост в конце, тогда как двусторонний или циклический индекс поддерживает рост с обоих концов. *Сегментированный* индекс сам является сегментированным массивом, обычно имеющим индекс с одним экстентом, но может быть также снабжен сегментированным индексом. *Кособокий* индекс имеет несколько уровней. Его корнем является единственный экстент фиксированного размера; первые несколько элементов служат указателями на экстенты данных; следующий элемент указывает на экстент с косвенным индексом, содержащий указатели на экстенты данных; очередной элемент указывает на другой экстент косвенного индекса с двойными указателями на экстенты с косвенными индексами[6] и т. д.

Проект 12.2. Разработайте полное семейство интерфейсов для динамических последовательностей. Оно должно предусматривать поддержку построения, вставки, стирания и соединения. Предусмотрите наличие вариантов для поддержки особых случаев в различных реализациях. Например, должна быть учтена возможность вставки не только после, но и до указанного итератора для обработки односвязных списков.

Проект 12.3. Реализуйте всесторонне развитую библиотеку динамических последовательностей, предоставляющую средства работы с различными односвязными, двухсвязными, имеющими один экстент и сегментированными структурами данных.

[6] В основе этого индекса лежит оригинальная файловая система UNIX [см. Thompson and Ritchie 1974].

Проект 12.4. Разработайте эталонный тест для динамических последовательностей, учитывающий реалистичные рабочие нагрузки в приложениях, определите производительность различных структур данных и предоставьте пользователю руководство по выбору подходящей структуры на основе этих результатов.

12.3. Основополагающий тип

В главах 2–5 были описаны алгоритмы на математических значениях и показано, что регулярные типы предоставляют возможность проводить рассуждения, основанные на равенствах, применительно к алгоритмам и доказательствам. В главах 6–11 представлены алгоритмы на структурах в оперативной памяти и продемонстрировано, что рассуждения, основанные на равенствах, остаются полезными в мире с изменяющимся состоянием. При этом речь шла о небольших объектах, таких как целые числа и указатели, для которых операции присваивания и копирования не являются дорогостоящими. В этой главе представлены составные объекты, которые удовлетворяют требованиям к регулярным типам, поэтому могут использоваться в качестве элементов других составных объектов. Динамические последовательности и другие составные объекты, в которых заголовок отделен от отдельно расположенных частей, предоставляют эффективный способ реализации переупорядочений: перемещение заголовков без перемещения отдельно расположенных частей.

При переупорядочении составных объектов эффективность может стать низкой; чтобы понять, с чем связана эта проблема, рассмотрим процедуру swap_basic, определенную следующим образом:

```
template<typename T>
    requires(Regular(T))
void swap_basic(T& x, T& y)
{
    T tmp = x;
    x = y;
    y = tmp;
}
```

Предположим, что мы вызываем функцию $\mathsf{swap\_basic}(\mathtt{a}, \mathtt{b})$, чтобы обменять местами две динамические последовательности. Конструирование копии и два выполняемых при этом присваивания требуют линейных затрат времени. Кроме того, в какой-то момент может возникнуть исключительная ситуация в связи с нехваткой памяти, даже если по завершении операции объем занимаемой памяти должен возвратиться к прежнему значению.

Мы можем избежать этого дорогостоящего копирования, специализируя функцию swap_basic для того, чтобы в ней осуществлялся обмен местами заголовков динамической последовательности определенного типа и, в случае необходимости, обновлялись связи от отдельно расположенных частей к заголовку. Однако при разработке специальной версии функции swap_basic возникают проблемы. Прежде всего, такую версию приходится разрабатывать отдельно для каждой структуры данных. Что еще более важно, многие алгоритмы переупорядочения не основаны на swap_basic; к ним относятся замещающие перестановки, такие как cycle_from, и алгоритмы, в которых используется буфер, наподобие merge_n_with_buffer. Наконец, есть такие ситуации, как перераспределение массива с одним экстентом, в которых объекты перемещаются из старого экстента в новый.

Мы хотим обобщить идею обмена местами заголовков на произвольные переупорядочения, что позволило бы использовать буферную память и перераспределение и продолжить писать абстрактные алгоритмы, которые не зависят от реализации управляемых ими объектов. Чтобы достичь этой цели, ассоциируем каждый регулярный тип T с его *основополагающим типом*, U = UnderlyingType(T). Тип U идентичен типу T, если тип T не имеет отдельно расположенных частей или имеет отдельно расположенные части со связями, ведущими обратно к заголовку[7]. В противном случае тип U идентичен типу T во всех отношениях, за исключением того, что он не поддерживает владение: уничтожение не влияет на отдельно расположенные части, а при конструировании копии и присваивании просто копируется заголовок без копирования отдельно расположенных частей. Если основополагающий тип отличается от исходного типа, он имеет ту же компоновку (комбинацию двоичных разрядов), что и заголовок исходного типа.

Сам факт, что одна и та же комбинация двоичных разрядов может интерпретироваться как объект некоторого типа и его основополагающего типа, позволяет нам рассматривать память как содержащую тот или другой тип, используя встроенный шаблон функции reinterpret_cast. Объекты типа UnderlyingType(T) могут использоваться для хранения только временных значений при реализации переупорядочения объектов типа T. Сложность конструирования копии и присваивания для *истинного* основополагающего типа (того, который не идентичен исходному типу) пропорциональна размеру заголовка типа T. Дополнительным преимуществом в этом случае является то, что при конструировании копии и присваивании для UnderlyingType(T) никогда не возникает исключение.

[7]Это позволяет понять смысл предупреждения в отношении связей от отдельных частей к заголовку в нашем обсуждении списков с двойными связями.

Реализация основополагающего типа для исходного типа T является несложной и может быть автоматизирована. U = UnderlyingType(T) всегда имеет ту же компоновку, что и заголовок T. Конструктор копии и присваивание для U только копируют биты; они не создают копию отдельно расположенных частей T. Например, основополагающий тип pair_{T_0,T_1} представляет собой пару, элементы которой — основополагающие типы T_0 и T_1; то же относится к кортежам других типов. Основополагающим типом для $\mathsf{array\_k}_{k,T}$ является тип $\mathsf{array\_k}_k$, элементы которого имеют основополагающий тип T.

После определения типа UnderlyingType(T) мы можем приводить ссылку на T к ссылке на UnderlyingType(T), не выполняя никаких вычислений, с помощью следующей процедуры:

```
template<typename T>
    requires(Regular(T))
UnderlyingType(T)& underlying_ref(T& x)
{
    return reinterpret_cast<UnderlyingType(T)&>(x);
}
```

Теперь мы можем эффективно менять местами составные объекты, переписав swap_basic следующим образом:

```
template<typename T>
    requires(Regular(T))
void swap(T& x, T& y)
{
    UnderlyingType(T) tmp = underlying_ref(x);
    underlying_ref(x)     = underlying_ref(y);
    underlying_ref(y)     = tmp;
}
```

что можно также выполнить с помощью такого вызова:

```
    swap_basic(underlying_ref(x), underlying_ref(y));
```

Многие алгоритмы переупорядочения можно модифицировать в целях использования с основополагающим типом путем повторной реализации exchange_values и cycle_from таким же образом, как мы повторно реализовали swap.

Для работы с другими алгоритмами переупорядочения мы используем вспомогательный итератор. Такой вспомогательный итератор выполняет такие же операции обхода, что и исходный итератор, но тип значений заменяется основополагающим типом исходного типа значения; source возвращает `underlying_ref(source(x.i))`, а sink возвращает `underlying_ref(sink(x.i))`, где `x` — вспомогательный объект, а `i` — исходный объект итератора в `x`.

Упражнение 12.9. Реализуйте подобный вспомогательный итератор, применимый для всех концепций итераторов.

Теперь мы можем повторно реализовать reverse_n_with_temporary_buffer следующим образом:

```
template<typename I>
    requires(Mutable(I) && ForwardIterator(I))
void reverse_n_with_temporary_buffer(I f, DistanceType(I) n)
{
    // Предусловие: mutable_counted_range(f, n)
    temporary_buffer<UnderlyingType(ValueType(I))> b(n);
    reverse_n_adaptive(underlying_iterator<I>(f), n,
                       begin(b), size(b));
}
```

где underlying_iterator — вспомогательный итератор, рассматриваемый в упражнении 12.9.

Проект 12.5. Систематически внедрите принцип использования основополагающего типа в какой-либо крупной библиотеке C++, такой как STL, или разработайте новую библиотеку на основе идей, изложенных в этой книге.

12.4. Резюме

Мы распространили типы структур и типы массивов постоянного размера языка C++ на динамические структуры данных на основе понятия отдельно расположенных частей. Концепции владения и регулярности определяют трактовку частей применительно к конструированию копии, присваиванию, равенству и полному упорядочению. Как было показано для случая динамических последовательностей, полезные разновидности структур данных необходимо тщательно реализовывать, классифицировать и документировать, чтобы программисты могли выбирать наиболее подходящие из них для каждого приложения. Переупорядочения на вложенных структурах данных эффективно реализуются путем временного ослабления инварианта владения.

Послесловие

В этом послесловии мы подведем итоги рассмотрения основных тем книги: регулярность, концепции, алгоритмы и их интерфейсы, методы программирования, а также значения указателей. Применительно к каждой теме обсуждаются также связанные с ней конкретные ограничения.

Регулярность

Регулярные типы определяют конструирование копии и присваивание с точки зрения равенства. Регулярные функции возвращают равные результаты, будучи применяемыми к равным аргументам. Например, регулярность преобразований позволила нам определять и рассуждать об алгоритмах анализа орбит. На свойства регулярности мы фактически опирались во всей книге, рассматривая упорядочение отношений, функцию определения последующего элемента для прямых итераторов и многие другие вопросы.

Работая со встроенными типами, мы обычно рассматриваем сложность равенства, копирования и присваивания как постоянную. Если же дело касается составных объектов, то предполагается, что сложность этих операций должна быть линейной в *области* определения объектов: общий объем памяти, включая отдельно расположенные, а также локальные части. Однако на практике надежда на то, что равенство в худшем случае будет линейным в области определения его аргументов, может не всегда оправдываться.

Например, рассмотрим представление *мультимножества*, или неупорядоченной коллекции потенциально повторных элементов, как неотсортированную динамическую последовательность. Хотя вставка нового элемента занимает постоянное время, проверка на равенство двух мультимножеств занимает время $O(\mathrm{n}\log\mathrm{n})$, которое требуется для их сортировки, а затем лексикографического сравнения. При условии, что проверка на равенство происходит не часто, с этим вполне можно смириться; однако, если такие мультимножества помещены в последовательность, в которой должен производиться поиск с помощью find, то производительность может стать неприемлемой. В качестве

примера крайне неблагоприятной рассмотрим ситуацию, в которой равенство для типа должно быть реализовано с применением изоморфизма графов — задачи, для которой неизвестен какой-либо полиномиальный алгоритм.

В разделе 1 было отмечено, что если реализация поведенческого равенства на значениях невыполнима, то часто можно реализовать представительное равенство. Для составных объектов мы часто реализовываем представительное равенство с помощью методов, описанных в разделе 7. Такое *структурное* равенство часто бывает применимым при предоставлении семантики конструирования копии и присваивания, а также может быть полезным в других целях. Напомним, что из представительного равенства следует поведенческое равенство. Аналогичным образом, хотя естественное полное упорядочение не всегда осуществимо, стандартное полное упорядочение, основанное на структуре (например, лексикографическое упорядочение для последовательностей), позволяет эффективно осуществлять сортировку и поиск. Есть, конечно, и такие объекты, для которых ни конструирование копии, ни присваивание, ни даже равенство, не имеют смысла, поскольку они владеют уникальным ресурсом.

Концепции

Мы используем концепции абстрактной алгебры (полугруппы, моноиды и модули) для описания таких алгоритмов, как возведение в степень, вычисление остатка и наибольшего общего делителя. Во многих случаях мы должны адаптировать стандартные математические понятия, чтобы они соответствовали алгоритмам. Иногда мы вводим новые концепции, такие как *HalvableMonoid*, чтобы ужесточить требования. Иногда ослабляем требования, как применительно к свойству partially_associative. Часто нам приходится сталкиваться с частичными доменами, как с предикатом области определения, передаваемым в collision_point. Математические понятия — это инструменты, которые должны использоваться и свободно модифицироваться. Эта логика также применима к концепциям информатики. Концепции итератора описывают фундаментальные свойства определенных алгоритмов и структур данных; однако есть и другие координатные структуры, описываемые концепциями, которые еще предстоит открыть. Задача программиста состоит в том, чтобы определить, полезна ли данная концепция.

Алгоритмы и их интерфейсы

Ограниченные полуоткрытые интервалы естественным образом соответствуют реализации многих структур данных и обеспечивают удобный способ представить входы и выходы для таких алгоритмов, как поиск, вращение, раз-

биение, слияние и т. д. Однако применительно к некоторым алгоритмам, таким как partition_point_n, естественным интерфейсом является счетный интервал. Даже для алгоритмов, для которых естественными являются ограниченные интервалы, обычно имеются естественные разновидности, принимающие счетные интервалы. Попытка с нашей стороны сэкономить усилия, ограничившись единственной разновидностью интерфейса, была бы неоправданной.

Три алгоритма вращения, описанные в главе 10, соответствуют трем концепциям итератора. Для каждого алгоритма мы должны установить его концептуальные требования, предусловия для его входов, и все прочие характеристики, которые необходимы для того, чтобы обеспечить его использование по назначению. Редко случается так, чтобы единственный алгоритм становился применимым при всех обстоятельствах.

Методы программирования

Использование successor, строго функционального преобразования, позволило нам написать целый ряд четких и эффективных программ. Однако в главе 9 мы предпочли инкапсулировать вызовы successor и predecessor в небольшие изменяемые машины, такие как copy_step, поскольку это способствовало созданию более четкого кода для семейства связанных алгоритмов. Точно так же оказалось приемлемым использовать `goto` в конечных автоматах в главе 8 и использовать `reinterpret_cast` для основополагающего типа в главе 12. Вместо того, чтобы ограничивать выразительную мощь компьютера и языка программирования, необходимо определить подходящую область использования для каждой доступной конструкции. Качественное программное обеспечение следует из надлежащей организации компонентов, а не из синтаксических или семантических ограничений.

Значения указателей

В настоящей книге демонстрируются два способа использования указателей: во-первых, как итераторов и других координат, представляющих промежуточные позиции в ходе работы алгоритма, и, во-вторых, как *соединителей*, обеспечивающих владение отдельно расположенными частями составного объекта. Например, в разделе 12 мы обсуждали использование указателей для соединения узлов в списке и экстентов в массиве.

Эти две разные роли для указателей определяют различное поведение при копировании объекта, его уничтожении или сравнении на равенство. При копировании объекта для создания копий отдельно расположенных частей отслеживаются связанные с ними соединители, поэтому новый объект содержит новые соединители, указывающие на скопированные части. С другой стороны,

при копировании объекта, содержащего итераторы (например, bounded_range), итераторы просто копируются, без их отслеживания. Точно так же при уничтожении объекта отслеживаются его соединители для уничтожения отдельно расположенных частей, в то время как уничтожение объекта, содержащего итераторы, не оказывает влияния на объект, на который указывают итераторы. Наконец, при проверке равенства на контейнере для сравнения соответствующих частей отслеживаются соединители, в то время как при проверке равенства на объекте, отличном от контейнера (например, bounded_range), просто происходит проверка на равенство соответствующих итераторов.

Однако есть и третий способ использования указателей: представление *отношений* между сущностями. Отношение между двумя или бóльшим количеством объектов не следует рассматривать как часть, которой владеют сами объекты; отношение имеет собственное существование и поддерживает взаимные зависимости между связываемыми им объектами. Вообще говоря, указатель, представляющий отношение, не участвует в регулярных операциях. Например, при копировании объекта не происходит отслеживание или копирование указателя отношения, поскольку отношение существует для скопированного объекта, но не для его копии. Если взаимно-однозначное отношение представлено как пара встроенных указателей, связывающих два объекта, то уничтожение любого из объектов должно приводить к очистке соответствующего указателя в другом объекте.

Разработка структур данных как составных объектов с владением и отдельно расположенными частями приводит к стилю программирования, в котором основные объекты (не являющиеся подчастями других объектов) находятся в статических переменных в течение всего выполнения программы или в локальных переменных, пока происходит выполнение блока. Динамически распределяемая память используется только для отдельно расположенных частей. Тем самым применение стековой блочной структуры Algol 60 распространяется на обработку произвольных структур данных. Такая структура естественным образом подходит для многих приложений. Однако при определенных обстоятельствах становятся в большей степени применимыми подсчет ссылок, сбор мусора или другие методы управления памятью.

Резюме

Программирование — это итеративный процесс: изучение полезных задач, поиск эффективных алгоритмов их решения, шлифовка концепций, лежащих в основе алгоритмов, и организация концепций и алгоритмов в целостную математическую теорию. Каждое новое открытие дополняет общую совокупность знаний, но путь научного развития остается непройденным до конца.

Глава A

Математическая система обозначений

Если P и Q — суждения, таковыми же являются $\neg P$ (читается как "не P"), $P \vee Q$ ("P или Q"), $P \wedge Q$ ("P и Q"), $P \Rightarrow Q$ ("из P следует Q") и $P \Leftrightarrow Q$ ("P эквивалентно Q"). Для указания на эквивалентность мы часто пишем "P, если и только если Q".

Если P — суждение и x — переменная, $(\exists x)P$ — суждение (читается "существует x, такой, что P"). Если P — суждение и x — переменная, $(\forall x)P$ — суждение (читается "для всех x, P"); $(\forall x)P \Leftrightarrow (\neg(\exists x)\neg P)$.

Мы используем следующие термины из теории множеств:

$a \in X$ ("a — *элемент* X"),

$X \subset Y$ ("X — *подмножество* Y"),

$\{a_0, \ldots, a_n\}$ ("*конечное множество* с элементами a_0, ... и a_n")

$\{a \in X | P(a)\}$ ("*подмножество* X, для которого соблюдается предикат P")

$X \cup Y$ ("*объединение* X и Y")

$X \cap Y$ ("*пересечение* X и Y")

$X \times Y$ ("*прямое произведение* X и Y")

$f : X \to Y$ ("f *функция* из X в Y")

$f : X_0 \times X_1 \to Y$ ("f функция из произведения X_0 и X_1 в Y")

$x \mapsto \mathcal{E}(x)$ ("x отображает в $\mathcal{E}(x)$", всегда указано вслед за сигнатурой функции).

Замкнутый интервал $[a, b]$ является множеством всех элементов x, таких, что $a \leqslant x \leqslant b$. *Открытый интервал* (a, b) является множеством всех элементов x, таких, что $a < x < b$. *Полуоткрытый справа интервал* $[a, b)$ является множеством всех элементов x, таких, что $a \leqslant x < b$. *Полуоткрытый слева интервал* $(a, b]$ является множеством всех элементов x, таких, что $a < x \leqslant b$. *Полуоткрытый интервал* — используемое нами сокращение для полуоткрытого справа интервала. Эти определения обобщаются на нестрогие упорядочения.

Мы используем следующую систему обозначений в спецификациях, где i и j — итераторы, а n — целое число:

$\mathtt{i} \prec \mathtt{j}$ (“i предшествует j”),
$\mathtt{i} \preceq \mathtt{j}$ (“i предшествует или равняется j”),
$[\mathtt{i}, \mathtt{j})$ (“полуоткрытый ограниченный интервал от i до j”)
$[\mathtt{i}, \mathtt{j}]$ (“закрытый ограниченный интервал от i до j”),
$[\![\mathtt{i}, \mathtt{n})\!)$ (“полуоткрытый нестрогий или счетный интервал от i для $\mathtt{n} \geqslant 0$”),
$[\![\mathtt{i}, \mathtt{n}]\!]$ (“закрытый нестрогий или счетный интервал от i для $\mathtt{n} \geqslant 0$”).

При обсуждении концепций мы используем следующую терминологию.

Нестрогий указывает на ослабление аксиом, в том числе их исключение. Например, при нестрогом упорядочении равенство заменяется эквивалентностью.

Полу — приставка, указывающая на исключение операций. Например, в полугруппе отсутствует обратная операция.

Частичный обозначает ограничение области определения. Например, частичное вычитание (сокращение) $\mathtt{a} - \mathtt{b}$ определено, если $\mathtt{a} \geqslant \mathtt{b}$.

Глава B

Язык программирования

Шон Пэрент (Sean Parent) и Бьярне Страуструп (Bjarne Stroustrup)

В настоящем приложении определяется подмножество C++, используемое в книге. Чтобы упростить синтаксис, мы используем несколько средств библиотеки в качестве встроенных средств. Эти встроенные средства не приведены в настоящем описании подмножества C++, но используют другие функции C++. Само подмножество определено в разделе B.1, а реализация встроенных средств рассматривается в разделе B.2.

B.1. Определение языка

Система обозначений синтаксиса

Используется расширенная форма Бэкуса–Наура, разработанная Никлаусом Виртом (Niklaus Wirth). В [Wirth 1977, стр. 822–823] эта форма описана следующим образом.

> Термин *идентификатор* служит для обозначения *нетерминального символа*, а термин *литерал* обозначает *терминальный символ*. Для краткости более подробное определение терминов *идентификатор* и *символ* не приведено.

```
syntax     = {production}.
production = identifier "=" expression ".".
expression = term {"|" term}.
term       = factor {factor}.
factor     = identifier | literal
               | "(" expression ")"
               | "[" expression "]"
               | "{" expression "}".
literal    = """" character {character} """".
```

Повторение обозначается фигурными скобками, т. е. применяется обозначение {a} вместо є | a | aa | aaa | .... Необязательность выражается квадратными скобками, т. е. применяется обозначение [a] вместо a | є. Круглые скобки служат для группирования, например, применяется обозначение (a | b) c вместо ac | bc. Терминальные символы, т. е. литералы, заключаются в кавычки (а если в качестве литерала появляется сама кавычка, то записывается дважды).

Лексические соглашения

Синтаксис идентификаторов и литералов задают следующие продукции:

```
identifer = (letter | "_") {letter | "_" | digit}.
literal   = boolean | integer | real.
boolean   = "false" | "true".
integer   = digit {digit}.
real      = integer "." [integer] | "." integer.
```

Комментарии распространяются от двух знаков косой черты до конца строки:

```
comment    = "//" {character} eol.
```

Основные типы

Используются три типа C++: Тип `bool` имеет значения `false` и `true`, тип `int` — значения целого числа со знаком, а тип `double` имеет 64-разрядные значения с плавающей точкой по стандарту IEEE:

```
basic_type = "bool" | "int" | "double".
```

Выражения

Выражения могут относиться ко времени выполнения или компиляции. Вычисление выражений времени компиляции может приводить к получению значения или типа.

Выражения определены следующей грамматикой. Операторы во внутренних продукциях (расположенных ниже в определении грамматики) имеют более высокий порядок приоритета по сравнению с таковыми во внешних продукциях:

```
expression      = conjunction {"||" conjunction}.
conjunction     = equality {"&&" equality}.
equality        = relational {("==" | "!=") relational}.
relational      = additive {("<" | ">" | "<=" | ">=") additive}.
additive        = multiplicative {("+" | "-") multiplicative}.
multiplicative  = prefix {("*" | "/" | "%") prefix}.
```

```
prefix          = ["-" | "!" | "const"] postfix.
postfix         = primary {"." identifier
                    | "(" [expression_list] ")"
                    | "[" expression "]"
                    | "&"}.
primary         = literal | identifier | "(" expression ")"
                    | basic_type | template_name | "typename".
expression_list = expression {"," expression}.
```

Операторы `||` и `&&` определяют $\vee$ (дизъюнкцию) и $\wedge$ (конъюнкцию) соответственно. Операнды должны быть булевыми значениями. Первый операнд вычисляется перед вторым операндом. Если первый операнд определяет результат выражения (`true` для `||` или `false` для `&&`), то второй операнд не вычисляется и результатом становится значение первого операнда. Префикс `!` обозначает $\neg$ (отрицание) и должен применяться к булеву значению.

== и != — соответственно операторы проверки на равенство и неравенство, возвращающие булевы значения.

<, >, <=, и >= обозначают соответственно операторы сравнения по условию “меньше “больше “меньше или равно"и “больше или равно которые также возвращают булевы значения.

+ и - — операторы сложения и вычитания; префикс - — аддитивная операция перемены знака.

*, /, и % — соответственно операторы умножения, деления и вычисления остатка.

Постфикс `.` (точка) принимает объект типа структуры и возвращает элемент, соответствующий идентификатору, который следует за точкой. Постфикс `()` принимает процедуру или объект, на которых определен оператор применения, и возвращает результат вызова процедуры или функционального объекта с данными аргументами. Будучи применен к типу, постфикс `()` выполняет построение, используя заданные аргументы; в случае применения к функции типа возвращает другой тип. Постфикс `[]` принимает объект, на котором определен оператор индекса, и возвращает элемент, позиция которого указана согласно значению выражения в квадратных скобках.

Префикс `const` — это оператор типа, возвращающий тип, который является постоянной версией его операнда. Если он применяется к ссылочному типу, результирующим типом становится ссылка на постоянную версию базового типа ссылки.

Постфикс `&` — оператор типа, возвращающий ссылочный тип его операнда.

Перечисления

Перечисление генерирует тип, в котором каждому идентификатору в списке соответствует уникальное значение. Единственными операциями, которые определены на перечислениях, являются операции, относящиеся к регулярным типам: проверка на равенство, реляционные операции, проверка на неравенство, построение, уничтожение и присваивание:

```
enumeration    = "enum" identifier "{" identifer_list "}" ";".
identifer_list = identifier {"," identifier}.
```

Структуры

Структура — это тип, состоящий из разнородного кортежа именованных, типизированных объектов, называемых элементами данных. Каждый элемент данных представляет собой отдельный объект или массив постоянного размера. Кроме того, структура может включать определения конструкторов, деструктора, операторы-члены (операторы присваивания, применения и индексации) и локальные определения типов. Структура с оператором-членом применения известна как *функциональный объект*. Тело структуры может быть исключено, что обеспечивает предваряющее объявление.

```
structure        = "struct" structure_name [structure_body] ";".
structure_name   = identifier.
structure_body   = "{" {member} "}".
member           = data_member
                     | constructor | destructor
                     | assign | apply | index
                     | typedef.
data_member      = expression identifier ["[" expression "]"] ";".
constructor      = structure_name "(" [parameter_list] ")"
                     [":" initializer_list] body.
destructor       = "~" structure_name "(" ")" body.
assign           = "void" "operator" "="
                     "(" parameter ")" body.
apply            = expression "operator" "(" ")"
                     "(" [parameter_list] ")" body.
index            = expression "operator" "[" "]"
                     "(" parameter ")" body.

initializer_list = initializer {"," initializer}.
initializer      = identifer "(" [expression_list] ")".
```

Конструктором, принимающим постоянную ссылку на тип структуры, является *конструктор копии*. Если конструктор копии не определен, создается

конструктор поэлементного копирования. Конструктором без аргументов является *стандартный конструктор*. Стандартный поэлементный конструктор создается, только если не определены какие-либо другие конструкторы. Если оператор присваивания не определен, создается оператор поэлементного присваивания. Если деструктор вообще не предусмотрен, создается поэлементный деструктор. Каждый идентификатор в списке инициализатора представляет собой идентификатор элемента данных структуры. Если конструктор содержит список инициализатора, каждый элемент данных структуры создается с помощью конструктора, согласующегося[1] со списком выражений инициализатора; все эти построения происходят перед выполнением тела конструктора.

Процедуры

Определение процедуры состоит из относящегося к ней обозначения типа возврата или `void` (если не происходит возврат какого-либо значения), за которым следуют имя процедуры и список параметров. Имя может представлять собой идентификатор или оператор. Применение выражения параметра должно приводить к получению типа. Исключение тела процедуры обеспечивает предваряющее объявление.

```
procedure      = (expression | "void") procedure_name
                   "(" [parameter_list] ")" (body | ";").
procedure_name = identifier | operator.
operator       = "operator"
                   ("==" | "<" | "+" | "-" | "*" | "/" | "%").
parameter_list = parameter {"," parameter}.
parameter      = expression [identifier].
body           = compound.
```

Могут быть определены только перечисленные операторы. Определение для оператора != создается на основе определения оператора ==; определения для операторов >, <= и >= создаются на основе определения оператора <. При вызове процедуры значение каждого выражения аргумента связывается с соответствующим параметром, и выполняется тело процедуры.

Инструкции

Инструкции составляют тело процедур, конструкторов, деструкторов и операторов-членов:

```
statement          = [identifier ":"]
                        (simple_statement | assignment
```

[1] Механизм согласования выполняет разрешение перегрузки путем точного согласования без каких-либо неявных преобразований.

```
                    | construction | control_statement
                    | typedef).
simple_statement  = expression ";".
assignment        = expression "=" expression ";".
construction      = expression identifier [initialization] ";".
initialization    = "(" expression_list ")" | "=" expression.
control_statement = return | conditional | switch | while | do
                    | compound | break | goto.
return            = "return" [expression] ";".
conditional       = "if" "(" expression ")" statement
                    ["else" statement].
switch            = "switch" "(" expression ")" "{" {case} "}".
case              = "case" expression ":" {statement}.
while             = "while" "(" expression ")" statement.
do                = "do" statement
                    "while" "(" expression ")" ";".
compound          = "{" {statement} "}".
break             = "break" ";".
goto              = "goto" identifier ";".
typedef           = "typedef" expression identifier ";".
```

Простая инструкция, которая часто представляет собой вызов процедуры, вычисляется для получения ее побочных эффектов. Операция присваивания сводится к тому, что применяется оператор присваивания, относящийся к типу объекта, который находится слева от знака этой операции. Первым выражением для построения является выражение типа, задающее создаваемый тип. Для построения без инициализации применяется стандартный конструктор. Для построения при наличии заключенного в скобки списка выражений применяется соответствующий конструктор. Для построения при наличии знака равенства, за которым следует выражение, применяется конструктор копии; выражение должно иметь такой же тип, как и создаваемый объект.

Инструкция `return` возвращает управление вызывающему объекту текущей функции со значением выражения как результатом функции. Вычисление выражения должно приводить к получению значения с возвращаемым типом функции.

Условный оператор выполняет первую инструкцию, если значение выражения равно true; если выражение равно false и имеется предложение `else`, выполняется вторая инструкция. Вычисление выражения должно приводить к получению булева значения.

Инструкция `switch` вычисляет выражение, а затем выполняет первую инструкцию после метки выбора с совпадающим значением; последующие инструкции выполняются до конца инструкции `switch` или до выполнения инструкции `break`. Вычисление выражения в инструкции `switch` должно приводить к получению целого числа или перечисления.

Инструкция `while` неоднократно вычисляет выражение и выполняет инструкцию до тех пор, пока выражение равно true. Инструкция `do` неоднократно выполняет инструкцию и вычисляет выражение до тех пор, пока выражение не становится равным false. В любом случае вычисление выражения должно приводить к получению булева значения.

Составной оператор выполняет последовательность инструкций в определенном порядке.

Инструкция `goto` передает выполнение той инструкции, которая следует за соответствующей меткой в текущей функции.

Инструкция `break` завершает выполнение ближайшей охватывающей инструкции `switch`, `while` или `do`; выполнение продолжается с инструкции, которая следует за завершенной инструкцией.

Инструкция `typedef` определяет псевдоним для типа.

Шаблоны

Шаблон обеспечивает параметризацию структуры или процедуры на основе одного или нескольких типов или констант. В определениях и именах шаблонов знаки < и > используются как разделители[2].

```
template       = template_decl
                   (structure | procedure | specialization).
specialization = "struct" structure_name "<" additive_list ">"
                   [structure_body] ";".
template_decl  = "template" "<" [parameter_list] ">" [constraint].
constraint     = "requires" "(" expression ")".

template_name  = (structure_name | procedure_name)
                   ["<" additive_list ">"].
additive_list  = additive {"," additive}.
```

Если `template_name` используется в качестве основного, определение шаблона служит для создания структуры или процедуры с параметрами шаблона, замененными соответствующими аргументами шаблона. Эти аргументы шаблона задаются явно как список выражений с разграничителями в `template_name` или, применительно к процедурам, могут быть выведены из типов аргументов процедуры.

Структура шаблона может быть специализирована, обеспечивая альтернативное определение для шаблона, который рассматривается, когда аргу-

[2] Для снятия неопределенности относительно того, используются ли < и > как отношения или как разделители в именах шаблонов, после синтаксического анализа `structure_name` или `procedure_name` в составе `template` эти знаки преобразуются в терминальные символы.

менты согласуются до согласования неспециализированной версии структуры шаблона.

Если определение шаблона включает ограничение, типы и значения аргументов шаблона должны удовлетворять булеву выражению после `requires`.

Встроенные средства

`pointer(T)` — конструктор типа, который возвращает указатель типа на `T`. Если `x` — объект типа `T`, то `addressof(x)` возвращает значение типа `pointer(T)`, ссылающееся на `x`. `source`, `sink` и `deref` — унарные функции, определяемые на типах указателей. Функция `source` определена для всех типов указателей и возвращает соответствующую постоянную ссылку (см. раздел 6). Функции `sink` и `deref` определены для типов указателей на непостоянные объекты и возвращают соответствующие непостоянные ссылки (см. раздел 9). `reinterpret_cast` — шаблон функции, который принимает ссылочный тип и объект (передаваемый по ссылке) и возвращает ссылку ссылочного типа на тот же объект. Объект должен также иметь допустимую интерпретации со ссылочным типом.

B.2. Макросы и характеристические структуры

Чтобы язык, определенный в разделе B.1, можно было компилировать как допустимую программу C++, необходимы некоторые макросы и определения структур.

Ограничения шаблонов

Предложение `requires` реализовано с помощью следующего макроса[3]:

```
#define requires(...)
```

Встроенные средства

`pointer(T)` и `addressof(x)` введены, чтобы предоставить пользователям простую линейную систему обозначений и обеспечить простой нисходящий синтаксический анализ. Они реализованы как

```
#define pointer(T) T*

template<typename T>
pointer(T) addressof(T& x)
```

[3] В этой реализации требования трактуются только как документация.

```
{
    return &x;
}
```

Функции типа

Функции типа реализованы с использованием средства C++, называемого *характеристическим классом*. Для каждой функции типа (скажем, ValueType) мы определяем соответствующий шаблон структуры: допустим, `value_type<T>`. Шаблон структуры содержит одно определение типа, в соответствии с принятым соглашением, называемое `type`; если это уместно, то в основном шаблоне структуры может быть предоставлено стандартное значение:

```
template<typename T>
struct value_type
{
    typedef T type;
};
```

Чтобы обеспечить удобную систему обозначений, мы определяем макрос[4], который извлекает определение типа как результат функции типа:

```
#define ValueType(T) typename value_type< T >::type
```

Уточняем глобальное определение для конкретного типа, специализируя следующее:

```
template<typename T>
struct value_type<pointer(T)>
{
    typedef T type;
};
```

[4]Такой макрос работает только в определении шаблона, поскольку используется ключевое слово `typename`.

Литература

Agarwal, Saurabh and Gudmund Skovbjerg Frandsen. 2004. Binary GCD like algorithms for some complex quadratic rings. In *Algorithmic Number Theory, 6th International Symposium, Burlington, VT, USA, June 13–18, 2004. Proceedings*, ed. Duncan A. Buell, vol. 3076 of *Lecture Notes in Computer Science*, p. 57–71. Springer.

Bentley, Jon. 1984. Programming pearls. *Communications of the ACM* 27(4), p. 287–291.

Bolzano, Bernard. 1817. *Rein analytischer Beweis des Lehrsatzes, daß zwischen je zwey Werthen, die ein entgegengesetztes Resultat gewhren, wenigstens eine reelle Wurzel der Gleichung liege*. Prague: Gottlieb Haase.

Boute, Raymond T. 1992. The Euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems* 14(2): 127–144.

Boyer, Robert S. and J. Strother Moore. 1977. A fast string searching algorithm. *Communications of the ACM* 20(10), p. 762–772.

Brent, Richard P. 1980. An improved Monte Carlo factorization algorithm. *BIT* 20: 176–184.

Cauchy, Augustin-Louis. 1821. *Cours D'Analyse de L'Ecole Royale Polytechnique*. L'Académie des Sciences.

Chrystal, G. 1904. *Algebra: An Elementary Text-Book. Parts I and II*. Adam and Charles Black, 1904. Reprint, AMS Chelsea Publishing, 1964.

Dehnert, James C. and Alexander A. Stepanov. 2000. Fundamentals of generic programming. In *Generic Programming, International Seminar on Generic Programming, Dagstuhl Castle, Germany, April/May 1998. Selected Papers*, eds. Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser, vol. 1766 of *Lecture Notes in Computer Science*, p. 1–11. Springer.

Diaconis, Persi and Paul Erdös. 2004. On the distribution of the greatest common divisor. In *A Festschrift for Herman Rubin*, ed. Anirban DasGupta, vol. 45 of *Lecture Notes—Monograph Series*, p. 56–61. Institute of Mathematical Statistics.

Dijkstra, Edsger W. 1972. Notes on structured programming. In *Structured Programming*, eds. O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, p. 1–82. London and New York: Academic Press.

Dirichlet, P. G. L. 1863. *Forlesungen über Zahlentheorie*. Vieweg und Sohn, 1863. С дополнениями Рихарда Дедекинда. Перевод на английский язык John Stillwell. *Lectures on Number Theory*, American Mathematical Society and London Mathematical Society, 1999.

Dudziński, Krzysztof and Andrzej Dydek. 1981. On a stable minimum storage merging algorithm. *Information Processing Letters* 12(1), p. 5–8.

Dwyer, Barry. 1974. Simple algorithms for traversing a tree without an auxiliary stack. *Information Processing Letters* 2: 143–145.

Fiduccia, Charles M. 1985. An efficient formula for linear recurrences. *SIAM Journal on Computing* 14(1), p. 106–112.

Fletcher, William and Roland Silver. 1966. Algorithm 284: Interchange of two blocks of data. *Communications of the ACM* 9(5), p. 326.

Floyd, Robert W. and Knuth, Donald E. 1990. Addition machines. *SIAM Journal on Computing* 19(2), p. 329-340.

Frobenius, Georg Ferdinand. 1895. Über endliche gruppen. In *Sitzungesberichte der Königlich Preussischen Akademie der Wissenschaften zu Berlin*, Phys.-math. Classe, p. 163–194. Berlin.

Grassmann, Hermann Günther. 1861. *Lehrbuch der Mathematik für höhere Lehranstalten*, vol. 1. Berlin: Enslin.

Gries, David and Harlan Mills. 1981. Swapping sections. Tech. Rep. 81-452, Department of Computer Science, Cornell University.

Heath, Sir Thomas L. 1925. *The Thirteen Books of Euclid's Elements*. Cambridge University Press, 1925. Reprint, Dover, 1956.

Heath, T. L. 1912. *The Works of Archimedes*. Cambridge University Press, 1912. Reprint, Dover, 2002.

Hoare, C. A. R. 1962. Quicksort. *The Computer Journal* 5(1), p. 10–16.

Iverson, Kenneth. 1962. *A Programming Language*. Wiley.

Knuth, Donald E. 1997. *The Art of Computer Programming Volume 2: Seminumerical Algorithms (3rd edition)*. Reading, MA: Addison-Wesley.

Knuth, Donald E. 1998. *The Art of Computer Programming Volume 3: Sorting and Searching (2nd edition)*. Reading, MA: Addison Wesley.

Knuth, Donald E. 2005. *The Art of Computer Programming Volume 1, fascicle 1: MMIX: A RISC Computer for the New Millenium*. Boston: Addison-Wesley.

Knuth, Donald E., J. Morris, and V. Pratt. 1977. Fast pattern matching in strings. *SIAM Journal on Computing* 6: 323–350.

Kwak, Jin Ho and Sungpyo Hong. 2004. *Linear Algebra*. Birkhäuser.

Lagrange, J.-L. 1795. *Leçons élémentaires sur les mathématiques, données à l'école normale en 1795*. 1795. Reprinted: *Oeuvres*, vol. VII, p. 181–288. Paris: Gauthier-Villars, 1877.

Levy, Leon S. 1982. An improved list-searching algorithm. *Information Processing Letters* 15(1), p. 43–45.

Lindstrom, Gary. 1973. Scanning list structures without stack or tag bits. *Information Processing Letters* 2: 47–51.

Mauchly, John W. 1946. Sorting and collating. In *Theory and Techniques for Design of Electronic Digital Computers*. Moore School of Electrical Engineering, University of Pennsylvania, 1946. Reprinted in: *The Moore School Lectures*, eds. Martin Campbell-Kelly and Michael R. Williams, p. 271–287. Cambridge, Massachusetts: MIT Press, 1985.

McCarthy, D. P. 1986. Effect of improved multiplication efficiency on exponentation algorithms derived from addition chains. *Mathematics of Computation* 46(174), p. 603–608.

Miller, J. C. P. and D. J. Spencer Brown. 1966. An algorithm for evaluation of remote terms in a linear recurrence sequence. *The Computer Journal* 9(2), p. 188–190.

Morris, Joseph M. 1979. Traversing binary trees simply and cheaply. *Information Processing Letters* 9(5), p. 197–200.

Musser, David R. 1975. Multivariate polynomial factorization. *Journal of the ACM* 22(2), p. 291–308.

Musser, David R. and Gor V. Nishanov. 1997. A fast generic sequence matching algorithm. Tech. Rep., Computer Science Department, Rensselaer Polytechnic Institute. Помещена в архив по адресу http://arxiv.org/abs/0810.0264v1.

Patterson, David A. and John L. Hennessy. 2007. *Computer Organization and Design: The Hardware/Software Interface (3rd revised edition)*. Morgan Kaufmann.

Peano, Giuseppe. 1908. *Formulario Mathematico, Editio V*. Torino: Fratres Bocca Editores, 1908. Reprinted: Roma: Edizioni Cremonese, 1960.

Rivest, R., A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21(2), p. 120–126.

Robins, Gay and Charles Shute. 1987. *The Rhind Mathematical Papyrus*. British Museum Publications.

Robson, J. M. 1973. An improved algorithm for traversing binary trees without auxiliary stack. *Information Processing Letters* 2: 12–14.

Schorr, H. and W. M. Waite. 1967. An efficient and machine-independent procedure for garbage collection in various list structures. *Communications of the ACM* 10(8), p. 501–506.

Sedgewick, R. T., T. G. Szymanski, and A. C. Yao. 1982. The complexity of finding cycles in periodic functions. In *SIAM Journal on Computing* 11(2): p. 376-390.

Sigler, Laurence E. 2002. *Fibonacci's Liber Abaci: Leonardo Pisano's Book of Calculation*. Springer-Verlag.

Stein, Josef. 1967. Computational problems associated with Racah algebra. *J. Comput. Phys.* 1: 397–405.

Stepanov, Alexander and Meng Lee. 1995. The Standard Template Library. Technical Report 95-11(R.1), HP Laboratories.

Stroustrup, Bjarne. 2000. *The C++ Programming Language: Special Edition (3rd edition)*. Boston: Addison-Wesley.

Tarjan, Robert Endre. 1983. *Data Structures and Network Algorithms*. SIAM.

Tarjan, Robert Endre. 1985. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods* 6(2), p. 306–318.

Thompson, Ken and Dennis Ritchie. 1974. The UNIX time-sharing system. *Communications of the ACM* 17(7), p. 365–375.

van der Waerden, Bartel Leenert. 1930. *Moderne Algebra Erster Teil*. Julius Springer, 1930. Перевод на английский язык Fred Blum. *Modern Algebra*, New York: Frederic Ungar Publishing, 1949.

Weilert, André. 2000. (1+i)-ary GCD computation in $\mathbb{Z}[i]$ as an analogue of the binary GCD algorithm. *J. Symb. Comput.* 30(5), p. 605–617.

Wirth, Niklaus. 1977. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM* 20(11), p. 822–823.

Предметный указатель

Н

О

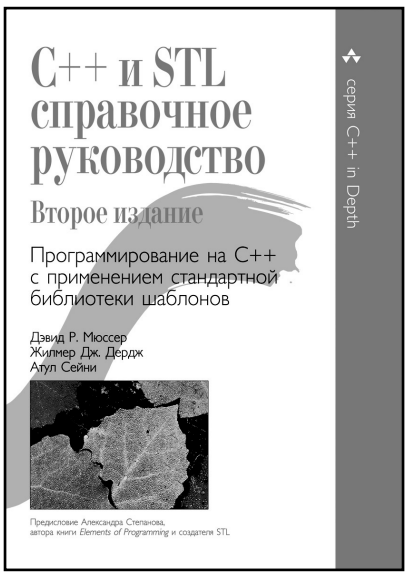
П

P

C

C++ И STL
СПРАВОЧНОЕ РУКОВОДСТВО
2-е издание

Дэвид Р. Мюссер,
Жилмер Дж. Дердж,
Атул Сейни



www.williamspublishing.com

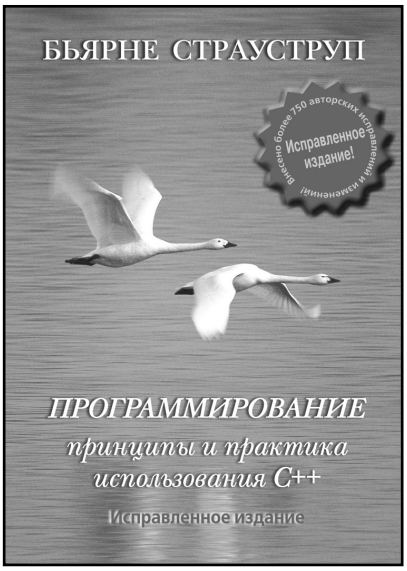
Написанная авторами, принимавшими участие в разработке и практическом применении STL, данная книга представляет собой полное справочное руководство по данной теме. Она включает небольшой учебный курс, подробное описание каждого элемента библиотеки и большое количество примеров.

В книге вы найдете подробное описание итераторов, обобщенных алгоритмов, контейнеров, функциональных объектов и т.д. Ряд нетривиальных приложений демонстрирует использование мощи и гибкости STL в повседневной работе программиста. Книга также разъясняет, как интегрировать STL с другими объектно-ориентированными методами программирования. Она будет вашим постоянным спутником и советчиком при работе над проектами любой степени сложности.

ISBN 978-5-8459-1665-5 **в продаже**

ПРОГРАММИРОВАНИЕ
принципы и практика использования C++
Исправленное издание

Бьярне Страуструп



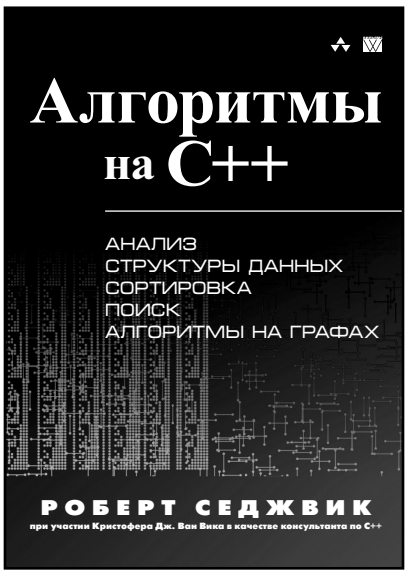
www.williamspublishing.com

Это учебник по программированию, написанный автором языка C++. Книга задумана как вводный курс по программированию с демонстрацией примеров программ, написанных на языке C++, и описывает широкий круг понятий и приемов объектно-ориентированного и обобщенного программирования, необходимых для того, чтобы стать профессиональным программистом.
В частности, в ней рассмотрены основы вычислений на языке C++, приемы обработки ошибок, система ввода и вывода данных, методы работы с графикой, стандартная библиотека шаблонов, методы обработки текста, основы разработки встроенных систем, приемы тестирования. Синтаксис языка C++ описан в приложении. Это отражает точку зрения автора на то, чему посвящена книга: в первую очередь — программированию, а во вторую — языку C++.
В первую очередь, книга адресована начинающим программистам, но она будет полезна и профессионалам, которые найдут в ней много новой информации, а главное, смогут узнать точку зрения изобретателя языка C++ на современные методы программировании.

ISBN 978-5-8459-1705-8 **в продаже**

АЛГОРИТМЫ НА C++
АНАЛИЗ, СТРУКТУРЫ ДАННЫХ, СОРТИРОВКА, ПОИСК, АЛГОРИТМЫ НА ГРАФАХ

Роберт Седжвик



www.williamspublishing.com

Эта классическая книга удачно сочетает в себе теорию и практику, что делает ее популярной у программистов на протяжении многих лет. Кристофер Ван Вик и Роберт Седжвик разработали новые лаконичные реализации на C++, которые естественным и наглядным образом описывают методы и могут применяться в реальных приложениях. Каждая часть содержит новые алгоритмы и реализации, усовершенствованные описания и диаграммы, а также множество новых упражнений для лучшего усвоения материала. Акцент на АТД расширяет диапазон применения программ и лучше соотносится с современными средами объектно-ориентированного программирования. Книга предназначена для широкого круга разработчиков и студентов.

ISBN 978-5-8459-1650-1

в продаже

C++: БАЗОВЫЙ КУРС
ТРЕТЬЕ ИЗДАНИЕ

Герберт Шилдт

В этой книге описаны все основные средства языка C++: от элементарных понятий до супервозможностей. После рассмотрения основ программирования на C++ (переменных, операторов, инструкций управления, функций, классов и объектов) читатель освоит такие более сложные средства языка, как механизм обработки исключительных ситуаций (исключений), шаблоны, пространства имен, динамическая идентификация типов, стандартная библиотека шаблонов (STL), а также познакомится с расширенным набором ключевых слов, используемым в программировании для .NET. Автор справочника — общепризнанный авторитет в области программирования на языках C и C++, Java и C# — включил в текст своей книги и советы программистам, которые позволят повысить эффективность их работы.

www.williamspublishing.com

ISBN 978-5-8459-0768-4 **в продаже**

VISUAL C++ 2010 ПОЛНЫЙ КУРС

Айвор Хортон

www.dialektika.com

По существу, в этой книге рассматриваются две обширные темы: язык программирования C++ и программирование приложений Windows с использованием MFC или .NET Framework. Прежде чем вы сможете разработать полнофункциональное приложение Windows, необходимо приобрести хороший уровень знаний языка C++, поэтому упражнения здесь на первом месте.

В первой части книги поэтапно изложены основные темы программирования на языке C++. Вы изучите синтаксис и использование базового языка C++, а также приобретете уверенность и опыт применения его на практике. Модификацию C++/CLI базового языка C++ вы также изучите на практических примерах.

Кроме того, вы узнаете о мощных инструментальных средствах, предоставляемых стандартной библиотекой шаблонов STL, для базовой версии языка C++ и версии C++/CLI. Одна из глав посвящена библиотеке шаблонов для параллельных вычислений, которая позволяет использовать мощь многоядерных PC для приложений с интенсивными вычислениями.

ISBN 978-5-8459-1698-3 **в продаже**

C# 4 И ПЛАТФОРМА .NET 4 ДЛЯ ПРОФЕССИОНАЛОВ

Кристиан Нейгел
Билл Ивьен
Джей Глинн
Карли Уотсон
Морган Скиннер

www.dialektika.com

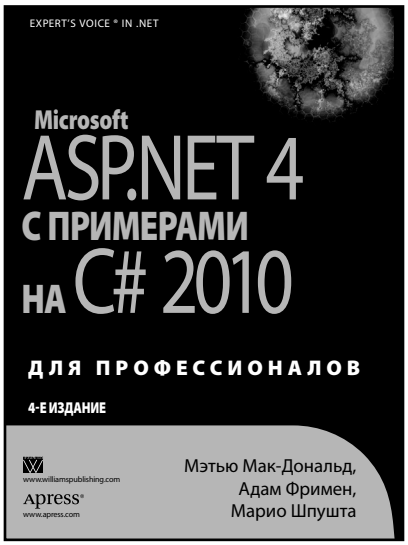
Книга известных специалистов в области разработки приложений с использованием .NET Framework посвящена программированию на языке C# 2010 в среде .NET Framework 4 и в предшествующих версиях. Книгу отличает простой и доступный стиль изложения, изобилие примеров и рекомендаций по написанию высококачественных программ. Подробно рассматриваются такие вопросы, как основы языка программирования C#, организация среды .NET, работа с данными, написание Windows- и веб-приложений, взаимодействие через сеть, создание веб-служб и многое другое. Немалое внимание уделено проблемам безопасности и сопровождения кода. Тщательно подобранный материал позволит без труда разобраться с тонкостями использования Windows Forms и построения веб-страниц. Читатели ознакомятся с работой в Visual Studio 2010, а также с применением различных технологий, встроенных в .NET. Книга рассчитана на программистов разной квалификации.

ISBN 978-5-8459-1656-3

в продаже

MICROSOFT ASP.NET 4 С ПРИМЕРАМИ НА C# 2010
ДЛЯ ПРОФЕССИОНАЛОВ
4-Е ИЗДАНИЕ

Мэтью Мак-Дональд
Адам Фримен
Марио Шпушта



www.williamspublishing.com

Книга известных специалистов в области технологий .NET представляет собой учебное и справочное пособие для разработчиков .NET-приложений, использующих новую версию ASP.NET 4. Настоящее издание было полностью обновлено и дополнено с учетом последней версии ASP.NET, и теперь включает описание ASP. NET MVC, ASP.NET AJAX 4, ASP.NET Dynamic Data и Silverlight 3. Предложенный авторами практический подход к изложению материала не является простым повторением документации, а позволяет сконцентрироваться на решении конкретных задач, связанных с разработкой веб-приложений разного уровня сложности. Глубина изложения материала превращает эту книгу в незаменимый источник информации для разработчиков приложений ASP.NET 4.

ISBN 978-5-8459-1702-7 **в продаже**

C# 4.0
ПОЛНОЕ РУКОВОДСТВО

Герберт Шилдт

www.williamspublishing.com

В этом полном руководстве по C# 4.0 — языку программирования, разработанному специально для среды .NET, — детально рассмотрены все основные средства языка: типы данных, операторы, управляющие операторы, классы, интерфейсы, методы, делегаты, индексаторы, события, указатели, обобщения, коллекции, основные библиотеки классов, средства многопоточного программирования и директивы препроцессора. Подробно описаны новые возможности C#, в том числе PLINQ, библиотека TPL, динамический тип данных, а также именованные и необязательные аргументы. Это справочное пособие снабжено массой полезных советов авторитетного автора и сотнями примеров программ с комментариями, благодаря которым они становятся понятными любому читателю независимо от уровня его подготовки. Книга рассчитана на широкий круг читателей, интересующихся программированием на C#.

ISBN 978-5-8459-1684-6 **в продаже**

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 2010 И ПЛАТФОРМА .NET 4

5-е издание

Эндрю Троелсен

www.williamspublishing.com

Версия .NET 4 привнесла множество новых API-интерфейсов в библиотеках базовых классов, а также новых синтаксических конструкций в языке C#.
В этой книге вы найдете полное описание всех нововведений в характерной для автора дружественной к читателю манере. Помимо общих вопросов, подробно рассматривается среда Dynamic Language Runtime (DLR); библиотека Task Parallel Library (TPL, включая PLINQ); технология ADO.NET Entity Framework (а также LINQ to EF); расширенное описание API-интерфейса Windows Presentation Foundation (WPF); улучшенная поддержка взаимодействия с COM.
В книге рассматриваются следующие темы

- Особенности платформы .NET 4 и языка Visual C# 2010
- Детали технологии .NET – лидера в производстве современного программного обеспечения
- Полезные советы по разработке от эксперта в .NET, который изучает эту платформу, начиная с ее первой версии
- Полное описание технологий WPF, WCF и WF, поддерживаемых ядром платформы .NET

ISBN 978-5-8459-1682-2

в продаже